URL Shortener (Microservices) — Engineering Design Document

System Purpose

A production-grade URL shortener decomposed into four independently deployable services, each owning its bounded context and data store. The design prioritizes correct service boundary enforcement and async event flow over feature breadth — making the architectural decisions the primary deliverable, not the product functionality.

Service Map

Service	Port	Responsibility	DB	Owns Events
api-gateway	8080	Routing, JWT validation, rate limiting, circuit breaker	None (Redis for rate limits)	None
url-service	8081	Shorten, redirect, CRUD, outbox polling	url_db (PostgreSQL) + Redis cache	URLCreatedEvent, URLClickedEvent, URLDeletedEvent
analytics-service	8082	Click ingestion, stats queries, milestone detection	analytics_db (PostgreSQL)	MilestoneReachedEvent
user-service	8083	Registration, login, JWT issuance	user_db (PostgreSQL)	None
notification-service	8084	Notification persistence, delivery logging	notification_db (PostgreSQL)	None

Tech Stack

- **Language:** Go (net/http, pgxpool, crypto/rand)
- **Message broker:** RabbitMQ (AMQP 0-9-1, topic exchange)
- **Databases:** PostgreSQL 16 — one container per service
- **Cache / rate-limit store:** Redis 7 (shared infrastructure, not a service DB)
- **Containerization:** Docker Compose with healthcheck-gated dependency ordering
- **Auth:** HS256 JWT; verified locally in each service, issued only by user-service

Key Design Decisions

- **Outbox pattern for event delivery:** URL Service writes events to an `outbox` table in the same transaction as the URL mutation. A worker pool polls and publishes to RabbitMQ. This eliminates the dual-write failure window where a service could commit a URL row but fail to publish the corresponding event.
- **Async click analytics:** URLClickedEvent is published asynchronously via the outbox rather than calling analytics-service synchronously on the redirect hot path. This decouples redirect latency from analytics availability entirely.
- **Redis as cache, not as a service database:** All durable writes go to PostgreSQL first. Redis holds read copies with TTL for the redirect hot path and token-bucket counters for gateway rate limiting. Redis unavailability degrades gracefully — it never causes a write failure.
- **Stateless JWT verification:** Every service verifies tokens locally using the shared secret. No per-request call to user-service. The gateway is the first verification point; downstream services verify independently as a defense layer.
- **Gateway enforces cross-cutting concerns only:** JWT validation, rate limiting, circuit breaking, and correlation ID injection live in the gateway. Zero domain logic crosses into it — no URL ownership checks, no stats computation.

Anti-Patterns Explicitly Avoided

Anti-Pattern	Mitigation
Shared Database	Four separate PostgreSQL containers; cross-service data flows via events or API calls only
Synchronous Call Chain	Gateway routes to exactly one downstream service; analytics reads from queue, never calls url-service
God Service	Each service maps to exactly one bounded context; auth, URL lifecycle, analytics, and notifications are split
Chatty Service	Analytics trusts event payloads; it does not call back to url-service to validate `short_code` on ingestion

TDD

Five independently deployable Go binaries communicating via HTTP/JSON (sync) and RabbitMQ (async), each with a private PostgreSQL schema. An API Gateway enforces cross-cutting concerns without business logic. The outbox pattern guarantees at-least-once event delivery. Redis serves dual roles: read-through cache in url-service and token-bucket rate-limit store in the gateway. Circuit breaker, correlation IDs, and structured JSON logging provide production-grade observability.

Technical Design Specification

Module: Foundation — Repo Layout, Shared Contracts, Local Dev Stack

Module ID: `url-shortener-m1`

1. Module Charter

This module establishes every piece of structural infrastructure that subsequent milestones build upon: the monorepo directory layout, Go module declarations per service, shared domain event contracts, Docker Compose topology with full healthcheck wiring, and connection bootstrapping (PostgreSQL via pgxpool, Redis, RabbitMQ) in each service's `main.go`. It also declares the RabbitMQ exchange and queue topology that later milestones publish into and consume from. This module does **not** implement any business logic, authentication, URL shortening, analytics ingestion, notification delivery, or message routing. No HTTP handler beyond `/health` is written. No SQL schema migrations are created (those belong to M2–M5). No JWT middleware, no outbox poller, no consumer goroutines. **Upstream dependencies:** None — this is the foundation layer. **Downstream dependencies:** Every subsequent milestone (M2–M5) imports `shared/events`, `shared/logger`, and relies on the Docker Compose topology defined here. **Invariants that must always hold after this milestone:**

- Every service owns exactly one PostgreSQL container; no two services share a container or schema.
- Redis is reachable by url-service but is treated as optional: `redis.Client` initialization failure must not prevent startup.
- RabbitMQ exchange `url-shortener` (topic) exists before any service attempts to publish or consume; declaration is idempotent.
- Every service returns HTTP 200 on `GET /health` within 10ms once healthy.
- `docker compose up` brings all containers to the healthy state within 60 seconds on commodity hardware.

2. File Structure

Create files in the numbered order below. Parent directories that do not yet exist must be created first.

```
url-shortener/                        # repo root
│
├── 01  docker-compose.yml
├── 02  .env.example
├── 03  README.md
│
├── shared/
│   ├── 04  events/
│   │   └── 05  events.go             # DomainEvent interface + all event structs
│   └── 06  logger/
│       └── 07  logger.go            # Structured JSON logger (zerolog wrapper)
│
├── services/
│   ├── url-service/
│   │   ├── 08  go.mod
│   │   ├── 09  go.sum                # generated by `go mod tidy`
│   │   ├── 10  main.go
│   │   ├── 11  Dockerfile
│   │   └── handler/
│   │       └── 12  health.go
│   │
│   ├── analytics-service/
│   │   ├── 13  go.mod
│   │   ├── 14  go.sum
│   │   ├── 15  main.go
│   │   ├── 16  Dockerfile
│   │   └── handler/
│   │       └── 17  health.go
│   │
│   ├── user-service/
│   │   ├── 18  go.mod
│   │   ├── 19  go.sum
│   │   ├── 20  main.go
│   │   ├── 21  Dockerfile
│   │   └── handler/
│   │       └── 22  health.go
│   │
│   └── notification-service/
│       ├── 23  go.mod
│       ├── 24  go.sum
│       ├── 25  main.go
│       ├── 26  Dockerfile
│       └── handler/
│           └── 27  health.go
│
└── gateway/
    ├── 28  go.mod
    ├── 29  go.sum
    ├── 30  main.go
    ├── 31  Dockerfile
    └── handler/
        └── 32  health.go
```

Total files to create: 32 (excluding auto-generated `go.sum` files).

3. Complete Data Model

3.1 Domain Event Structs (`shared/events/events.go`)

All domain events share a common interface and a set of concrete structs. Every struct is defined here regardless of which service produces or consumes it — this is the single source of truth for event schema.

GO

```go
package events

import "time"

// DomainEvent is the interface satisfied by every event struct.

// EventType returns the routing key used on the RabbitMQ topic exchange.

// OccurredAt returns the wall-clock time the domain action happened.

type DomainEvent interface {

    GetEventType() string

    GetOccurredAt() time.Time

}

// EventType constants — used as RabbitMQ routing keys.

// Pattern: <aggregate>.<action> (dot-separated, lower-case).

const (

    EventTypeURLCreated       = "url.created"

    EventTypeURLClicked       = "url.clicked"

    EventTypeURLDeleted       = "url.deleted"

    EventTypeMilestoneReached = "milestone.reached"

)

// URLCreatedEvent is published by url-service when a short URL is created.

// CorrelationID is propagated from the HTTP request's X-Correlation-ID header

// so that async log traces can be stitched together.

type URLCreatedEvent struct {

    EventType     string    `json:"event_type"`      // always "url.created"

    OccurredAt    time.Time `json:"occurred_at"`     // RFC3339Nano

    CorrelationID string    `json:"correlation_id"`  // UUID v4; propagated from HTTP header

    ShortCode     string    `json:"short_code"`      // 7-char base62 code

    OriginalURL   string    `json:"original_url"`    // full destination URL

    UserID        string    `json:"user_id"`         // UUID string; owner

    UserEmail     string    `json:"user_email"`      // included so consumers need no callback

    ExpiresAt     *time.Time `json:"expires_at,omitempty"` // nil = no expiry

}

func (e URLCreatedEvent) GetEventType() string    { return e.EventType }

func (e URLCreatedEvent) GetOccurredAt() time.Time { return e.OccurredAt }

// URLClickedEvent is published by url-service on every redirect (via outbox).

// IPHash is SHA-256(raw_ip + salt) — never the raw IP address.

type URLClickedEvent struct {

    EventType     string    `json:"event_type"`      // always "url.clicked"

    OccurredAt    time.Time `json:"occurred_at"`

    CorrelationID string    `json:"correlation_id"`

    EventID       string    `json:"event_id"`        // UUID v4; used for idempotency dedup

    ShortCode     string    `json:"short_code"`
```

```
    UserID        string    `json:"user_id"`         // owner of the short URL (not clicker)
    IPHash        string    `json:"ip_hash"`         // SHA-256(ip+salt), hex encoded
    UserAgent     string    `json:"user_agent"`
    Referer       string    `json:"referer,omitempty"`
}
func (e URLClickedEvent) GetEventType() string    { return e.EventType }
func (e URLClickedEvent) GetOccurredAt() time.Time { return e.OccurredAt }
// URLDeletedEvent is published by url-service when a URL is soft-deleted.
type URLDeletedEvent struct {
    EventType     string    `json:"event_type"`      // always "url.deleted"
    OccurredAt    time.Time `json:"occurred_at"`
    CorrelationID string    `json:"correlation_id"`
    ShortCode     string    `json:"short_code"`
    UserID        string    `json:"user_id"`
    UserEmail     string    `json:"user_email"`      // avoid cross-service lookup
}
func (e URLDeletedEvent) GetEventType() string    { return e.EventType }
func (e URLDeletedEvent) GetOccurredAt() time.Time { return e.OccurredAt }
// MilestoneReachedEvent is published by analytics-service when a click
// threshold (10, 100, 1000) is crossed.
type MilestoneReachedEvent struct {
    EventType     string    `json:"event_type"`      // always "milestone.reached"
    OccurredAt    time.Time `json:"occurred_at"`
    CorrelationID string    `json:"correlation_id"`
    ShortCode     string    `json:"short_code"`
    UserID        string    `json:"user_id"`
    UserEmail     string    `json:"user_email"`
    Milestone     int       `json:"milestone"`       // 10 | 100 | 1000
    TotalClicks   int64     `json:"total_clicks"`    // current count at time of trigger
}
func (e MilestoneReachedEvent) GetEventType() string    { return e.EventType }
func (e MilestoneReachedEvent) GetOccurredAt() time.Time { return e.OccurredAt }
```

Field rationale:

- `UserEmail` is embedded in every event because notification-service must send email without calling user-service. If this field is absent, notification-service would need a synchronous callback — the exact Chatty Service anti-pattern.
- `CorrelationID` appears on every event so async log traces can be followed through the message broker.
- `EventID` on `URLClickedEvent` is the deduplication key used by analytics-service (see M4).
- `*time.Time` (pointer) for `ExpiresAt` allows `null` in JSON without a sentinel value.

3.2 Health Response Schema

```go
// HealthResponse is the JSON body returned by GET /health on every service.
type HealthResponse struct {
    Status  string `json:"status"`   // always "ok" in this milestone
    Service string `json:"service"`  // service name: "url-service", "analytics-service", etc.
}
```

3.3 Logger Fields (Structured JSON)

Every log line must contain these fields. Fields absent from the current scope should be empty strings, not omitted.

```go
// LogEntry documents the expected JSON shape of every log line.
// Actual emission is via zerolog — this struct is documentation only.
type LogEntry struct {
    Level         string `json:"level"`          // "debug"|"info"|"warn"|"error"
    Time          string `json:"time"`            // RFC3339Nano
    Service       string `json:"service"`
    CorrelationID string `json:"correlation_id"`  // empty string if not yet available
    Method        string `json:"method"`          // HTTP method; empty for non-request logs
    Path          string `json:"path"`            // HTTP path; empty for non-request logs
    Status        int    `json:"status"`          // HTTP status; 0 for non-request logs
    DurationMS    int64  `json:"duration_ms"`     // 0 for non-request logs
    Msg           string `json:"msg"`
}
```

4. Interface Contracts

4.1 `shared/logger` Package

```go
package logger

import (
    "io"
    "github.com/rs/zerolog"
)

// New returns a zerolog.Logger configured for structured JSON output.
// serviceName is embedded in every log line as the "service" field.
// w is the output writer; pass os.Stdout in production.
// Call once at service startup; store in the service's dependency struct.
func New(serviceName string, w io.Writer) zerolog.Logger
```

Behavior:

- Sets `zerolog.TimeFieldFormat = zerolog.TimeFormatUnixMs` globally.
- Returns a logger with the `service` field pre-populated.
- The returned `zerolog.Logger` is value-safe to copy; pass it by value into handlers.
- No global logger variable is set. All callers receive the instance and pass it forward.

4.2 `handler.HealthHandler`

Each service has an identical health handler. Specify once; replicate across services.

```go
// NewHealthHandler returns an http.HandlerFunc that responds to GET /health.

// serviceName must be one of: "url-service", "analytics-service",

// "user-service", "notification-service", "gateway".

func NewHealthHandler(serviceName string, log zerolog.Logger) http.HandlerFunc
```

Contract:

- Method: `GET /health` (method not enforced in this milestone; any method returns 200).
- Response status: `200 OK`.
- Response `Content-Type`: `application/json`.
- Response body: `{"status":"ok","service":"<serviceName>"}`.
- Latency: must respond in < 10ms (no DB ping, no external call — pure in-memory).
- The handler must not block. It must not acquire any lock that a slow DB connection could hold. **Error cases:** None — this handler has no external dependencies and cannot fail.

4.3 DB Connection Bootstrap

```go
// ConnectDB opens a pgxpool connection pool and verifies connectivity.

// dsn is the full PostgreSQL DSN, e.g.:

//   "postgres://user:pass@host:5432/dbname?sslmode=disable"

// If the pool cannot ping the DB within 15 seconds (with exponential backoff

// starting at 500ms, doubling, cap 5s), the function returns a non-nil error.

// Callers must call os.Exit(1) on error — DB unavailability is fatal.

func ConnectDB(ctx context.Context, dsn string) (*pgxpool.Pool, error)
```

pgxpool configuration applied inside ConnectDB:

```go
config.MaxConns = 10

config.MinConns = 2

config.MaxConnLifetime = 30 * time.Minute

config.MaxConnIdleTime = 5 * time.Minute

config.HealthCheckPeriod = 1 * time.Minute
```

Retry logic (applied before returning an error):

- Attempt 1: immediate.
- Attempt 2: wait 500ms.
- Attempt 3: wait 1s.
- Attempt 4: wait 2s.
- Attempt 5: wait 4s (cap).
- After attempt 5 fails: return `fmt.Errorf("connectDB: failed after 5 attempts: %w", lastErr)`.
- Each attempt calls `pool.Ping(ctx)`. A successful ping returns the pool immediately. **Log on success:** `log.Info().Str("dsn_host", extractHost(dsn)).Msg("connected to DB")` *(Strip credentials from DSN before logging — see § 6 Error Handling Matrix.)*

4.4 Redis Connection Bootstrap (url-service only)

```go
// ConnectRedis initializes a redis.Client and attempts a PING.
// addr is "host:port", e.g. "redis:6379".
// Returns (client, nil) on success.
// Returns (nil, err) if PING fails — but the CALLER must treat this as
// a non-fatal warning and continue startup.
// The client is nil-safe in the cache layer: all cache helpers must check
// for a nil client and return a cache-miss sentinel instead of panicking.
func ConnectRedis(ctx context.Context, addr string) (*redis.Client, error)
```

Behavior:

- Uses `redis.NewClient` with `DialTimeout: 3s`, `ReadTimeout: 1s`, `WriteTimeout: 1s`.
- Single PING attempt — no retry loop (Redis is optional; don't block startup).
- On failure: logs `log.Warn().Err(err).Msg("redis unavailable; cache disabled")`.
- Returns `nil, err` to signal disabled cache. Callers store the `*redis.Client` as-is; nil means "cache is off."

4.5 RabbitMQ Bootstrap

```go
// ConnectRabbitMQ establishes an AMQP connection and channel, then declares
// the topic exchange and all queues + bindings.
// url is the AMQP connection URL, e.g. "amqp://guest:guest@rabbitmq:5672/"
// declareFunc is called once the channel is open; it receives the channel
// and declares whichever exchanges/queues/bindings are appropriate for the
// calling service (different services declare different queues).
// Returns (conn, channel, nil) on success.
// Retries with exponential backoff (same schedule as ConnectDB) for up to
// 5 attempts. Returns error if all attempts fail — callers log a warning
// and schedule a retry; RabbitMQ unavailability must NOT crash services.
func ConnectRabbitMQ(
    ctx context.Context,
    url string,
    declareFunc func(ch *amqp091.Channel) error,
) (*amqp091.Connection, *amqp091.Channel, error)
```

Exchange declaration (performed by url-service's declareFunc):

```go
err = ch.ExchangeDeclare(
    "url-shortener", // name
    "topic",         // kind
    true,            // durable
    false,           // auto-deleted
    false,           // internal
    false,           // no-wait
    nil,             // args
)
```

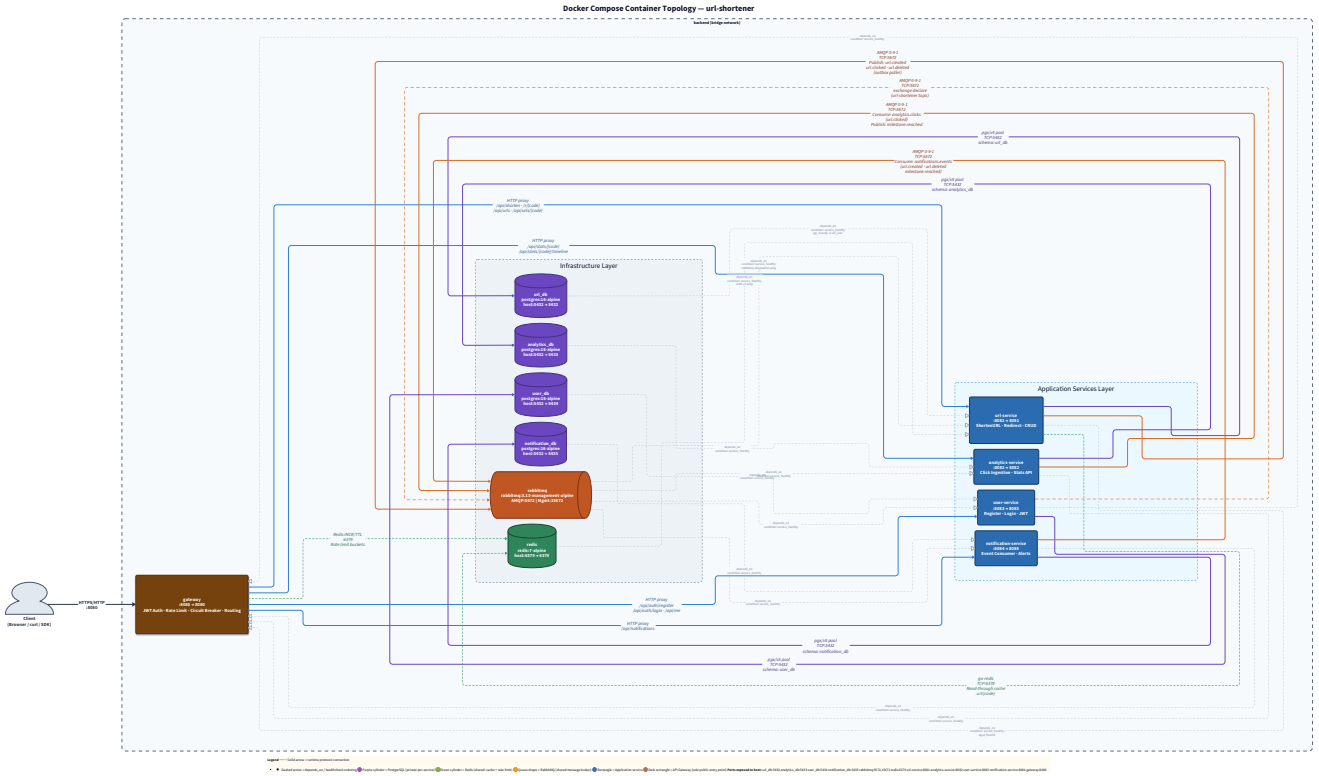
Queue + binding declaration (analytics-service's declareFunc):

```go
// Declare queue
q, err := ch.QueueDeclare(
    "analytics.clicks", // name
    true,               // durable
    false,              // delete when unused
    false,              // exclusive
    false,              // no-wait
    nil,
)
// Bind to exchange with routing key pattern
err = ch.QueueBind(
    q.Name,
    "url.clicked",   // routing key
    "url-shortener", // exchange
    false, nil,
)
```

Queue + binding declaration (notification-service's declareFunc):

```go
q, err := ch.QueueDeclare("notifications.events", true, false, false, false, nil)
// Bind three routing keys
for _, key := range []string{"url.created", "url.deleted", "milestone.reached"} {
    err = ch.QueueBind(q.Name, key, "url-shortener", false, nil)
}
```

Why consumers declare their own queues: If only the producer declares the queue, messages published before the consumer starts are dropped. Each consumer declaring its own queue-binding means the queue exists as soon as the consumer starts, regardless of producer order.



5. Algorithm Specification

5.1 Service Startup Sequence

Every service's `main.go` follows this exact procedure. Deviations (e.g., treating DB errors as warnings) are forbidden.

```
PROCEDURE StartService(serviceName, config):
  1. Parse configuration from environment variables.
     Abort (log.Fatal) if any required variable is missing.
  2. Initialize logger:
       log = logger.New(serviceName, os.Stdout)
  3. Connect to PostgreSQL:
       pool, err = ConnectDB(ctx, config.DatabaseDSN)
       IF err != nil:
         log.Fatal().Err(err).Msg("database connection failed")
         os.Exit(1)                    // non-zero exit; Docker will not mark healthy
       log.Info().Msg("connected to DB")
  4. [url-service only] Connect to Redis:
       redisClient, err = ConnectRedis(ctx, config.RedisAddr)
       IF err != nil:
         log.Warn().Err(err).Msg("redis unavailable; cache disabled")
         redisClient = nil             // service continues without cache
  5. Connect to RabbitMQ (all services):
       conn, ch, err = ConnectRabbitMQ(ctx, config.RabbitMQURL, declareFuncForService)
       IF err != nil:
         log.Warn().Err(err).Msg("rabbitmq unavailable; will retry in background")
         // Do NOT exit. Schedule a background goroutine to retry connection
         // every 10s. In this milestone, no messages are published/consumed,
         // so this is a non-fatal degradation.
  6. Register routes:
       mux = http.NewServeMux()
       mux.HandleFunc("GET /health", handler.NewHealthHandler(serviceName, log))
  7. Start HTTP server:
       server = &http.Server{
           Addr:         ":" + config.Port,
           Handler:      mux,
           ReadTimeout:  5 * time.Second,
           WriteTimeout: 10 * time.Second,
           IdleTimeout:  120 * time.Second,
       }
       log.Info().Str("port", config.Port).Msg("service starting")
       if err = server.ListenAndServe(); err != nil {
           log.Fatal().Err(err).Msg("server exited")
       }
END PROCEDURE
```

Step 5 background retry goroutine (skeleton only — actual publishing is M3+):

```go
go func() {

    for {

        time.Sleep(10 * time.Second)

        conn, ch, err := ConnectRabbitMQ(ctx, config.RabbitMQURL, declareFunc)

        if err != nil {

            log.Warn().Err(err).Msg("rabbitmq reconnect failed; will retry")

            continue

        }

        // Store conn and ch in service-level state.

        // In M1, just log success.

        log.Info().Msg("rabbitmq reconnected")

        return

    }

}()
```

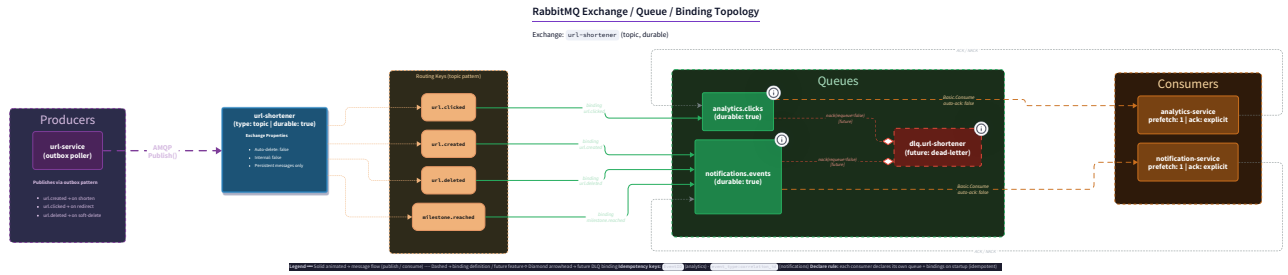
5.2 Gateway Startup Health-Check Loop

The gateway's `main.go` performs health-checks against all four downstream services during startup. It does NOT block startup on failure — it logs the degraded state and proceeds.

```
PROCEDURE GatewayStartupHealthCheck(downstreamURLs []string, log):
  FOR each serviceURL in downstreamURLs:
    resp, err = http.Get(serviceURL + "/health")
    IF err != nil OR resp.StatusCode != 200:
      log.Warn().Str("service", serviceURL).Msg("downstream health check failed on startup")
    ELSE:
      log.Info().Str("service", serviceURL).Msg("downstream healthy")
    close resp.Body
  // Gateway starts regardless of downstream health.
  // Circuit breaker (M5) will handle runtime failures.
END PROCEDURE
```

Downstream URLs are read from environment variables:

- `URL_SERVICE_URL` → `http://url-service:8081`
- `ANALYTICS_SERVICE_URL` → `http://analytics-service:8082`
- `USER_SERVICE_URL` → `http://user-service:8083`
- `NOTIFICATION_SERVICE_URL` → `http://notification-service:8084`



6. Docker Compose Specification

6.1 Container Inventory

Container Name	Image	Internal Port	Host Port	Role
`url-service`	built from `services/url-service/Dockerfile`	8081	8081	URL CRUD + redirects
`analytics-service`	built from `services/analytics-service/Dockerfile`	8082	8082	Click ingestion + stats
`user-service`	built from `services/user-service/Dockerfile`	8083	8083	Auth
`notification-service`	built from `services/notification-service/Dockerfile`	8084	8084	Alerts
`gateway`	built from `gateway/Dockerfile`	8080	8080	Single entry point
`url_db`	`postgres:16-alpine`	5432	5432	url-service DB
`analytics_db`	`postgres:16-alpine`	5432	5433	analytics-service DB
`user_db`	`postgres:16-alpine`	5432	5434	user-service DB
`notification_db`	`postgres:16-alpine`	5432	5435	notification-service DB
`rabbitmq`	`rabbitmq:3.13-management-alpine`	5672 / 15672	5672 / 15672	Message broker
`redis`	`redis:7-alpine`	6379	6379	Shared cache + rate limit store
Total: 11 containers.				

6.2 docker-compose.yml (complete, annotated)

```yaml
# docker-compose.yml

# Run: docker compose up --build

# Tear down: docker compose down -v

name: url-shortener

x-postgres-base: &postgres-base

  image: postgres:16-alpine

  restart: unless-stopped

  environment:

    POSTGRES_PASSWORD: secret

x-app-base: &app-base

  restart: unless-stopped

  networks:

    - backend

networks:

  backend:

    driver: bridge

volumes:

  url_db_data:

  analytics_db_data:

  user_db_data:

  notification_db_data:

  rabbitmq_data:

  redis_data:

services:

  # ── Databases ──────────────────────────────────────

  url_db:

    <<: *postgres-base

    container_name: url_db

    environment:

      POSTGRES_DB: url_db

      POSTGRES_USER: url_user

      POSTGRES_PASSWORD: url_secret

    ports:

      - "5432:5432"

    volumes:

      - url_db_data:/var/lib/postgresql/data

    networks:

      - backend

    healthcheck:
```

```
    test: ["CMD-SHELL", "pg_isready -U url_user -d url_db"]
    interval: 5s
    timeout: 5s
    retries: 10
    start_period: 10s
analytics_db:
  <<: *postgres-base
  container_name: analytics_db
  environment:
    POSTGRES_DB: analytics_db
    POSTGRES_USER: analytics_user
    POSTGRES_PASSWORD: analytics_secret
  ports:
    - "5433:5432"
  volumes:
    - analytics_db_data:/var/lib/postgresql/data
  networks:
    - backend
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U analytics_user -d analytics_db"]
    interval: 5s
    timeout: 5s
    retries: 10
    start_period: 10s
user_db:
  <<: *postgres-base
  container_name: user_db
  environment:
    POSTGRES_DB: user_db
    POSTGRES_USER: user_user
    POSTGRES_PASSWORD: user_secret
  ports:
    - "5434:5432"
  volumes:
    - user_db_data:/var/lib/postgresql/data
  networks:
    - backend
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U user_user -d user_db"]
    interval: 5s
```

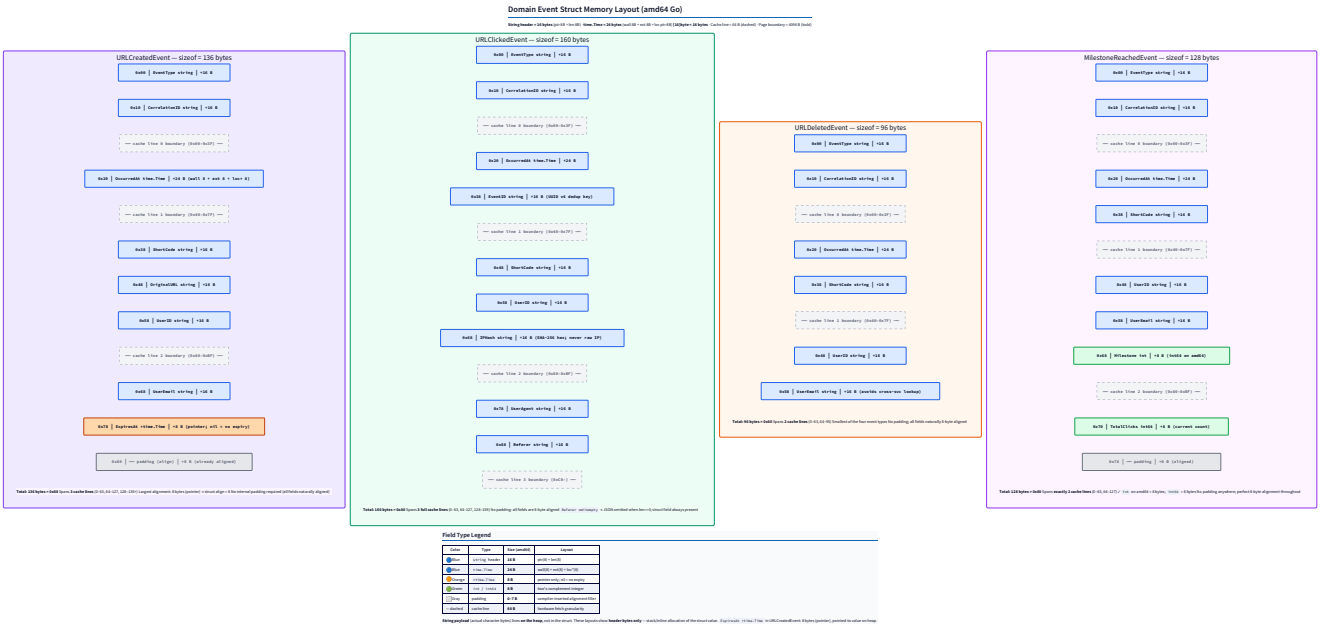
```
    timeout: 5s
    retries: 10
    start_period: 10s

notification_db:
  <<: *postgres-base
  container_name: notification_db
  environment:
    POSTGRES_DB: notification_db
    POSTGRES_USER: notification_user
    POSTGRES_PASSWORD: notification_secret
  ports:
    - "5435:5432"
  volumes:
    - notification_db_data:/var/lib/postgresql/data
  networks:
    - backend
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U notification_user -d notification_db"]
    interval: 5s
    timeout: 5s
    retries: 10
    start_period: 10s

# ── Message Broker ──────────────────────────────────────

rabbitmq:
  image: rabbitmq:3.13-management-alpine
  container_name: rabbitmq
  ports:
    - "5672:5672"     # AMQP
    - "15672:15672"   # Management UI (admin/admin)
  volumes:
    - rabbitmq_data:/var/lib/rabbitmq
  environment:
    RABBITMQ_DEFAULT_USER: admin
    RABBITMQ_DEFAULT_PASS: admin
  networks:
    - backend
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "ping"]
    interval: 10s
    timeout: 10s
```

```yaml
    retries: 10
    start_period: 30s

# ── Cache ──────────────────────────────────────────────
redis:
  image: redis:7-alpine
  container_name: redis
  ports:
    - "6379:6379"
  volumes:
    - redis_data:/data
  networks:
    - backend
  command: redis-server --appendonly yes
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 5s
    timeout: 3s
    retries: 10
    start_period: 5s

# ── Application Services ──────────────────────────────
url-service:
  <<: *app-base
  container_name: url-service
  build:
    context: ./services/url-service
    dockerfile: Dockerfile
  ports:
    - "8081:8081"
  environment:
    PORT: "8081"
    DATABASE_DSN: "postgres://url_user:url_secret@url_db:5432/url_db?sslmode=disable"
    REDIS_ADDR: "redis:6379"
    RABBITMQ_URL: "amqp://admin:admin@rabbitmq:5672/"
    SERVICE_NAME: "url-service"
  depends_on:
    url_db:
      condition: service_healthy
    redis:
      condition: service_healthy
    rabbitmq:
```

```
      condition: service_healthy
  healthcheck:
    test: ["CMD-SHELL", "wget -qO- http://localhost:8081/health || exit 1"]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 15s
analytics-service:
  <<: *app-base
  container_name: analytics-service
  build:
    context: ./services/analytics-service
    dockerfile: Dockerfile
  ports:
    - "8082:8082"
  environment:
    PORT: "8082"
    DATABASE_DSN: "postgres://analytics_user:analytics_secret@analytics_db:5432/analytics_db?sslmode=disable"
    RABBITMQ_URL: "amqp://admin:admin@rabbitmq:5672/"
    SERVICE_NAME: "analytics-service"
  depends_on:
    analytics_db:
      condition: service_healthy
    rabbitmq:
      condition: service_healthy
  healthcheck:
    test: ["CMD-SHELL", "wget -qO- http://localhost:8082/health || exit 1"]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 15s
user-service:
  <<: *app-base
  container_name: user-service
  build:
    context: ./services/user-service
    dockerfile: Dockerfile
  ports:
    - "8083:8083"
  environment:
```

```
    PORT: "8083"
    DATABASE_DSN: "postgres://user_user:user_secret@user_db:5432/user_db?sslmode=disable"
    RABBITMQ_URL: "amqp://admin:admin@rabbitmq:5672/"
    SERVICE_NAME: "user-service"
  depends_on:
    user_db:
      condition: service_healthy
    rabbitmq:
      condition: service_healthy
  healthcheck:
    test: ["CMD-SHELL", "wget -qO- http://localhost:8083/health || exit 1"]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 15s
notification-service:
  <<: *app-base
  container_name: notification-service
  build:
    context: ./services/notification-service
    dockerfile: Dockerfile
  ports:
    - "8084:8084"
  environment:
    PORT: "8084"
    DATABASE_DSN: "postgres://notification_user:notification_secret@notification_db:5432/notification_db?sslmode=disable"
    RABBITMQ_URL: "amqp://admin:admin@rabbitmq:5672/"
    SERVICE_NAME: "notification-service"
  depends_on:
    notification_db:
      condition: service_healthy
    rabbitmq:
      condition: service_healthy
  healthcheck:
    test: ["CMD-SHELL", "wget -qO- http://localhost:8084/health || exit 1"]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 15s
gateway:
```

```yaml
  <<: *app-base

  container_name: gateway

  build:

    context: ./gateway

    dockerfile: Dockerfile

  ports:

    - "8080:8080"

  environment:

    PORT: "8080"

    SERVICE_NAME: "gateway"

    URL_SERVICE_URL: "http://url-service:8081"

    ANALYTICS_SERVICE_URL: "http://analytics-service:8082"

    USER_SERVICE_URL: "http://user-service:8083"

    NOTIFICATION_SERVICE_URL: "http://notification-service:8084"

    REDIS_ADDR: "redis:6379"

  depends_on:

    url-service:

      condition: service_healthy

    analytics-service:

      condition: service_healthy

    user-service:

      condition: service_healthy

    notification-service:

      condition: service_healthy

  healthcheck:

    test: ["CMD-SHELL", "wget -qO- http://localhost:8080/health || exit 1"]

    interval: 10s

    timeout: 5s

    retries: 5

    start_period: 20s
```



7. Go Module Declarations

Each service has its own `go.mod`. The module path prefix is `github.com/yourhandle/url-shortener`.

```
# services/url-service/go.mod
module github.com/yourhandle/url-shortener/url-service
go 1.23
require (
    github.com/jackc/pgx/v5 v5.6.0
    github.com/redis/go-redis/v9 v9.5.1
    github.com/rabbitmq/amqp091-go v1.10.0
    github.com/rs/zerolog v1.33.0
    github.com/yourhandle/url-shortener/shared v0.0.0
)
replace github.com/yourhandle/url-shortener/shared => ../../shared
```

Identical `require` blocks (adjusting the replace path) for `analytics-service`, `user-service`, `notification-service`, and `gateway`. The `shared` module:

```
# shared/go.mod
module github.com/yourhandle/url-shortener/shared
go 1.23
require (
    github.com/rs/zerolog v1.33.0
)
```

Important: Use `replace` directives in each service's `go.mod` to reference the local `shared` module. This avoids publishing to a registry during development. After adding the replace directive, run `go mod tidy` in each service directory.

8. Dockerfile Specification

All five Dockerfiles are identical in structure. One canonical example:

```dockerfile
# services/url-service/Dockerfile

# Stage 1: Build

FROM golang:1.23-alpine AS builder

# Install wget for healthcheck probe (not in the final image — it uses wget from Alpine)

RUN apk add --no-cache git

WORKDIR /app

# Copy go.mod and go.sum first for layer caching

COPY go.mod go.sum ./

# Copy shared module source (referenced via replace directive)

COPY ../../shared /shared

RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux go build -trimpath -o /bin/service ./...

# Stage 2: Runtime

FROM alpine:3.20

RUN apk add --no-cache wget ca-certificates tzdata

WORKDIR /app

COPY --from=builder /bin/service /app/service

EXPOSE 8081

ENTRYPOINT ["/app/service"]
```

Build context note: Because each service's Dockerfile needs access to the `shared` directory (one level up from the service root), the `docker-compose.yml` sets `context` to the service directory. Use a `.dockerignore` that excludes nothing needed, or structure the Dockerfile COPY to handle the parent reference. An alternative that avoids context issues: Set `context: .` (repo root) and `dockerfile: services/url-service/Dockerfile` in `docker-compose.yml`. Then COPY paths are relative to the repo root:

```dockerfile
# Alternate approach with context=. (repo root) in docker-compose

COPY services/url-service/go.mod services/url-service/go.sum ./

COPY shared/ /shared/

COPY services/url-service/ .
```

Choose one approach and apply consistently. The repo-root context approach is simpler for multi-module monorepos and is the recommended path.

9. Environment Variable Reference

Variable	Used By	Required	Default	Description
`PORT`	all	yes	—	HTTP listen port
`SERVICE_NAME`	all	yes	—	Embedded in logs
`DATABASE_DSN`	all app services	yes	—	Full PostgreSQL DSN
`REDIS_ADDR`	url-service, gateway	yes	—	`host:port`
`RABBITMQ_URL`	all app services	yes	—	AMQP URL
`URL_SERVICE_URL`	gateway	yes	—	Downstream base URL
`ANALYTICS_SERVICE_URL`	gateway	yes	—	Downstream base URL
`USER_SERVICE_URL`	gateway	yes	—	Downstream base URL
`NOTIFICATION_SERVICE_URL`	gateway	yes	—	Downstream base URL
`JWT_SECRET`	user-service, all (M2+)	no in M1	—	Added in M2
Parsing pattern (applied in each service's `main.go`):				

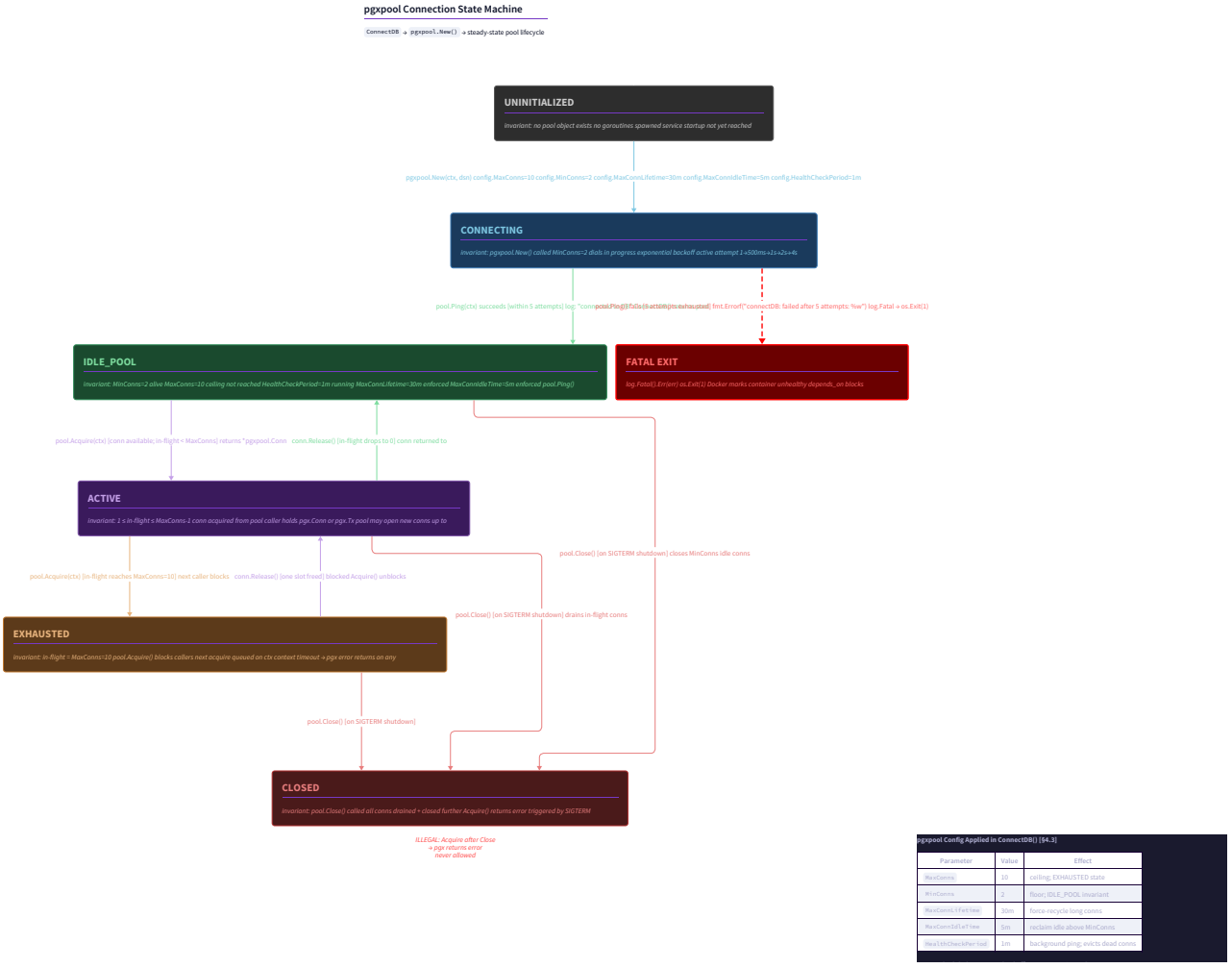
```go
func mustGetEnv(key string) string {
    v := os.Getenv(key)
    if v == "" {
        fmt.Fprintf(os.Stderr, "FATAL: required environment variable %q is not set\n", key)
        os.Exit(1)
    }
    return v
}
```

Call `mustGetEnv` for every required variable before any I/O. This guarantees a clean failure message rather than a nil-pointer panic deep in a library.

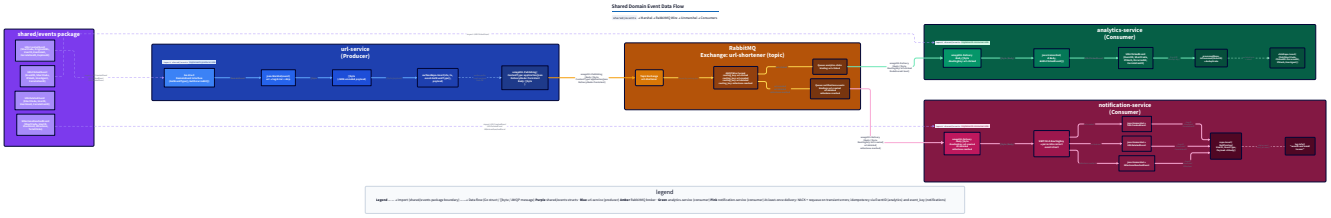
10. Error Handling Matrix

Error	Detection Point	Severity	Recovery	User-Visible Effect
`DB_UNREACHABLE`	`ConnectDB` after 5 retry attempts	**Fatal**	`log.Fatal` + `os.Exit(1)`	Container exits; Docker healthcheck fails; depends_on prevents downstream startup
`REDIS_UNREACHABLE`	`ConnectRedis` single PING attempt	Warning	Log warning, set client to nil, continue	url-service starts without cache; all cache calls no-op
`RABBITMQ_UNREACHABLE`	`ConnectRabbitMQ` after 5 retry attempts	Warning	Log warning, start background retry goroutine	Service starts; exchange/queue not yet declared; no messages published/consumed until reconnect
`REQUIRED_ENV_MISSING`	`mustGetEnv`	Fatal	`os.Exit(1)`	Container exits immediately with clear message
`DSN_CREDENTIALS_IN_LOG`	`ConnectDB` logging	N/A	Strip credentials from DSN before logging	Never log passwords
`HEALTHCHECK_DOWNSTREAM_FAIL`	Gateway startup loop	Warning	Log degraded state, continue	Gateway starts; circuit breaker (M5) handles runtime failures
`PORT_BIND_FAIL`	`server.ListenAndServe`	Fatal	`log.Fatal`	Container exits; Docker restart policy applies
DSN credential stripping:				

```go
// extractHost parses a DSN URL and returns only scheme+host+dbname for logging.
// Input:  "postgres://url_user:url_secret@url_db:5432/url_db?sslmode=disable"
// Output: "postgres://url_db:5432/url_db"
func extractHost(dsn string) string {
    u, err := url.Parse(dsn)
    if err != nil {
        return "<unparseable DSN>"
    }
    u.User = nil // strip username and password
    u.RawQuery = "" // strip query params (may contain passwords)
    return u.String()
}
```



11. RabbitMQ Topology Specification



Exchange

Property	Value
Name	`url-shortener`
Type	`topic`
Durable	`true`
Auto-delete	`false`
Internal	`false`

Queues

Queue	Durable	Exclusive	Auto-delete	Declared by
`analytics.clicks`	true	false	false	analytics-service
`notifications.events`	true	false	false	notification-service

Bindings

Queue	Exchange	Routing Key	Purpose
`analytics.clicks`	`url-shortener`	`url.clicked`	Click events → analytics
`notifications.events`	`url-shortener`	`url.created`	New URL notification
`notifications.events`	`url-shortener`	`url.deleted`	Deleted URL notification
`notifications.events`	`url-shortener`	`milestone.reached`	Milestone notification
Idempotent declaration: AMQP `ExchangeDeclare` and `QueueDeclare` are idempotent if called with the same parameters. If a service restarts and redeclares, the broker accepts it silently. If parameters differ, the broker returns a channel-level error — ensure parameters are identical across all restart scenarios.			
Startup order guarantee: `url-service` declares the exchange. `analytics-service` and `notification-service` declare their queues and bindings. Both consumers must succeed in `QueueBind` before the exchange exists — this is handled by the RabbitMQ `depends_on: rabbitmq: condition: service_healthy` in docker-compose. The exchange is declared when url-service starts (which also waits on rabbitmq healthy). However, to be fully safe, each consumer's `declareFunc` should also call `ExchangeDeclare` — AMQP declare is idempotent and this removes the startup ordering dependency between app services.			

12. Implementation Sequence with Checkpoints

Phase 1 — Monorepo Scaffold + go.mod (0.5–1 hr)

1. Create the directory tree from § 2.
2. Write `go.mod` for `shared/` first.
3. Write `go.mod` for each service with the `replace` directive.
4. Create empty `main.go` stubs (`package main\nfunc main() {}`) in each service.
5. Run `go mod tidy` in `shared/`, then in each service directory. **Checkpoint:** `go build ./...` succeeds in every service directory with no errors. No business logic yet — just valid Go modules that compile.

Phase 2 — shared/events/ + shared/logger/ (0.5–1 hr)

1. Write `shared/events/events.go` (§ 3.1 — all four event structs + constants).
2. Write `shared/logger/logger.go` (§ 4.1).
3. Run `go test ./...` in `shared/` — no test files yet, but must compile clean.
4. In each service, add `import _ "github.com/yourhandle/url-shortener/shared/events"` temporarily to verify the replace directive works. **Checkpoint:** `go build ./...` in every service compiles without error. `shared/events` types are importable.

Phase 3 — docker-compose.yml (1–1.5 hr)

1. Write `docker-compose.yml` from § 6.2.
2. Write `.env.example` documenting all variables from § 9.
3. Write `Dockerfile` for each service (§ 8) — they are all identical except port numbers.
4. Run `docker compose config` — must produce no YAML errors.
5. Run `docker compose up rabbitmq redis url_db analytics_db user_db notification_db` (infra only). **Checkpoint:** All six infrastructure containers reach `healthy` state within 60 seconds. Verify with `docker compose ps` — every status column reads `healthy`. RabbitMQ management UI accessible at `http://localhost:15672` (admin/admin).

Phase 4 — DB + Redis + RabbitMQ Bootstrapping (1–1.5 hr)

1. Write `ConnectDB`, `ConnectRedis`, `ConnectRabbitMQ` in each service's `main.go` (or a shared `infra/` sub-package within each service).
2. Write `mustGetEnv` and parse all required environment variables.
3. Implement startup procedure from § 5.1 in each `main.go`.
4. Implement service-specific `declareFunc` for each service (§ 4.5).
5. Run `docker compose up --build` (all services). **Checkpoint:** `docker compose logs url-service | grep "connected to DB"` prints the success message. `docker compose logs url-service | grep "connected to Redis cache"` prints success. RabbitMQ management UI at `http://localhost:15672` → Exchanges tab shows `url-shortener` (topic, durable). Queues tab shows `analytics.clicks` and `notifications.events` with correct bindings.

Phase 5 — /health Handlers + Gateway Startup Check (0.5–1 hr)

1. Write `handler.NewHealthHandler` in each service's `handler/health.go`.
2. Register the handler in each service's `main.go`.
3. Write gateway's startup health-check loop (§ 5.2).
4. Write README with setup commands. **Checkpoint:**

```bash
curl -s http://localhost:8081/health | python3 -m json.tool

# Expected: {"status": "ok", "service": "url-service"}

curl -s http://localhost:8082/health

# Expected: {"status":"ok","service":"analytics-service"}

curl -s http://localhost:8083/health

# Expected: {"status":"ok","service":"user-service"}

curl -s http://localhost:8084/health

# Expected: {"status":"ok","service":"notification-service"}

curl -s http://localhost:8080/health

# Expected: {"status":"ok","service":"gateway"}

docker compose ps

# All 11 containers: Status = "healthy"
```

13. Test Specification

13.1 shared/events Package Tests

File: `shared/events/events_test.go`

```go
// Test: JSON round-trip for each event struct

// Happy path: marshal URLClickedEvent → unmarshal → fields match

// Edge case: ExpiresAt = nil in URLCreatedEvent → JSON contains no "expires_at" key

// Edge case: OccurredAt preserves nanosecond precision through JSON round-trip

// Verify: GetEventType() returns the correct constant for each struct

// Verify: EventType constants match expected routing key strings exactly

func TestEventJSONRoundTrip(t *testing.T)

func TestNilExpiresAtOmittedFromJSON(t *testing.T)

func TestGetEventTypeReturnsCorrectConstant(t *testing.T)
```

13.2 ConnectDB Tests (integration, requires Docker)

```go
// Happy path: valid DSN → pool returned, pool.Ping() succeeds

// Failure: invalid host → all 5 retries exhausted → error returned

// Verify: pool has MaxConns=10, MinConns=2 (inspect config after Connect)

// Verify: error message contains "failed after 5 attempts"

func TestConnectDB_HappyPath(t *testing.T)          // requires url_db running

func TestConnectDB_UnreachableHost(t *testing.T)     // use port 9999 (guaranteed unreachable)

func TestConnectDB_PoolConfig(t *testing.T)
```

13.3 ConnectRedis Tests

```go
// Happy path: valid addr → client returned, err == nil

// Failure: unreachable addr → err != nil, returned client == nil

// Verify: function does NOT panic on PING failure

func TestConnectRedis_HappyPath(t *testing.T)        // requires redis running

func TestConnectRedis_Unreachable_ReturnsNilClient(t *testing.T)
```

13.4 Health Handler Tests

```go
// Happy path: GET /health → 200, body = {"status":"ok","service":"url-service"}

// Content-Type header = "application/json"

// Response time < 10ms (use testing.B for benchmark)

// Any HTTP method → 200 (handler is method-agnostic in M1)

func TestHealthHandler_Returns200(t *testing.T)

func TestHealthHandler_JSONBody(t *testing.T)

func TestHealthHandler_ContentType(t *testing.T)

func BenchmarkHealthHandler(b *testing.B)
```

13.5 extractHost Tests

```go
// Input: "postgres://user:pass@url_db:5432/url_db?sslmode=disable"

// Expected: "postgres://url_db:5432/url_db"

// Input: unparseable string → "<unparseable DSN>"

// Verify: no user info in output

// Verify: no query params in output

func TestExtractHost_StripsCreds(t *testing.T)

func TestExtractHost_UnparseableDSN(t *testing.T)
```

13.6 End-to-End Smoke Test (shell script)

```bash
#!/bin/bash

# test/smoke_m1.sh — run after `docker compose up --build`

set -euo pipefail

BASE_URLS=("http://localhost:8081" "http://localhost:8082" "http://localhost:8083" "http://localhost:8084" "http://localhost:8080")

NAMES=("url-service" "analytics-service" "user-service" "notification-service" "gateway")

for i in "${!BASE_URLS[@]}"; do

    RESPONSE=$(curl -sf "${BASE_URLS[$i]}/health")

    EXPECTED="{\"status\":\"ok\",\"service\":\"${NAMES[$i]}\"}"

    if [ "$RESPONSE" != "$EXPECTED" ]; then

        echo "FAIL: ${NAMES[$i]} returned: $RESPONSE"

        exit 1

    fi

    echo "PASS: ${NAMES[$i]}"

done

echo "All health checks passed."
```

14. Performance Targets

Operation	Target	Measurement Method
`GET /health` response time	< 10ms p99	`wrk -t2 -c10 -d10s http://localhost:8081/health` → `Latency 99%`
`docker compose up` to all-healthy	< 60s	`time docker compose up --wait` on commodity hardware (4 core, 8GB RAM)
`ConnectDB` retry exhaustion time	< 25s (5 attempts × up to 5s each)	integration test with fake DSN + `testing.Timer`
`go build ./...` in each service	< 30s	cold build on CI; warm build < 5s
Container image size (each service)	< 20MB	`docker image ls` after build (Alpine multi-stage)

15. Anti-Pattern Guard Rail Checklist

Before completing this milestone, verify none of the following are present:

- ☐ **No shared PostgreSQL container.** Run `docker compose config | grep POSTGRES_DB | sort | uniq -c` — expect 4 unique DB names across 4 separate containers.
- ☐ **No `localhost` in any connection string inside Docker.** All hostnames in environment variables use Docker service names (`url_db` , `redis` , `rabbitmq` , etc.).
- ☐ **Redis is not used as a primary store.** No `SET` or `HSET` to Redis in `main.go` — only the connection bootstrap PING.
- ☐ **RabbitMQ queue declarations are in consumers, not only in url-service.** `analytics-service/main.go` declares `analytics.clicks` ; `notification-service/main.go` declares `notifications.events` .
- ☐ **`depends_on` uses `condition: service_healthy` for all DB and broker dependencies.** Plain `depends_on: [url_db]` without a condition does not wait for readiness.
- ☐ **No hardcoded passwords in Go source.** All credentials come from `mustGetEnv` .

Technical Design Specification

Module: User Service — Registration, Login, JWT Issuance

Module ID: `url-shortener-m2`

1. Module Charter

The User Service is the single authority for identity within the URL shortener system. It accepts user registrations, validates credentials, stores password hashes, and issues signed JWT access tokens. It exposes exactly three HTTP endpoints: `POST /register`, `POST /login`, and `GET /me`. It owns one PostgreSQL table (`users`) in its private database and communicates with no other service — ever. This module does **not** implement password reset, OAuth, refresh tokens, email verification, session storage, role-based access control, or any form of inter-service communication. It does not call the URL Service, Analytics Service, or Notification Service. The gateway and all other services verify tokens locally using the shared `JWT_SECRET` — they never proxy a verification request to the User Service. **Upstream dependencies:** M1 foundation (Docker Compose stack, `shared/logger` package, PostgreSQL `user_db` container). The `user_db` database must be reachable before this service starts. **Downstream dependencies:** The `shared/middleware` JWT verification package produced in Phase 4 of this module is imported by url-service (M3), analytics-service (M4), notification-service (M5), and the gateway (M5). The `JWT_SECRET` environment variable must be identical across all services that verify tokens. **Invariants that must always hold after this milestone:**

- No plaintext password ever persists to disk or appears in any log line.
- A failed login returns exactly the same response body and HTTP status regardless of whether the email exists or the password is wrong.
- Token verification never makes a network call — it is pure HMAC-SHA256 computation against the in-process secret.
- The `users.email` column has a `UNIQUE NOT NULL` constraint enforced at the database level, not only at the application level.
- `GET /me` response time is bounded by HMAC computation only, not by database latency.

2. File Structure

Create files in the numbered order below. All paths are relative to the monorepo root.

```
url-shortener/
│
├── services/user-service/
│   ├── 01  migrations/
│   │   └── 02  001_create_users.sql
│   ├── 03  repository/
│   │   ├── 04  user_repository.go       # UserRepository interface + pgx implementation
│   │   └── 05  user_repository_test.go
│   ├── 06  auth/
│   │   ├── 07  password.go              # PasswordHasher: bcrypt Hash + Compare
│   │   ├── 08  password_test.go
│   │   ├── 09  jwt.go                   # JWTSigner + JWTVerifier implementations
│   │   └── 10  jwt_test.go
│   ├── 11  handler/
│   │   ├── 12  register.go              # POST /register
│   │   ├── 13  login.go                 # POST /login
│   │   ├── 14  me.go                    # GET /me
│   │   ├── 15  health.go               # GET /health (from M1, updated)
│   │   └── 16  handler_test.go          # integration tests (register→login→/me)
│   ├── 17  service/
│   │   └── 18  user_service.go          # UserService: orchestrates repo + auth
│   └── 19  main.go                      # wiring: env, DB, routes, server
│
└── shared/
    └── 20  middleware/
        ├── 21  jwt_middleware.go         # http.Handler wrapper: Bearer extraction + verify
        └── 22  jwt_middleware_test.go
```

Total new files: 22 (`go.sum` files are auto-generated and excluded from the count).

3. Complete Data Model

3.1 PostgreSQL Schema (`migrations/001_create_users.sql`)

```sql
-- migrations/001_create_users.sql

BEGIN;

CREATE TABLE IF NOT EXISTS users (

    id           UUID        PRIMARY KEY DEFAULT gen_random_uuid(),

    email        TEXT        NOT NULL UNIQUE,

    password_hash TEXT        NOT NULL,

    created_at   TIMESTAMPTZ NOT NULL DEFAULT now()

);

-- Index for login: find user by email quickly.

-- UNIQUE already implies a B-tree index; this is explicit documentation.

-- The UNIQUE constraint on email is enforced by the database, not application code.

CREATE UNIQUE INDEX IF NOT EXISTS idx_users_email ON users(email);

COMMIT;
```

Field rationale:

- `id UUID` : UUIDs are non-enumerable. An integer PK would leak user count via `user_id=42` . `gen_random_uuid()` requires `pgcrypto` on PostgreSQL < 14; on PostgreSQL 16 (as declared in docker-compose) it is a built-in function.
- `email TEXT NOT NULL UNIQUE` : Application-level uniqueness checks are racey; the constraint is the true guard. `TEXT` is used over `VARCHAR(255)` because PostgreSQL stores both identically; `TEXT` removes an arbitrary limit.
- `password_hash TEXT NOT NULL` : bcrypt output is always 60 characters of ASCII. `TEXT` is correct; never use `BYTEA` for bcrypt strings.
- `created_at TIMESTAMPTZ NOT NULL DEFAULT now()` : `TIMESTAMPTZ` stores UTC internally and is unambiguous across timezone changes. Application never writes this field.

3.2 Go Structs

```go
// repository/user_repository.go

package repository

import (

    "context"

    "time"

)

// User is the domain entity stored in the users table.

// It is the single source of truth for user identity within this service.

// Password hashes are stored here; plaintext passwords never reach this struct.

type User struct {

    ID           string    // UUID v4, primary key

    Email        string    // unique, lowercase-normalized before storage

    PasswordHash string    // bcrypt output, cost=12; never logged

    CreatedAt    time.Time // set by DB DEFAULT; application does not write this

}

// UserRepository defines the persistence contract for the User entity.

// Implementations must be safe for concurrent use.

type UserRepository interface {

    // Create inserts a new user. Returns ErrDuplicateEmail if the email

    // already exists. All other DB errors are returned as-is for the

    // service layer to wrap.

    Create(ctx context.Context, u User) error

    // FindByEmail retrieves a user by email address.

    // Returns ErrUserNotFound if no row matches.

    FindByEmail(ctx context.Context, email string) (User, error)

}

// Sentinel errors returned by UserRepository.

// Service layer maps these to HTTP status codes.

var (

    ErrDuplicateEmail = errors.New("email already registered")

    ErrUserNotFound   = errors.New("user not found")

)
```

```go
// auth/jwt.go

package auth

import "time"

// Claims is the set of fields embedded in a JWT payload.

// Only these fields are read on verification; any extra fields are ignored.

type Claims struct {

    Sub   string    // user_id UUID string — "subject" per RFC 7519

    Email string    // user email; embedded to avoid /me needing a DB call

    Iat   time.Time // issued at

    Exp   time.Time // expiry; 24h after Iat

    Iss   string    // always "url-shortener"

}

// jwtPayload is the raw JSON-mapped representation used for marshaling.

// Claims is the clean domain struct; jwtPayload handles the Unix timestamp encoding.

type jwtPayload struct {

    Sub   string `json:"sub"`

    Email string `json:"email"`

    Iat   int64  `json:"iat"` // Unix seconds

    Exp   int64  `json:"exp"` // Unix seconds

    Iss   string `json:"iss"`

}
```

```go
// handler/register.go — request/response DTOs

package handler

// RegisterRequest is the JSON body accepted by POST /register.

type RegisterRequest struct {

    Email    string `json:"email"`

    Password string `json:"password"`

}

// RegisterResponse is the JSON body returned on 201 Created.

// Password hash is never included. User ID is always a UUID string.

type RegisterResponse struct {

    UserID string `json:"user_id"`

    Email  string `json:"email"`

}

// LoginRequest is the JSON body accepted by POST /login.

type LoginRequest struct {

    Email    string `json:"email"`

    Password string `json:"password"`

}

// LoginResponse is the JSON body returned on 200 OK after successful login.

type LoginResponse struct {

    Token     string `json:"token"`

    ExpiresAt string `json:"expires_at"` // RFC3339 format

}

// MeResponse is the JSON body returned by GET /me.

type MeResponse struct {

    UserID string `json:"user_id"`

    Email  string `json:"email"`

}

// ErrorResponse is the JSON body for all error responses.

type ErrorResponse struct {

    Error string `json:"error"`

}
```

3.3 JWT Payload Wire Format

The JWT is a standard three-part HS256 token: `base64url(header).base64url(payload).base64url(signature)`. **Header (fixed):**

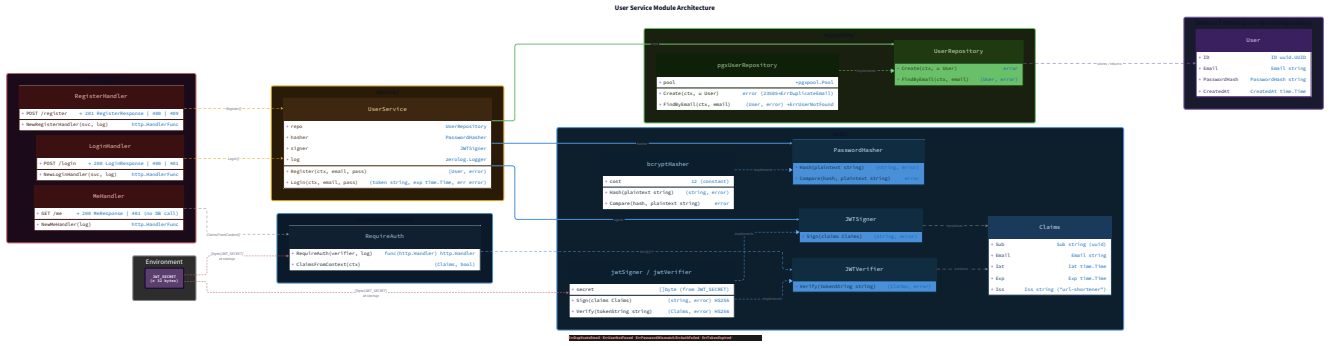
```json
{"alg":"HS256","typ":"JWT"}
```

Payload (per token):

```json
{
  "sub": "550e8400-e29b-41d4-a716-446655440000",
  "email": "alice@example.com",
  "iat": 1740873600,
  "exp": 1740960000,
  "iss": "url-shortener"
}
```

Field constraints:

Field	Type	Constraint
`sub`	string	UUID v4 format; must match `users.id`
`email`	string	Must match `users.email` at time of issuance
`iat`	number	Unix seconds; UTC
`exp`	number	`iat + 86400` (24 hours exactly)
`iss`	string	Literal `"url-shortener"` — reject tokens with any other issuer
Implementation note: Do NOT use a JWT library that pulls in JOSE complexity (e.g., `golang-jwt/jwt` is acceptable; `lestrrat-go/jwx` is overkill). The recommended library is `golang-jwt/jwt/v5` which handles HMAC signing, payload marshaling, and expiry checking natively.		



4. Interface Contracts

4.1 `UserRepository` — `pgxUserRepository` Implementation

```go
// NewPgxUserRepository constructs a UserRepository backed by a pgxpool.Pool.

// pool must be non-nil and already connected.

func NewPgxUserRepository(pool *pgxpool.Pool) UserRepository
```

`Create(ctx context.Context, u User) error`

- Executes: `INSERT INTO users (id, email, password_hash) VALUES ($1, $2, $3)`
- `u.ID` must be a pre-generated UUID v4 string (caller generates, not DB gen_random_uuid — we want the ID before the INSERT for event publishing in later milestones).
- `u.CreatedAt` is ignored; the DB sets it via DEFAULT.
- On `pgconn.PgError` with `Code == "23505"` (unique violation): returns `ErrDuplicateEmail`.
- On any other error: returns `fmt.Errorf("userRepository.Create: %w", err)`.

- **Never** returns nil without the row existing in the database. `FindByEmail(ctx context.Context, email string) (User, error)`
- Executes: `SELECT id, email, password_hash, created_at FROM users WHERE email = $1`
- On `pgx.ErrNoRows` : returns `User{}, ErrUserNotFound` .
- On any other error: returns `User{}, fmt.Errorf("userRepository.FindByEmail: %w", err)` .
- Returns the fully populated `User` struct on success.

4.2 `PasswordHasher`

```go
// auth/password.go

// PasswordHasher wraps bcrypt operations.

// All implementations must be stateless and safe for concurrent use.

type PasswordHasher interface {

    // Hash returns a bcrypt hash of plaintext at cost=12.

    // plaintext must be non-empty; returns error if bcrypt fails.

    // The returned string is always 60 ASCII characters.

    Hash(plaintext string) (string, error)

    // Compare returns nil if hash matches plaintext.

    // Returns ErrPasswordMismatch if they do not match.

    // Returns a wrapped error for any other bcrypt failure.

    // This function takes ~100ms at cost=12 — do not call in goroutines

    // holding locks.

    Compare(hash, plaintext string) error

}

// bcryptHasher is the production implementation.

type bcryptHasher struct{}

// NewPasswordHasher returns the production bcrypt implementation.

func NewPasswordHasher() PasswordHasher { return bcryptHasher{} }

// ErrPasswordMismatch is returned by Compare when credentials do not match.

// It is intentionally distinct from bcrypt internal errors.

var ErrPasswordMismatch = errors.New("password mismatch")
```

`Hash` implementation:

```go
func (b bcryptHasher) Hash(plaintext string) (string, error) {

    hash, err := bcrypt.GenerateFromPassword([]byte(plaintext), 12)

    if err != nil {

        return "", fmt.Errorf("passwordHasher.Hash: %w", err)

    }

    return string(hash), nil

}
```

`Compare` implementation:

```go
func (b bcryptHasher) Compare(hash, plaintext string) error {

    err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(plaintext))

    if errors.Is(err, bcrypt.ErrMismatchedHashAndPassword) {

        return ErrPasswordMismatch

    }

    if err != nil {

        return fmt.Errorf("passwordHasher.Compare: %w", err)

    }

    return nil

}
```

4.3 `JWTSigner` and `JWTVerifier`

```go
// auth/jwt.go

// JWTSigner issues signed tokens.
type JWTSigner interface {
    // Sign creates an HS256 JWT from the given Claims.
    // The Iat and Exp fields on the input Claims are set by Sign (caller need not set them).
    // Returns the compact serialized token string on success.
    Sign(claims Claims) (tokenString string, err error)
}

// JWTVerifier validates tokens.
type JWTVerifier interface {
    // Verify parses and validates a compact JWT string.
    // Returns parsed Claims on success.
    // Returns ErrTokenExpired if exp is in the past.
    // Returns ErrTokenInvalid for any other validation failure
    //   (bad signature, wrong issuer, malformed).
    Verify(tokenString string) (Claims, error)
}

// Sentinel errors for token verification.
var (
    ErrTokenExpired = errors.New("token expired")
    ErrTokenInvalid = errors.New("token invalid")
)

// NewJWTSigner returns a JWTSigner using HS256 with the given secret.
// secret must be the raw bytes of JWT_SECRET from the environment.
// Panics if secret is empty — misconfigured secrets must fail at startup, not runtime.
func NewJWTSigner(secret []byte) JWTSigner

// NewJWTVerifier returns a JWTVerifier using HS256 with the given secret.
// secret must be the raw bytes of JWT_SECRET from the environment.
// Panics if secret is empty.
func NewJWTVerifier(secret []byte) JWTVerifier
```

`Sign` implementation detail:

```go
func (s *jwtSigner) Sign(claims Claims) (string, error) {
    now := time.Now().UTC()
    payload := jwt.MapClaims{
        "sub":   claims.Sub,
        "email": claims.Email,
        "iat":   now.Unix(),
        "exp":   now.Add(24 * time.Hour).Unix(),
        "iss":   "url-shortener",
    }
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, payload)
    signed, err := token.SignedString(s.secret)
    if err != nil {
        return "", fmt.Errorf("jwtSigner.Sign: %w", err)
    }
    return signed, nil
}
```

`Verify` implementation detail:

```go
func (v *jwtVerifier) Verify(tokenString string) (Claims, error) {

    token, err := jwt.Parse(tokenString, func(t *jwt.Token) (interface{}, error) {

        if _, ok := t.Method.(*jwt.SigningMethodHMAC); !ok {

            return nil, fmt.Errorf("unexpected signing method: %v", t.Header["alg"])

        }

        return v.secret, nil

    })

    if err != nil {

        // Distinguish expiry from other errors.

        var ve *jwt.ValidationError

        if errors.As(err, &ve) && ve.Errors&jwt.ValidationErrorExpired != 0 {

            return Claims{}, ErrTokenExpired

        }

        return Claims{}, ErrTokenInvalid

    }

    mc, ok := token.Claims.(jwt.MapClaims)

    if !ok || !token.Valid {

        return Claims{}, ErrTokenInvalid

    }

    // Validate issuer explicitly — reject tokens issued by other systems.

    if iss, _ := mc["iss"].(string); iss != "url-shortener" {

        return Claims{}, ErrTokenInvalid

    }

    return Claims{

        Sub:   mc["sub"].(string),

        Email: mc["email"].(string),

        Exp:   time.Unix(int64(mc["exp"].(float64)), 0).UTC(),

        Iat:   time.Unix(int64(mc["iat"].(float64)), 0).UTC(),

        Iss:   "url-shortener",

    }, nil

}
```

Note on `golang-jwt/jwt/v5`: In v5 the `ValidationError` API changed. Use `errors.Is(err, jwt.ErrTokenExpired)` instead of the bitmask approach shown above (which is v4). The implementation must match the version declared in `go.mod`. The spec shows the v4 pattern; adapt as needed after pinning the version.

4.4 `UserService`

```go
// service/user_service.go

// UserService orchestrates the repository, password hasher, and JWT signer.

// It owns all business logic for the auth domain.

// It must not reference http.Request or http.ResponseWriter — that belongs to handlers.

type UserService struct {

    repo    repository.UserRepository

    hasher  auth.PasswordHasher

    signer  auth.JWTSigner

    log     zerolog.Logger

}

func NewUserService(

    repo repository.UserRepository,

    hasher auth.PasswordHasher,

    signer auth.JWTSigner,

    log zerolog.Logger,

) *UserService

// Register creates a new user. Returns the created User (with ID and CreatedAt populated).

// Error types:

//   - repository.ErrDuplicateEmail → caller maps to 409

//   - validation error (invalid email, short password) → caller maps to 400

//   - any other error → caller maps to 500

func (s *UserService) Register(ctx context.Context, email, password string) (repository.User, error)

// Login verifies credentials and returns a signed JWT string and expiry time.

// Error types:

//   - auth.ErrPasswordMismatch or repository.ErrUserNotFound → caller maps to 401

//     (same error, no distinction — prevents email enumeration)

//   - any other error → caller maps to 500

func (s *UserService) Login(ctx context.Context, email, password string) (token string, expiresAt time.Time, err error)
```

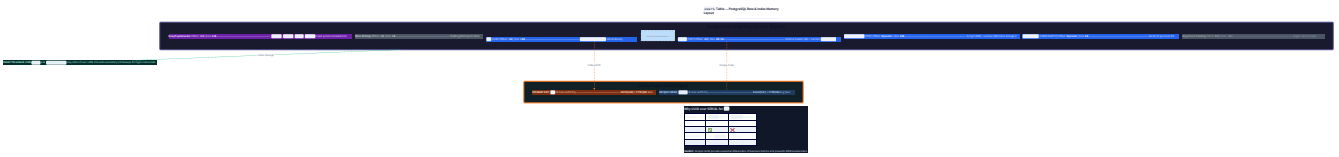
`Register` logic:

```
PROCEDURE Register(ctx, email, password):
  1. Normalize email: strings.ToLower(strings.TrimSpace(email))
  2. Validate email: use net/mail.ParseAddress(email) — returns error on invalid format
  3. Validate password length: len(password) < 8 → return ErrValidation("password must be at least 8 characters")
  4. Hash password: hash, err = hasher.Hash(password)
     IF err != nil → return wrapped error (→ 500)
  5. Generate user ID: id = uuid.New().String() — use google/uuid package
  6. Construct User{ID: id, Email: email, PasswordHash: hash}
  7. repo.Create(ctx, user)
     IF ErrDuplicateEmail → return as-is (→ 409)
     IF other error → return wrapped (→ 500)
  8. Return the constructed User (CreatedAt will be zero; that is acceptable — caller only needs ID and email)
END PROCEDURE
```

`Login` logic:

```
PROCEDURE Login(ctx, email, password):
  1. Normalize email: strings.ToLower(strings.TrimSpace(email))
  2. user, err = repo.FindByEmail(ctx, email)
     IF ErrUserNotFound → return "", time.Time{}, ErrAuthFailed
     IF other error → return wrapped (→ 500)
  3. err = hasher.Compare(user.PasswordHash, password)
     IF ErrPasswordMismatch → return "", time.Time{}, ErrAuthFailed
     IF other error → return wrapped (→ 500)
  4. claims = auth.Claims{Sub: user.ID, Email: user.Email}
  5. tokenStr, err = signer.Sign(claims)
     IF err → return wrapped (→ 500)
  6. expiresAt = time.Now().UTC().Add(24 * time.Hour)
  7. Return tokenStr, expiresAt, nil
END PROCEDURE
// ErrAuthFailed is the single sentinel for both "email not found" and "wrong password".
// It must never distinguish between the two cases.
var ErrAuthFailed = errors.New("invalid credentials")
```

Why one sentinel for both failures: Returning different errors for "email not found" vs "wrong password" would allow an attacker to enumerate registered emails by observing the response. `ErrAuthFailed` collapses both into an identical 401 response.



4.5 `shared/middleware` — JWT Middleware

GO

```go
// shared/middleware/jwt_middleware.go

package middleware

// contextKey is an unexported type for context keys to avoid collisions.

type contextKey string

const ClaimsContextKey contextKey = "jwt_claims"

// RequireAuth returns an http.Handler that:

//  1. Extracts the Bearer token from the Authorization header.

//  2. Verifies the token using verifier.

//  3. Stores the parsed Claims in the request context under ClaimsContextKey.

//  4. Calls next if verification succeeds.

//  5. Returns 401 with {"error":"unauthorized"} if token is missing, malformed, expired, or invalid.

//

// This middleware is designed to be imported by any service that needs to

// validate JWTs issued by user-service. The verifier must be constructed

// with the same JWT_SECRET used by user-service.

func RequireAuth(verifier auth.JWTVerifier, log zerolog.Logger) func(http.Handler) http.Handler

// ClaimsFromContext retrieves the verified Claims from a request context.

// Returns zero-value Claims and false if not present.

// Handlers call this after RequireAuth has run.

func ClaimsFromContext(ctx context.Context) (auth.Claims, bool)
```

`RequireAuth` implementation:

```go
func RequireAuth(verifier auth.JWTVerifier, log zerolog.Logger) func(http.Handler) http.Handler {

    return func(next http.Handler) http.Handler {

        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

            authHeader := r.Header.Get("Authorization")

            if authHeader == "" || !strings.HasPrefix(authHeader, "Bearer ") {

                writeJSON(w, http.StatusUnauthorized, ErrorResponse{Error: "unauthorized"})

                return

            }

            tokenStr := strings.TrimPrefix(authHeader, "Bearer ")

            claims, err := verifier.Verify(tokenStr)

            if err != nil {

                // Do NOT distinguish between expired and invalid in the response.

                // Log the specific reason for debugging; return generic 401 to client.

                log.Warn().Err(err).Str("path", r.URL.Path).Msg("jwt verification failed")

                writeJSON(w, http.StatusUnauthorized, ErrorResponse{Error: "unauthorized"})

                return

            }

            ctx := context.WithValue(r.Context(), ClaimsContextKey, claims)

            next.ServeHTTP(w, r.WithContext(ctx))

        })

    }

}
```

Edge cases:

- `Authorization: Bearer ` (trailing space, empty token) → `Verify` will fail → 401.
- `Authorization: bearer TOKEN` (lowercase) → `HasPrefix("Bearer ")` fails → 401. The spec requires exact case.
- Multiple `Authorization` headers → `r.Header.Get` returns only the first value per Go's `net/http` semantics → safe.

---

5. Algorithm Specification

5.1 Input Validation

Email validation:

```
PROCEDURE ValidateEmail(email string) error:
  trimmed = strings.TrimSpace(email)
  IF len(trimmed) == 0 → return validation error "email is required"
  _, err = mail.ParseAddress(trimmed)
  IF err != nil → return validation error "email format is invalid"
  return nil
// Note: mail.ParseAddress accepts "Name <email>" format.
// To reject display names, check that the returned address equals the trimmed input.
// Implementation: addr, err := mail.ParseAddress(trimmed); if addr.Address != trimmed → error
```

Password validation:

```
PROCEDURE ValidatePassword(password string) error:
  IF len(password) < 8 → return validation error "password must be at least 8 characters"
  // No other complexity rules per spec. No maximum length check is needed because
  // bcrypt truncates input at 72 bytes internally — document this behavior but
  // do not reject longer passwords (bcrypt handles it silently).
  return nil
```

Why `net/mail.ParseAddress` and not regex: RFC 5322 email syntax is not regular. The standard library's parser correctly handles quoted local parts, international domains (after IDNA encoding), and other edge cases that naive regexes reject or wrongly accept.

5.2 POST /register Handler Flow

```
PROCEDURE HandleRegister(w http.ResponseWriter, r *http.Request):
  1. Decode JSON body into RegisterRequest.
     IF body is empty or malformed JSON → 400 {"error":"invalid request body"}
     Limit body size: http.MaxBytesReader(w, r.Body, 1024) to prevent oversized payloads.
  2. Validate email and password (§ 5.1).
     IF invalid → 400 {"error":"<specific validation message>"}
  3. Call userService.Register(r.Context(), req.Email, req.Password).
     IF ErrDuplicateEmail → 409 {"error":"email already registered"}
     IF ErrValidation → 400 {"error":"<message>"}
     IF other error → log.Error, return 500 {"error":"internal server error"}
  4. On success → 201 {"user_id":"<uuid>","email":"<email>"}
END PROCEDURE
```

5.3 POST /login Handler Flow

```
PROCEDURE HandleLogin(w http.ResponseWriter, r *http.Request):
  1. Decode JSON body into LoginRequest.
     IF malformed → 400 {"error":"invalid request body"}
  2. No validation of email format or password length here —
     invalid credentials simply fail the auth check, returning 401.
     (Omitting validation prevents leaking "this email format doesn't even exist" information.)
  3. Call userService.Login(r.Context(), req.Email, req.Password).
     IF ErrAuthFailed → 401 {"error":"invalid credentials"}
       -- identical response for wrong password AND unknown email --
     IF other error → log.Error, return 500 {"error":"internal server error"}
  4. On success → 200 {"token":"<jwt>","expires_at":"<RFC3339>"}
     expires_at formatted with time.RFC3339: "2026-03-03T10:00:00Z"
END PROCEDURE
```

5.4 GET /me Handler Flow

```
PROCEDURE HandleMe(w http.ResponseWriter, r *http.Request):
  -- This handler is mounted BEHIND RequireAuth middleware.
  -- By the time this handler runs, the JWT is already verified.
  1. claims, ok = middleware.ClaimsFromContext(r.Context())
     IF !ok → 401 {"error":"unauthorized"}
     (Defensive check; RequireAuth should prevent this path.)
  2. Return 200 {"user_id": claims.Sub, "email": claims.Email}
     -- No DB call. All data comes from the verified token claims.
END PROCEDURE
```

5.5 Route Registration in `main.go`

```go
mux := http.NewServeMux()

// Public routes — no auth required
mux.HandleFunc("POST /register", handler.NewRegisterHandler(userSvc, log))
mux.HandleFunc("POST /login",    handler.NewLoginHandler(userSvc, log))
mux.HandleFunc("GET /health",    handler.NewHealthHandler("user-service", log))

// Protected routes — wrapped with RequireAuth middleware
verifier := auth.NewJWTVerifier([]byte(jwtSecret))
requireAuth := middleware.RequireAuth(verifier, log)
mux.Handle("GET /me", requireAuth(http.HandlerFunc(handler.NewMeHandler(log))))
```

Go 1.22+ routing note: The `net/http` mux in Go 1.22 supports method-prefixed patterns (`"POST /register"`). Since `go.mod` declares `go 1.23`, this is available. Do not use `http.ServeMux` workarounds; use the native method patterns.

6. Threat Model

6.1 Attack Surface

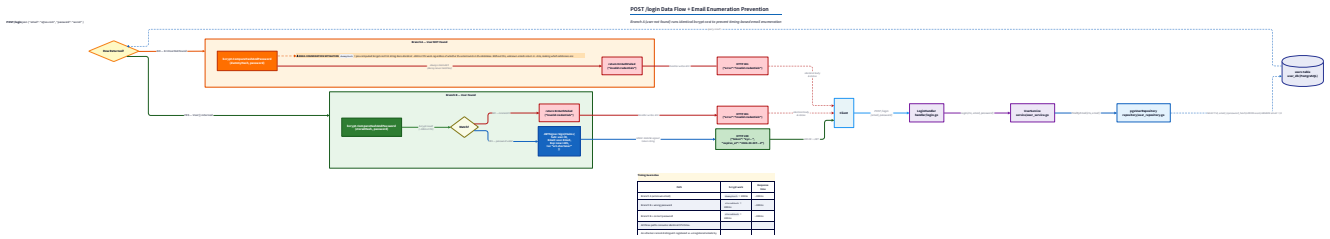
Attack	Vector	Mitigation
Password exfiltration	DB breach	bcrypt cost=12; hash is computationally infeasible to reverse
Email enumeration via login	Different 401 bodies for "no user" vs "wrong password"	`ErrAuthFailed` collapses both cases; same response body and same 401 status
Email enumeration via timing	bcrypt takes 100ms for valid email, 0ms for unknown	Always run `hasher.Compare` even for unknown emails (see § 6.2)
JWT secret exposure	Hardcoded secret in source	`mustGetEnv("JWT_SECRET")` — no fallback, no default
Token forgery	Weak HMAC key	Document minimum key entropy: JWT_SECRET must be ≥ 32 random bytes in `.env.example`
Algorithm confusion	JWT header `alg:none` or RSA confusion	`Verify` checks `t.Method.(*jwt.SigningMethodHMAC)` — rejects any other algorithm
Long password DoS	bcrypt memory allocation for 10MB body	`http.MaxBytesReader(w, r.Body, 1024)` limits request body before any processing
Concurrent registration race	Two goroutines both pass the uniqueness check before DB insert	Unique constraint on `users.email` enforced at DB level; application maps `23505` → 409

6.2 Timing Attack Mitigation for Login

When an email does not exist, skip `hasher.Compare` is a timing leak: the attacker observes that responses for unknown emails return in ~0ms while valid emails take ~100ms.

```go
// In UserService.Login — always run a compare operation to equalize timing.

user, err := s.repo.FindByEmail(ctx, email)

if errors.Is(err, repository.ErrUserNotFound) {

    // Run a dummy compare to burn the same ~100ms as a real compare.

    // The hash is a valid bcrypt string that will never match anything.

    _ = s.hasher.Compare(dummyHash, password)

    return "", time.Time{}, ErrAuthFailed

}
```

```go
// Declare in user_service.go as a package-level constant.

// Generated once with bcrypt.GenerateFromPassword([]byte("dummypassword"), 12).

// Never changes at runtime — it is not a security secret, just a timing dummy.

const dummyHash = "$2a$12$LQv3c1yqBWVHxkd0LHAkCOYz6TtxMQJqhN8/LewdBPj6hsxq.bxi."
```



7. Error Handling Matrix

Error	Detected By	HTTP Status	Response Body	Logged?	Notes
`VALIDATION_ERROR` (email format)	`ValidateEmail` in `UserService.Register`	400	`{"error":"email format is invalid"}`	No	User mistake; not logged
`VALIDATION_ERROR` (password < 8)	`ValidatePassword` in `UserService.Register`	400	`{"error":"password must be at least 8 characters"}`	No	User mistake; not logged
`VALIDATION_ERROR` (empty body)	JSON decoder in handler	400	`{"error":"invalid request body"}`	No	
`VALIDATION_ERROR` (body > 1024B)	`http.MaxBytesReader`	400	`{"error":"request body too large"}`	No	
`DUPLICATE_EMAIL`	`pgxUserRepository.Create` → `23505`	409	`{"error":"email already registered"}`	No	Expected race; not an error
`AUTH_FAILED` (unknown email)	`UserService.Login` after `ErrUserNotFound`	401	`{"error":"invalid credentials"}`	No	Same body as wrong password
`AUTH_FAILED` (wrong password)	`UserService.Login` after `ErrPasswordMismatch`	401	`{"error":"invalid credentials"}`	No	Same body as unknown email
`TOKEN_EXPIRED`	`JWTVerifier.Verify` → `ErrTokenExpired`	401	`{"error":"unauthorized"}`	Warn	Log path for debugging
`TOKEN_INVALID`	`JWTVerifier.Verify` → `ErrTokenInvalid`	401	`{"error":"unauthorized"}`	Warn	Potential attack; log it
`TOKEN_MISSING`	`RequireAuth` → missing/malformed header	401	`{"error":"unauthorized"}`	No	Unauthenticated request
`DB_ERROR` (unexpected)	Any repo call	500	`{"error":"internal server error"}`	**Error**	Log full error; never expose to client
`BCRYPT_ERROR` (unexpected)	`hasher.Hash` or `hasher.Compare`	500	`{"error":"internal server error"}`	**Error**	Should never occur with valid input
`JWT_SIGN_ERROR`	`signer.Sign`	500	`{"error":"internal server error"}`	**Error**	Indicates misconfigured secret
500 response pattern (used consistently in all handlers):					

```go
func writeInternalError(w http.ResponseWriter, log zerolog.Logger, err error, msg string) {
    log.Error().Err(err).Msg(msg)
    writeJSON(w, http.StatusInternalServerError, ErrorResponse{Error: "internal server error"})
}
```

Shared `writeJSON` helper:

```go
// writeJSON encodes v as JSON and writes it to w with the given status code.
// Sets Content-Type: application/json.
// Panics are impossible: encoding a fixed struct to JSON cannot fail.
func writeJSON(w http.ResponseWriter, status int, v any) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    json.NewEncoder(w).Encode(v) // error intentionally ignored; nothing we can do after WriteHeader
}
```

8. Implementation Sequence with Checkpoints

Phase 1 — users table migration + UserRepository (1–1.5 hr)

1. Write `migrations/001_create_users.sql` (§ 3.1).
2. Run the migration against `user_db`: `docker exec -i user_db psql -U user_user -d user_db < services/user-service/migrations/001_create_users.sql`
3. Write `repository/user_repository.go`: define `User` struct, sentinel errors, `UserRepository` interface, `pgxUserRepository` struct, `NewPgxUserRepository`, `Create`, `FindByEmail`.
4. Write `repository/user_repository_test.go`: integration tests (§ 9.2).
5. Run `go test ./repository/... -v` with `user_db` running. **Checkpoint:** `go test ./repository/... -v` passes all tests. `TestCreate_DuplicateEmail` returns `ErrDuplicateEmail` (not a 500). `TestFindByEmail_NotFound` returns `ErrUserNotFound`. Run `docker exec user_db psql -U user_user -d user_db -c "\d users"` — table and indexes visible.

Phase 2 — PasswordHasher + JWTSigner/JWTVerifier (1–1.5 hr)

1. Add `golang.org/x/crypto` to `go.mod`: `go get golang.org/x/crypto`.
2. Add `github.com/golang-jwt/jwt/v5`: `go get github.com/golang-jwt/jwt/v5`.
3. Add `github.com/google/uuid`: `go get github.com/google/uuid`.
4. Write `auth/password.go` (§ 4.2).
5. Write `auth/password_test.go` (§ 9.3).
6. Write `auth/jwt.go` (§ 4.3).
7. Write `auth/jwt_test.go` (§ 9.4).
8. Run `go test ./auth/...`. **Checkpoint:** `go test ./auth/... -v` passes all tests. `TestBcryptCost` verifies cost=12 is embedded in hash. `TestJWTRoundTrip` verifies Sign→Verify returns identical claims. `TestVerify_ExpiredToken` returns `ErrTokenExpired`. `TestVerify_TamperedSignature` returns `ErrTokenInvalid`. `TestVerify_WrongIssuer` returns `ErrTokenInvalid`.

Phase 3 — POST /register + POST /login Handlers (1.5–2 hr)

1. Write `service/user_service.go`: `UserService` struct, `NewUserService`, `Register`, `Login`, dummy hash constant (§ 4.4, § 6.2).
2. Write `handler/register.go`: `RegisterRequest`, `RegisterResponse`, `NewRegisterHandler` (§ 5.2).
3. Write `handler/login.go`: `LoginRequest`, `LoginResponse`, `NewLoginHandler` (§ 5.3).
4. Write the `writeJSON` and `writeInternalError` helpers (§ 7) — put them in `handler/helpers.go`.
5. Wire together in `main.go`: connect DB, construct `pgxUserRepository`, `bcryptHasher`, `jwtSigner`, `UserService`, register routes.
6. Run `docker compose up --build user-service`. **Checkpoint:**

```bash
# Register
curl -s -X POST http://localhost:8083/register \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"password123"}' | jq .
# Expected: {"user_id":"<uuid>","email":"alice@example.com"} with 201
# Duplicate
curl -s -o /dev/null -w "%{http_code}" -X POST http://localhost:8083/register \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"password123"}'
# Expected: 409
# Login
curl -s -X POST http://localhost:8083/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"password123"}' | jq .
# Expected: {"token":"eyJ...","expires_at":"2026-..."}
# Wrong password
curl -s -o /dev/null -w "%{http_code}" -X POST http://localhost:8083/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"wrongpassword"}'
# Expected: 401
```

Phase 4 — JWT Middleware in `shared/middleware/` (1–1.5 hr)

1. Create `shared/middleware/` directory.
2. Add `github.com/golang-jwt/jwt/v5` to `shared/go.mod` (since middleware imports auth types).
3. Write `shared/middleware/jwt_middleware.go` (§ 4.5).
4. Write `shared/middleware/jwt_middleware_test.go` (§ 9.5).
5. Run `go test ./...` from `shared/`.
6. Update each service's `go.mod` to import `shared/middleware` via the existing replace directive. **Checkpoint:** `go test ./... -v` in `shared/` passes. `TestRequireAuth_MissingHeader` → 401. `TestRequireAuth_ValidToken` → calls next handler. `TestRequireAuth_ExpiredToken` → 401. `TestClaimsFromContext` → returns claims stored by middleware. **Note on shared package design:** The `shared/middleware` package must import `auth.JWTVerifier` and `auth.Claims`. These types are defined in each service's `auth/` package, which creates an import cycle if `shared/middleware` imports a specific service. Resolution: move `auth.JWTVerifier` and `auth.Claims` interface/struct definitions into `shared/auth/` so that both user-service and `shared/middleware` can import from a neutral location. **Revised structure addition:**

```
shared/
  auth/
    types.go    # JWTVerifier interface, Claims struct, sentinel errors
  middleware/
    jwt_middleware.go
```

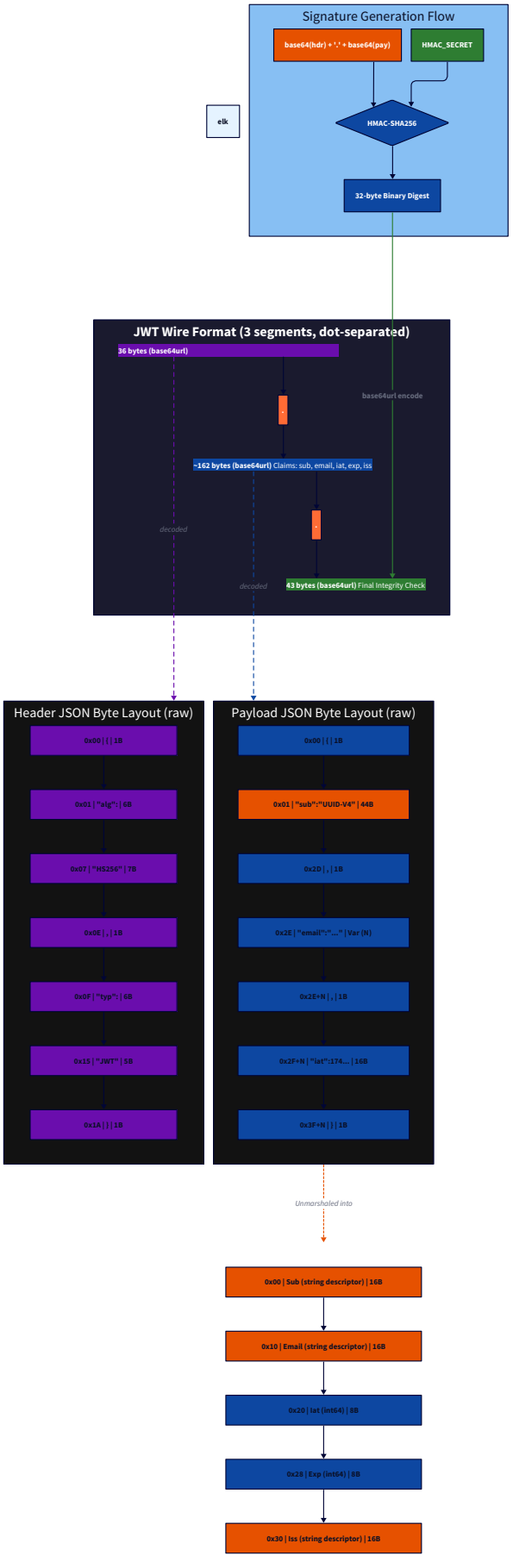
User-service's `auth/jwt.go` implements `shared/auth.JWTVerifier` interface. This eliminates the cycle.

Phase 5 — GET /me + Integration Test (1–1.5 hr)

1. Write `handler/me.go` (§ 5.4).
2. Add the `/me` route behind `RequireAuth` middleware to `main.go`.
3. Write `handler/handler_test.go`: full round-trip integration test (§ 9.6).
4. Run the integration test against the live Docker stack. **Checkpoint:**

```bash
# Full round trip
TOKEN=$(curl -s -X POST http://localhost:8083/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"password123"}' | jq -r .token)
curl -s http://localhost:8083/me \
  -H "Authorization: Bearer $TOKEN" | jq .
# Expected: {"user_id":"<uuid>","email":"alice@example.com"}
# Expired / invalid token
curl -s -o /dev/null -w "%{http_code}" http://localhost:8083/me \
  -H "Authorization: Bearer invalidtoken"
# Expected: 401
```

JWT Claims Memory Layout — HS256 Wire Format



Memory Alignment Note

Struct fields are aligned to 8-byte boundaries. The total size of 64B allows the CPU to fetch the entire claims set in a single cache line transaction, minimizing memory latency.

Segment Size Summary (HS256)			

9. Test Specification

9.1 Test File Organization

All tests in this module follow the pattern of same-package unit tests for pure logic, and `_test` package (external) tests for integration against the live DB or HTTP server.

9.2 `repository/user_repository_test.go`

```go
// Build tag: //go:build integration — run with: go test -tags integration ./repository/...

// Requires: user_db container running. Read DSN from TEST_DATABASE_DSN env var.

// TestCreate_HappyPath: insert user → row exists in DB with correct fields

// TestCreate_DuplicateEmail: insert same email twice → second returns ErrDuplicateEmail

// TestCreate_UUIDPreserved: user.ID provided by caller is stored verbatim (not overwritten)

// TestCreate_EmailNormalization: upper-case email in → lower-case stored

//   (if normalization is done in service; verify service layer, not repo)

// TestFindByEmail_HappyPath: insert user → FindByEmail returns same user

// TestFindByEmail_NotFound: FindByEmail on unknown email → ErrUserNotFound

// TestFindByEmail_CaseSensitivity: store "alice@example.com" → find "ALICE@EXAMPLE.COM" → not found

//   (repository is case-sensitive; normalization is service responsibility)

func TestCreate_HappyPath(t *testing.T)

func TestCreate_DuplicateEmail(t *testing.T)

func TestCreate_UUIDPreserved(t *testing.T)

func TestFindByEmail_HappyPath(t *testing.T)

func TestFindByEmail_NotFound(t *testing.T)

func TestFindByEmail_CaseSensitivity(t *testing.T)
```

9.3 `auth/password_test.go`

```go
// TestHash_ProducesValidBcrypt: Hash("mypassword") → string starts with "$2a$12$"

// TestHash_DifferentSalts: Hash("same") called twice → two different hashes

// TestCompare_HappyPath: Hash(p) → Compare(hash, p) returns nil

// TestCompare_WrongPassword: Compare(hash, "wrong") → ErrPasswordMismatch

// TestCompare_EmptyPassword: Hash("") returns error (bcrypt rejects empty)

// TestBcryptCost: hash output contains "$2a$12$" (cost 12 confirmed)

// BenchmarkHash: bcrypt at cost=12 should take 80-200ms (document timing)

func TestHash_ProducesValidBcrypt(t *testing.T)

func TestHash_DifferentSalts(t *testing.T)

func TestCompare_HappyPath(t *testing.T)

func TestCompare_WrongPassword(t *testing.T)

func TestBcryptCost(t *testing.T)

func BenchmarkHash(b *testing.B)
```

9.4 `auth/jwt_test.go`

```go
// TestSign_ReturnsThreeParts: token string contains exactly two "." chars

// TestSign_PayloadFields: decode payload (no verify) → sub, email, iss, iat, exp present

// TestSign_ExpiryIs24h: exp - iat == 86400 seconds

// TestVerify_HappyPath: Sign → Verify → claims match original

// TestVerify_ExpiredToken: manually set exp to past → Verify returns ErrTokenExpired

// TestVerify_TamperedSignature: alter one char in signature → ErrTokenInvalid

// TestVerify_WrongAlgorithm: header alg=none → ErrTokenInvalid

// TestVerify_WrongIssuer: sign with iss="other" → ErrTokenInvalid

// TestVerify_EmptySecret: NewJWTSigner(nil) → panics

// TestVerify_SubIsUUID: claims.Sub from verified token is parseable by uuid.Parse

func TestSign_ReturnsThreeParts(t *testing.T)

func TestSign_ExpiryIs24h(t *testing.T)

func TestVerify_HappyPath(t *testing.T)

func TestVerify_ExpiredToken(t *testing.T)

func TestVerify_TamperedSignature(t *testing.T)

func TestVerify_WrongIssuer(t *testing.T)

func TestVerify_EmptySecret(t *testing.T) // expects panic; use recover()
```

9.5 `shared/middleware/jwt_middleware_test.go`

```go
// All tests use httptest.NewRecorder and httptest.NewRequest.

// A real JWTVerifier (not a mock) is used to produce valid/invalid tokens.

// TestRequireAuth_ValidToken: valid token → next called, claims in context

// TestRequireAuth_MissingHeader: no Authorization header → 401

// TestRequireAuth_MalformedHeader: "Token abc" (not Bearer) → 401

// TestRequireAuth_ExpiredToken: expired token → 401, next NOT called

// TestRequireAuth_InvalidSignature: tampered token → 401

// TestRequireAuth_EmptyBearerValue: "Bearer " (nothing after space) → 401

// TestClaimsFromContext_Present: context with claims → returns claims, true

// TestClaimsFromContext_Absent: empty context → returns zero Claims, false

func TestRequireAuth_ValidToken(t *testing.T)

func TestRequireAuth_MissingHeader(t *testing.T)

func TestRequireAuth_MalformedHeader(t *testing.T)

func TestRequireAuth_ExpiredToken(t *testing.T)

func TestRequireAuth_InvalidSignature(t *testing.T)

func TestClaimsFromContext_Present(t *testing.T)

func TestClaimsFromContext_Absent(t *testing.T)
```

9.6 `handler/handler_test.go` — Integration Test

```go
// Build tag: //go:build integration

// Requires: user-service running on TEST_USER_SERVICE_URL (default: http://localhost:8083)

// TestRegisterLoginMe_RoundTrip:

//   1. POST /register → 201, capture user_id

//   2. POST /login → 200, capture token and expires_at

//   3. Verify expires_at is approximately 24h from now (within 60s tolerance)

//   4. GET /me with token → 200, user_id and email match registration values

//   5. GET /me with no token → 401

//   6. GET /me with expired token → 401

// TestRegister_DuplicateEmail:

//   1. POST /register (email A) → 201

//   2. POST /register (email A again) → 409, body contains "email already registered"

// TestRegister_InvalidEmail:

//   POST /register with "notanemail" → 400

// TestRegister_ShortPassword:

//   POST /register with 7-char password → 400

// TestLogin_WrongPassword:

//   POST /login with correct email, wrong password → 401

//   Response body must equal 401 body for unknown email (same message)

// TestLogin_UnknownEmail:

//   POST /login with unknown email → 401

//   Response body must equal 401 body for wrong password (same message)

func TestRegisterLoginMe_RoundTrip(t *testing.T)

func TestRegister_DuplicateEmail(t *testing.T)

func TestRegister_InvalidEmail(t *testing.T)

func TestRegister_ShortPassword(t *testing.T)

func TestLogin_WrongPassword(t *testing.T)

func TestLogin_UnknownEmail(t *testing.T)
```

10. Performance Targets

Operation	Target	How to Measure
`POST /register` p99	< 200ms	`wrk -t2 -c5 -d30s -s register.lua http://localhost:8083/register` — bcrypt dominates
`POST /login` (valid) p99	< 200ms	Same `wrk` script with login payload
`POST /login` (unknown email) p99	80–200ms	Must NOT be near-zero — timing equalization required (§ 6.2)
`GET /me` p99	< 5ms	`wrk -t4 -c50 -d10s` with valid token — no DB access
JWT `Verify` (HMAC-SHA256)	< 1ms	`go test -bench=BenchmarkVerify ./auth/...`
JWT `Sign`	< 1ms	`go test -bench=BenchmarkSign ./auth/...`
`bcrypt.Hash` (cost=12)	80–150ms	`go test -bench=BenchmarkHash ./auth/...` on commodity hardware
`go build ./...`	< 10s incremental	CI warm build
`GET /health` p99	< 10ms	(inherited from M1)
bcrypt cost=12 justification: At cost=12, a single hash takes ~100ms on a modern CPU. This bounds `/register` and `/login` to ~100–200ms total (DB query + bcrypt). This is acceptable for auth endpoints which are not on the hot redirect path. The hot path (`GET /:code` redirect) in url-service does not touch user-service at all.		
Concurrency note: At cost=12, bcrypt saturates a CPU core for ~100ms. With default GOMAXPROCS, a sustained load of 10 concurrent login requests will queue behind each other. This is a deliberate security tradeoff, not a bug. The service is not designed to serve thousands of logins per second — it is designed to make brute-force attacks computationally expensive.		

11. Concurrency Specification

11.1 Handler Concurrency

`net/http` launches each request in its own goroutine. The following invariants must hold:

- `UserService` is stateless: no shared mutable fields. The `dummyHash` constant is read-only. Safe for concurrent use.
- `pgxUserRepository` stores only a `*pgxpool.Pool` reference. `pgxpool` is safe for concurrent use — each `Acquire(ctx)` call gets an exclusive connection from the pool.
- `bcryptHasher` is stateless. `bcrypt.GenerateFromPassword` and `bcrypt.CompareHashAndPassword` are goroutine-safe per `golang.org/x/crypto` documentation.
- `jwtSigner` and `jwtVerifier` store only `[]byte` (the secret), which is immutable after construction. Safe for concurrent use.
- No global variables in the handler, service, or repository packages (only the `log` instance, which `zerolog.Logger` is value-safe by design).

11.2 Blocking Operations

`bcrypt.CompareHashAndPassword` at cost=12 blocks its goroutine for ~100ms. This is acceptable because:

1. Auth endpoints are not on the hot redirect path.
2. Go's scheduler will context-switch other goroutines while this one blocks.
3. The pgxpool connection is held for DB query time only (< 5ms), not during bcrypt computation — acquire, query, release, then bcrypt. **Connection acquisition pattern:**

```go
// In UserService.Login — release DB connection before bcrypt, not after.

user, err := s.repo.FindByEmail(ctx, email) // acquires + releases pool conn internally

if err != nil { ... }

// DB connection is released here. bcrypt runs without holding any pool connection.

err = s.hasher.Compare(user.PasswordHash, password)
```

12. Environment Variable Reference for This Module

Variable	Required	Example	Description
`JWT_SECRET`	Yes	`supersecretkey32bytesminimum!!!`	HMAC-SHA256 signing key. Must be ≥ 32 bytes. No default — service exits if unset.
`PORT`	Yes	`8083`	HTTP listen port (inherited from M1)
`DATABASE_DSN`	Yes	`postgres://user_user:user_secret@user_db:5432/user_db?sslmode=disable`	(inherited from M1)
`SERVICE_NAME`	Yes	`user-service`	Embedded in log lines (inherited from M1)
`JWT_SECRET` startup validation (add to `main.go`):			

```go
jwtSecret := mustGetEnv("JWT_SECRET")

if len(jwtSecret) < 32 {

    log.Fatal().Msg("JWT_SECRET must be at least 32 characters")

    os.Exit(1)

}
```

`.env.example` addition:

```
JWT_SECRET=change-me-to-at-least-32-random-bytes-in-production
```

Cross-service secret sharing: The same `JWT_SECRET` value must be set in the environment of url-service, analytics-service, notification-service, and the gateway. They use it only to construct a `JWTVerifier` — never a `JWTSigner`. Add `JWT_SECRET` to each service's `docker-compose.yml` environment block in M3–M5.

Technical Design Specification

Module: URL Service — Shorten, Redirect, CRUD + Domain Event Publishing

Module ID: `url-shortener-m3`

1. Module Charter

The URL Service is the core business logic service of the URL shortener system. It accepts long URLs from authenticated users, generates collision-resistant 7-character base62 short codes, stores them in its private PostgreSQL database, serves 301 redirects via a Redis read-through cache, and publishes domain events (`URLCreatedEvent` , `URLClickedEvent` , `URLDeletedEvent`) to RabbitMQ using the outbox pattern to guarantee at-least-once delivery without coupling the write path to broker availability. This module does **not** call the analytics-service, notification-service, or user-service at runtime. It does not share its PostgreSQL database with any other service. It does not verify user existence — the JWT `sub` claim is trusted as the owner identifier. It does not implement URL preview, custom domain

mapping, QR code generation, link health checking, or bulk import. The outbox poller is the only component that touches RabbitMQ; HTTP handlers write only to PostgreSQL. **Upstream dependencies:** M1 foundation (Docker Compose stack, `shared/events`, `shared/logger`, Redis container, RabbitMQ container, `url_db` PostgreSQL container). M2 `shared/auth` package for `JWTVerifier` and `Claims`; `shared/middleware` for `RequireAuth`. **Downstream dependencies:** analytics-service (M4) and notification-service (M5) consume events published by the outbox poller. The gateway (M5) proxies all requests to this service. No service calls url-service synchronously except the gateway. **Invariants that must always hold after this milestone:**

- Every URL creation and its corresponding `URLCreatedEvent` are written in a single PostgreSQL transaction — they either both commit or both roll back. No URL exists without an outbox row; no outbox row exists for a URL that was not created.
- The same atomic-transaction invariant holds for clicks: every redirect and its `URLClickedEvent` are written together. If the service crashes after the redirect response is sent but before the commit, the event is not published (acceptable: the redirect still happened; the loss of one click event is a known at-least-once tradeoff documented below).
- A Redis cache error never causes a redirect failure. The cache is always optional; PostgreSQL is the source of truth.
- Short codes are generated from `crypto/rand`; `math/rand` is never used in the code generation path.
- A user can only delete their own URLs. The JWT `sub` must match `urls.user_id`; mismatches return 403, not 404.
- Expired URLs return 410, not 404. Inactive URLs (soft-deleted) return 410, not 404. Only truly absent codes return 404.

---

2. File Structure

Create files in the numbered order below. All paths are relative to the monorepo root.

```
url-shortener/
│
├── services/url-service/
│   ├── migrations/
│   │   ├── 01  001_create_urls.sql
│   │   └── 02  002_create_outbox.sql
│   ├── codegen/
│   │   ├── 03  codegen.go              # Base62 alphabet, Generate(), collision retry
│   │   └── 04  codegen_test.go
│   ├── repository/
│   │   ├── 05  url_repository.go       # URLRepository interface + pgx implementation
│   │   ├── 06  url_repository_test.go
│   │   ├── 07  outbox_repository.go    # OutboxRepository interface + pgx implementation
│   │   └── 08  outbox_repository_test.go
│   ├── cache/
│   │   ├── 09  cache.go                # CacheClient interface + Redis implementation
│   │   └── 10  cache_test.go
│   ├── amqp/
│   │   ├── 11  publisher.go            # AMQPPublisher interface + amqp091 implementation
│   │   └── 12  publisher_test.go
│   ├── outbox/
│   │   ├── 13  poller.go               # OutboxPoller: coordinator + 3 workers
│   │   └── 14  poller_test.go
│   ├── service/
│   │   └── 15  url_service.go          # URLService: orchestrates repo, codegen, cache, outbox
│   ├── handler/
│   │   ├── 16  shorten.go             # POST /shorten
│   │   ├── 17  redirect.go            # GET /:code
│   │   ├── 18  list.go                # GET /urls
│   │   ├── 19  delete.go              # DELETE /urls/:code
│   │   ├── 20  health.go              # GET /health (update from M1)
│   │   ├── 21  helpers.go             # writeJSON, writeError, extractIP, hashIP
│   │   └── 22  handler_test.go        # integration tests
│   └── 23  main.go                    # wiring: env, DB, Redis, RabbitMQ, routes, outbox start
```

Total new files: 23 (excluding auto-generated `go.sum` changes).

3. Complete Data Model

3.1 PostgreSQL Schema — `migrations/001_create_urls.sql`

```sql
-- migrations/001_create_urls.sql

BEGIN;

CREATE TABLE IF NOT EXISTS urls (

    id           UUID         PRIMARY KEY DEFAULT gen_random_uuid(),

    short_code   VARCHAR(10)  NOT NULL,

    original_url TEXT         NOT NULL,

    user_id      UUID         NOT NULL,

    created_at   TIMESTAMPTZ  NOT NULL DEFAULT now(),

    expires_at   TIMESTAMPTZ  NULL,           -- NULL means no expiry

    is_active    BOOLEAN      NOT NULL DEFAULT true

);

-- Primary lookup: redirect hot path. Must be unique; drives cache key.

CREATE UNIQUE INDEX IF NOT EXISTS idx_urls_short_code

    ON urls(short_code);

-- Cursor pagination: user's URL list, newest first.

-- Composite index matches the WHERE user_id = $1 AND id < $2 ORDER BY id DESC query.

CREATE INDEX IF NOT EXISTS idx_urls_user_id_created

    ON urls(user_id, created_at DESC);

COMMIT;
```

3.2 PostgreSQL Schema — `migrations/002_create_outbox.sql`

```sql
-- migrations/002_create_outbox.sql

BEGIN;

CREATE TABLE IF NOT EXISTS outbox (

    id           UUID         PRIMARY KEY DEFAULT gen_random_uuid(),

    event_type   TEXT         NOT NULL,       -- routing key, e.g. "url.clicked"

    payload      JSONB        NOT NULL,       -- serialized event struct

    created_at   TIMESTAMPTZ  NOT NULL DEFAULT now(),

    published_at TIMESTAMPTZ  NULL            -- NULL = not yet published

);

-- Poller query: unpublished rows, oldest first, bounded.

-- Partial index (WHERE published_at IS NULL) keeps the index small

-- as published rows accumulate.

CREATE INDEX IF NOT EXISTS idx_outbox_unpublished

    ON outbox(created_at ASC)

    WHERE published_at IS NULL;

COMMIT;
```

Field rationale:

- `outbox.payload JSONB`: The full event struct is serialized here. Workers read this column and publish it directly to RabbitMQ as the message body. No JOIN to `urls` is needed at publish time.
- `outbox.published_at NULL`: The poller filters on `WHERE published_at IS NULL`. Marking a row published sets this to `now()`. Rows are never deleted — they form an audit trail.
- `urls.is_active BOOLEAN NOT NULL DEFAULT true`: Soft delete. Hard deletes would leave broken cache entries and orphaned analytics rows. `is_active=false` + Redis DEL is the correct sequence.
- `urls.expires_at TIMESTAMPTZ NULL`: Nullable means "no expiry." A sentinel value (e.g., year 9999) would work but is less idiomatic in PostgreSQL.

3.3 Go Structs

```go
// repository/url_repository.go

package repository

import "time"

// URL is the domain entity stored in the urls table.

type URL struct {

    ID          string     // UUID v4, primary key

    ShortCode   string     // 7-char base62; up to 10 chars for custom codes

    OriginalURL string     // full destination URL, validated before storage

    UserID      string     // UUID string; JWT sub claim of the creator

    CreatedAt   time.Time  // set by DB DEFAULT

    ExpiresAt   *time.Time // nil = no expiry

    IsActive    bool       // false = soft-deleted

}

// URLSummary is the projection returned by ListByUser.

// It omits internal fields (ID is included as cursor value only).

type URLSummary struct {

    ID          string     // UUID; used as cursor value, not exposed in API response

    ShortCode   string

    OriginalURL string

    CreatedAt   time.Time

    ExpiresAt   *time.Time

    IsActive    bool

}

// Sentinel errors returned by URLRepository.

var (

    ErrURLNotFound    = errors.New("url not found")

    ErrCodeConflict   = errors.New("short code already exists")

    ErrNotOwner       = errors.New("caller does not own this url")

)
```

```go
// repository/outbox_repository.go

package repository

// OutboxEntry is a row in the outbox table.

type OutboxEntry struct {

    ID          string    // UUID; primary key

    EventType   string    // RabbitMQ routing key

    Payload     []byte    // JSON-encoded event struct

    CreatedAt   time.Time

    PublishedAt *time.Time // nil = not yet published

}
```

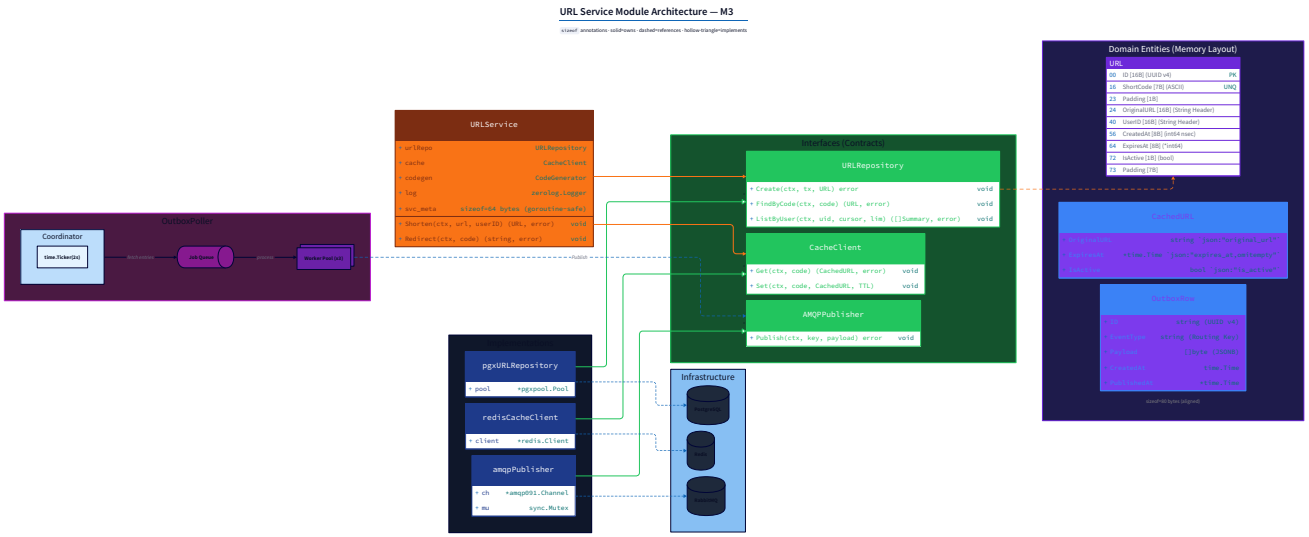
GO

```go
// cache/cache.go

package cache

import "time"

// CachedURL is the value stored in Redis for a given short_code key.

// It contains exactly enough information to serve a redirect without a DB query.

// The outbox event is NOT cached — it always goes to PostgreSQL.

type CachedURL struct {

    OriginalURL string     `json:"original_url"`

    ExpiresAt   *time.Time `json:"expires_at,omitempty"`

    IsActive    bool       `json:"is_active"`

}

// RedisKey for a short code: "url:{short_code}"

// Example: "url:aB3xY9z"

func RedisKey(shortCode string) string {

    return "url:" + shortCode

}
```

GO

GO

```go
// handler/shorten.go — Request/Response DTOs

package handler

import "time"

// ShortenRequest is the JSON body accepted by POST /shorten.

type ShortenRequest struct {

    URL        string  `json:"url"`                 // required; must have scheme + host

    CustomCode string  `json:"custom_code,omitempty"` // optional; 4-10 alphanumeric chars

    ExpiresAt  *string `json:"expires_at,omitempty"`  // optional; RFC3339 string

}

// ShortenResponse is the JSON body returned on 201 Created.

type ShortenResponse struct {

    ShortCode   string  `json:"short_code"`

    ShortURL    string  `json:"short_url"`    // e.g. "http://localhost:8081/aB3xY9z"

    OriginalURL string  `json:"original_url"`

    ExpiresAt   *string `json:"expires_at,omitempty"` // RFC3339 or absent

}

// URLListResponse is the JSON body returned by GET /urls.

type URLListResponse struct {

    URLs       []URLItem `json:"urls"`

    NextCursor *string   `json:"next_cursor,omitempty"` // UUID string; absent if no more pages

}

// URLItem is a single URL entry in the list response.

type URLItem struct {

    ShortCode   string  `json:"short_code"`

    OriginalURL string  `json:"original_url"`

    CreatedAt   string  `json:"created_at"`            // RFC3339

    ExpiresAt   *string `json:"expires_at,omitempty"`

    IsActive    bool    `json:"is_active"`

}
```


URL Service Module Architecture — M3
stencil annotations · solid=owns · dashed=references · hollow-triangle=implements
Domain Entities (Memory Layout)
URL
00 ID [16B] (UUID v4) PK
16 ShortCode [7B] (ASCII) UNQ
23 _Padding [1B]
24 OriginalURL [16B] (String Header)
40 UserID [16B] (String Header)
56 CreatedAt [8B] (int64 nsec)
64 ExpiresAt [8B] (*int64)
72 IsActive [1B] (bool)
73 _Padding [7B]
CachedURL
+ OriginalURL string `json:"original_url"`
+ ExpiresAt *time.Time `json:"expires_at,omitempty"`
+ IsActive bool `json:"is_active"`
OutboxEvent
+ ID string (UUID v4)
+ EventType string (Routing Key)
+ Payload []byte (JSON)
+ CreatedAt time.Time
+ _pad [1]time.Time
sizeof=80 bytes (aligned)
URLService
+ urlRepo URLRepository
+ cache CacheClient
+ codegen CodeGenerator
+ log zerolog.Logger
+ svc_meta sizeof=64 bytes (goroutine-safe)
+ Shorten(ctx, url, userID) (URL, error) void
+ Redirect(ctx, code) (string, error) void
Interfaces (Contracts)
URLRepository
+ Create(ctx, tx, URL) error void
+ FindByCode(ctx, code) (URL, error) void
+ ListByUser(ctx, uid, cursor, lim) ([]Summary, error) void
CacheClient
+ Get(ctx, code) (CachedURL, error) void
+ Set(ctx, code, CachedURL, TTL) void
AMQPPublisher
+ Publish(ctx, key, payload) error void
OutboxPoller
Coordinator
time.Ticker[1s]
fetch entries
Job Queue
process
Worker Pool [x3]
publish
pgxURLRepository
+ pool *pgxpool.Pool
redisCacheClient
+ client *redis.Client
amqpPublisher
+ ch *amqp091.Channel
+ mu sync.Mutex
Infrastructure
PostgreSQL
Redis
RabbitMQ

4. Interface Contracts

4.1 `URLRepository`

```go
// repository/url_repository.go

// URLRepository defines persistence operations for URL entities.

// All implementations must be safe for concurrent use.

// Implementations must NOT start or commit transactions; callers provide pgx.Tx

// for operations that require atomicity.

type URLRepository interface {

    // Create inserts a new URL row. Executes within the provided transaction tx.

    // tx must be non-nil — Create is always called inside a transaction that also

    // calls OutboxRepository.Insert. If short_code violates the unique constraint,

    // returns ErrCodeConflict (pgconn code "23505"). Other DB errors are wrapped

    // with fmt.Errorf("urlRepository.Create: %w", err).

    Create(ctx context.Context, tx pgx.Tx, u URL) error

    // FindByCode retrieves a URL by short_code using the pool (not a transaction).

    // Returns ErrURLNotFound if no row matches.

    // The returned URL contains all fields including is_active and expires_at.

    FindByCode(ctx context.Context, shortCode string) (URL, error)

    // ListByUser returns a page of URLs owned by userID.

    // afterID is the cursor: if non-empty, returns rows with created_at strictly

    // less than the row identified by afterID. Rows are ordered by (created_at DESC, id DESC).

    // limit is the maximum number of rows to return (caller passes pageSize + 1 to detect

    // whether a next page exists; caller slices the result before returning to the API client).

    // Returns an empty slice (not nil) if no rows match.

    ListByUser(ctx context.Context, userID string, afterID string, limit int) ([]URLSummary, error)

    // SoftDelete sets is_active=false for the row identified by shortCode.

    // Returns ErrURLNotFound if no row exists.

    // Returns ErrNotOwner if the row's user_id does not match ownerID.

    // Executes within the provided transaction tx.

    SoftDelete(ctx context.Context, tx pgx.Tx, shortCode, ownerID string) error

}
```

SQL queries for each method:

```sql
-- Create
INSERT INTO urls (id, short_code, original_url, user_id, expires_at)
VALUES ($1, $2, $3, $4, $5)

-- FindByCode
SELECT id, short_code, original_url, user_id, created_at, expires_at, is_active
FROM urls
WHERE short_code = $1

-- ListByUser (with cursor)
SELECT id, short_code, original_url, created_at, expires_at, is_active
FROM urls
WHERE user_id = $1
  AND ($2::uuid IS NULL OR (created_at, id) < (
      SELECT created_at, id FROM urls WHERE id = $2::uuid
  ))
ORDER BY created_at DESC, id DESC
LIMIT $3

-- SoftDelete (with ownership check — single atomic UPDATE)
UPDATE urls
SET is_active = false
WHERE short_code = $1
  AND user_id = $2
RETURNING id

-- If 0 rows affected: determine whether code exists to distinguish 404 vs 403:
--   SELECT id, user_id FROM urls WHERE short_code = $1
--   No row → ErrURLNotFound; row with wrong user_id → ErrNotOwner
```

Note on `ListByUser` cursor: Using `(created_at, id) < (SELECT ...)` is a keyset/cursor approach. The subquery fetches the anchor row's `(created_at, id)` so the comparison is correct even if multiple rows share the same `created_at`. The index `idx_urls_user_id_created` covers `(user_id, created_at DESC)` which satisfies the `WHERE user_id = $1` filter and `ORDER BY created_at DESC`.

4.2 `OutboxRepository`

```go
// repository/outbox_repository.go

// OutboxRepository writes event rows to the outbox table and marks them published.
type OutboxRepository interface {
    // Insert writes a new outbox row within the provided transaction tx.
    // tx must be the same transaction as the URL write — atomicity is the contract.
    // eventType is the RabbitMQ routing key (e.g., "url.clicked").
    // payload is the JSON-serialized event struct.
    Insert(ctx context.Context, tx pgx.Tx, eventType string, payload []byte) error
    // FetchUnpublished returns up to limit unpublished outbox rows, ordered by
    // created_at ASC (oldest first). Uses the pool (no transaction) because the
    // poller reads and marks-published in two separate steps.
    // Returns an empty slice (not nil) if no rows are pending.
    FetchUnpublished(ctx context.Context, limit int) ([]OutboxEntry, error)
    // MarkPublished sets published_at = now() for the row identified by id.
    // Uses the pool (no transaction). Idempotent: if already published, no-op.
    MarkPublished(ctx context.Context, id string) error
}
```

SQL queries:

```sql
-- Insert (within caller's transaction)
INSERT INTO outbox (id, event_type, payload, created_at)
VALUES (gen_random_uuid(), $1, $2, now())
-- FetchUnpublished
SELECT id, event_type, payload, created_at
FROM outbox
WHERE published_at IS NULL
ORDER BY created_at ASC
LIMIT $1
-- MarkPublished
UPDATE outbox
SET published_at = now()
WHERE id = $1
```

4.3 `CacheClient`

```go
// cache/cache.go

// CacheClient wraps Redis operations for URL caching.

// All methods must be nil-safe: if the underlying Redis client is nil

// (set during startup when Redis was unavailable), every method returns

// a cache-miss sentinel or no-op without panicking.

// All errors are logged internally; callers never receive Redis errors —

// cache failures are transparent.

type CacheClient interface {

    // Get retrieves a CachedURL by short_code key ("url:{code}").

    // Returns (entry, nil) on hit.

    // Returns (CachedURL{}, ErrCacheMiss) on miss or any Redis error.

    // Never returns any error other than ErrCacheMiss to the caller.

    Get(ctx context.Context, shortCode string) (CachedURL, error)

    // Set stores a CachedURL with the given TTL.

    // ttl must be > 0; if ttl <= 0, Set is a no-op (never cache with no expiry).

    // Any Redis error is logged and silently swallowed.

    Set(ctx context.Context, shortCode string, entry CachedURL, ttl time.Duration)

    // Del removes the cache entry for shortCode.

    // Any Redis error is logged and silently swallowed.

    // This is a best-effort operation; callers must not assume the key is gone.

    Del(ctx context.Context, shortCode string)

}

// ErrCacheMiss is the only error CacheClient.Get returns to callers.

var ErrCacheMiss = errors.New("cache miss")
```

Implementation detail — nil safety:

```go
type redisCacheClient struct {
    client *redis.Client // may be nil if Redis was unavailable at startup
    log    zerolog.Logger
}
func (c *redisCacheClient) Get(ctx context.Context, shortCode string) (CachedURL, error) {
    if c.client == nil {
        return CachedURL{}, ErrCacheMiss
    }
    val, err := c.client.Get(ctx, RedisKey(shortCode)).Result()
    if errors.Is(err, redis.Nil) {
        return CachedURL{}, ErrCacheMiss
    }
    if err != nil {
        c.log.Warn().Err(err).Str("key", RedisKey(shortCode)).Msg("redis GET failed; cache miss")
        return CachedURL{}, ErrCacheMiss
    }
    var entry CachedURL
    if err = json.Unmarshal([]byte(val), &entry); err != nil {
        c.log.Warn().Err(err).Msg("redis value unmarshal failed; cache miss")
        return CachedURL{}, ErrCacheMiss
    }
    return entry, nil
}
```

4.4 `CodeGenerator`

```go
// codegen/codegen.go
// CodeGenerator generates short codes.
type CodeGenerator interface {
    // Generate produces a 7-character base62 string from crypto/rand.
    // Returns an error only if the OS random source fails (extremely rare).
    // Collision detection is NOT performed here — that is the caller's
    // responsibility (URLService.createShortCode retry loop).
    Generate() (string, error)
}
// NewCodeGenerator returns the default 7-character base62 generator.
func NewCodeGenerator() CodeGenerator
```

4.5 `AMQPPublisher`

```go
// amqp/publisher.go

// AMQPPublisher publishes messages to the RabbitMQ topic exchange.
type AMQPPublisher interface {
    // Publish sends body to the exchange "url-shortener" with the given routingKey.
    // body must be a JSON-encoded event struct.
    // Returns an error if the AMQP channel is closed or the broker is unreachable.
    // Callers (outbox workers) handle errors by logging and leaving the outbox row
    // unpublished for the next poll cycle.
    Publish(ctx context.Context, routingKey string, body []byte) error
}
```

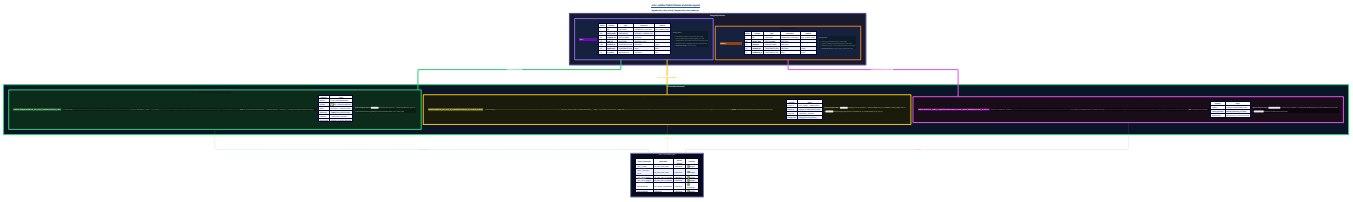
4.6 `URLService`

```go
// service/url_service.go

// URLService orchestrates URL creation, lookup, listing, deletion, and event emission.

// It owns no state beyond its dependencies. Safe for concurrent use.

type URLService struct {

    pool      *pgxpool.Pool

    urlRepo   repository.URLRepository

    outboxRepo repository.OutboxRepository

    cache     cache.CacheClient

    codegen   codegen.CodeGenerator

    baseURL   string // e.g. "http://localhost:8081" — prepended to short_code for short_url

    log       zerolog.Logger

}

func NewURLService(

    pool *pgxpool.Pool,

    urlRepo repository.URLRepository,

    outboxRepo repository.OutboxRepository,

    cache cache.CacheClient,

    codegen codegen.CodeGenerator,

    baseURL string,

    log zerolog.Logger,

) *URLService

// Shorten creates a new short URL and writes a URLCreatedEvent to the outbox

// in the same transaction. Returns the created URL entity.

// Errors: ErrURLInvalid, ErrCodeConflict (exhausted retries → 503), DB errors.

func (s *URLService) Shorten(

    ctx context.Context,

    originalURL string,

    customCode string,       // empty string = auto-generate

    expiresAt *time.Time,

    userID string,

    userEmail string,        // from JWT claims; embedded in URLCreatedEvent

    correlationID string,

) (repository.URL, error)

// Redirect looks up a URL by short_code using the read-through cache.

// Writes a URLClickedEvent to the outbox within a DB transaction on cache hit or miss.

// Returns the original URL for redirect.

// Errors: ErrURLNotFound, ErrURLExpired, ErrURLInactive.

func (s *URLService) Redirect(

    ctx context.Context,
```

```
    shortCode string,
    ipHash string,
    userAgent string,
    referer string,
    correlationID string,
) (originalURL string, err error)
// ListURLs returns a cursor-paginated list of URLs owned by userID.
// pageSize is capped at 100 internally regardless of caller input.
// afterCursor is the UUID of the last seen URL (empty = first page).
func (s *URLService) ListURLs(
    ctx context.Context,
    userID string,
    afterCursor string,
    pageSize int,
) (urls []repository.URLSummary, nextCursor *string, err error)
// Delete soft-deletes a URL and publishes URLDeletedEvent via outbox.
// Invalidates the Redis cache entry as a best-effort operation.
// Errors: ErrURLNotFound, ErrNotOwner, DB errors.
func (s *URLService) Delete(
    ctx context.Context,
    shortCode string,
    userID string,
    userEmail string,
    correlationID string,
) error
```

Sentinel errors for URLService:

```go
var (
    ErrURLInvalid  = errors.New("url is invalid: must have scheme and host")
    ErrURLExpired  = errors.New("url has expired")
    ErrURLInactive = errors.New("url is inactive")
    ErrExhausted   = errors.New("short code generation exhausted: 5 collisions")
)
```



5. Algorithm Specification

5.1 Base62 Code Generation

```go
// codegen/codegen.go

// base62Alphabet is the character set for short codes.

// Chosen for URL-safety and case-sensitivity.

// Length = 62. Index → character mapping is stable and must not change.

const base62Alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

// codeLength is the fixed length of auto-generated short codes.

const codeLength = 7

// Generate produces a 7-character base62 string.

// Algorithm:

//   1. Allocate a byte slice of length 7.

//   2. For each position i in [0, 7):

//      a. Read 1 random byte from crypto/rand.Reader.

//      b. Map the byte to an alphabet index: idx = byte % 62.

//         NOTE: This introduces a tiny modulo bias (256 % 62 = 8, so indices

//         0–7 are ~0.4% more likely). This bias is acceptable for URL shortening;

//         it does not compromise security. For uniform distribution, use

//         rejection sampling (discard bytes >= 248 and re-read), but this

//         complexity is not warranted here.

//      c. code[i] = base62Alphabet[idx]

//   3. Return string(code), nil.

//   4. If crypto/rand.Reader.Read fails, return "", fmt.Errorf("codegen.Generate: %w", err).
```

Implementation:

```go
func (g *base62Generator) Generate() (string, error) {

    buf := make([]byte, codeLength)

    code := make([]byte, codeLength)

    if _, err := io.ReadFull(rand.Reader, buf); err != nil {

        return "", fmt.Errorf("codegen.Generate: %w", err)

    }

    for i, b := range buf {

        code[i] = base62Alphabet[int(b)%62]

    }

    return string(code), nil

}
```

5.2 Short Code Selection with Collision Retry

```
PROCEDURE URLService.createShortCode(ctx, customCode string):
  IF customCode != "":
    VALIDATE customCode: len ∈ [4, 10], all chars in base62Alphabet
    IF invalid → return "", ErrURLInvalid (caller maps to 400)
    return customCode, nil  -- collision will be detected by DB unique constraint
  FOR attempt := 1; attempt <= 5; attempt++:
    code, err = s.codegen.Generate()
    IF err → return "", fmt.Errorf("createShortCode: %w", err) (→ 500)
    -- Optimistically attempt INSERT; let the DB unique constraint catch collisions.
    -- We do NOT pre-check existence with a SELECT — that is a TOCTOU race.
    -- The caller (Shorten) handles ErrCodeConflict by retrying this procedure.
    return code, nil  -- caller retries the full transaction on ErrCodeConflict
  -- This path is reached only if the caller has retried 5 times and all failed.
  return "", ErrExhausted  -- (→ 503)
END PROCEDURE
```

Collision retry in `URLService.Shorten`:

```go
const maxCollisionRetries = 5

func (s *URLService) Shorten(ctx context.Context, ...) (repository.URL, error) {

    // ... validation ...

    for attempt := 1; attempt <= maxCollisionRetries; attempt++ {

        code, err := s.selectCode(ctx, customCode)

        if err != nil {

            return repository.URL{}, err

        }

        url, err := s.createURLWithOutbox(ctx, code, ...)

        if errors.Is(err, repository.ErrCodeConflict) {

            if customCode != "" {

                // Custom code conflict is not retryable — it's a user error.

                return repository.URL{}, fmt.Errorf("%w: custom code already taken", repository.ErrCodeConflict)

            }

            s.log.Debug().Int("attempt", attempt).Str("code", code).Msg("short code collision; retrying")

            continue

        }

        if err != nil {

            return repository.URL{}, err

        }

        return url, nil

    }

    return repository.URL{}, ErrExhausted

}
```

5.3 Atomic URL + Outbox Write

```
PROCEDURE URLService.createURLWithOutbox(ctx, url URL, event URLCreatedEvent):
  1. tx, err = pool.Begin(ctx)
     IF err → return wrapped error (→ 500)
  2. DEFER tx.Rollback(ctx)  -- no-op if Commit succeeds
  3. err = urlRepo.Create(ctx, tx, url)
     IF ErrCodeConflict → return ErrCodeConflict (caller retries)
     IF other error → return wrapped error (rollback via defer)
  4. payload, err = json.Marshal(event)
     IF err → return wrapped error (rollback via defer)
  5. err = outboxRepo.Insert(ctx, tx, event.EventType, payload)
     IF err → return wrapped error (rollback via defer)
  6. err = tx.Commit(ctx)
     IF err → return fmt.Errorf("createURLWithOutbox: commit: %w", err)
  7. return nil
END PROCEDURE
```

Invariant enforced: Steps 3 and 5 execute within the same `pgx.Tx`. A crash between step 6 and the HTTP response causes the transaction to be rolled back by PostgreSQL (connection close triggers rollback). The URL does not exist; the outbox row does not exist; the client receives a connection reset — acceptable for a rare crash scenario.

5.4 Redirect with Read-Through Cache

```
PROCEDURE URLService.Redirect(ctx, shortCode, ipHash, userAgent, referer, correlationID):
  1. entry, err = cache.Get(ctx, shortCode)
     IF err == nil (cache HIT):
       IF !entry.IsActive → return "", ErrURLInactive
       IF entry.ExpiresAt != nil && entry.ExpiresAt.Before(time.Now()) → return "", ErrURLExpired
       -- Record click event via outbox (must still go to DB)
       err = s.recordClick(ctx, shortCode, ipHash, userAgent, referer, correlationID)
       IF err → log.Warn (do NOT fail the redirect); continue
       return entry.OriginalURL, nil
  2. -- Cache miss or Redis error; fall through to DB
     url, err = urlRepo.FindByCode(ctx, shortCode)
     IF ErrURLNotFound → return "", ErrURLNotFound
     IF other error → return "", wrapped error (→ 500)
  3. IF !url.IsActive → return "", ErrURLInactive
     IF url.ExpiresAt != nil && url.ExpiresAt.Before(time.Now()) → return "", ErrURLExpired
  4. -- Populate cache BEFORE recording click (faster response to next caller)
     ttl = computeTTL(url.ExpiresAt)  -- see § 5.5
     cache.Set(ctx, shortCode, CachedURL{
         OriginalURL: url.OriginalURL,
         ExpiresAt:   url.ExpiresAt,
         IsActive:    url.IsActive,
     }, ttl)
  5. -- Record click event via outbox
     err = s.recordClick(ctx, shortCode, ipHash, userAgent, referer, correlationID)
     IF err → log.Warn (do NOT fail the redirect)
  6. return url.OriginalURL, nil
END PROCEDURE
```

`recordClick` — atomic click + outbox:

```go
func (s *URLService) recordClick(ctx context.Context, shortCode, ipHash, userAgent, referer, correlationID string) error {
    event := events.URLClickedEvent{
        EventType:     events.EventTypeURLClicked,
        OccurredAt:    time.Now().UTC(),
        CorrelationID: correlationID,
        EventID:       uuid.New().String(),
        ShortCode:     shortCode,
        IPHash:        ipHash,
        UserAgent:     userAgent,
        Referer:       referer,
    }
    payload, err := json.Marshal(event)
    if err != nil {
        return fmt.Errorf("recordClick marshal: %w", err)
    }
    tx, err := s.pool.Begin(ctx)
    if err != nil {
        return fmt.Errorf("recordClick begin tx: %w", err)
    }
    defer tx.Rollback(ctx)
    if err = s.outboxRepo.Insert(ctx, tx, event.EventType, payload); err != nil {
        return fmt.Errorf("recordClick insert outbox: %w", err)
    }
    return tx.Commit(ctx)
}
```

Note on click atomicity: The click outbox row is written in its own transaction separate from the redirect response. If the service crashes after sending the HTTP 301 but before committing this transaction, the click is lost. This is an acknowledged at-least-once tradeoff: the redirect always succeeds; the click event may occasionally be lost. To achieve true at-least-once for clicks, one would buffer the click in a pre-response transaction that also locks the URL row — too expensive for the redirect hot path. The current design is correct for this system's requirements.

5.5 Cache TTL Computation

```
FUNCTION computeTTL(expiresAt *time.Time) time.Duration:
  maxTTL = 1 * time.Hour
  IF expiresAt == nil:
    return maxTTL
  remaining = time.Until(*expiresAt)
  IF remaining <= 0:
    -- URL is already expired; do not cache it.
    -- Caller should have returned ErrURLExpired before reaching Set.
    -- Defensive: return a very short TTL so stale data doesn't linger.
    return 1 * time.Second
  IF remaining < maxTTL:
    return remaining
  return maxTTL
```

Invariant: A Redis entry for a URL with `expires_at` set never outlives that expiry time. TTL = `min(expires_at - now(), 1h)`.

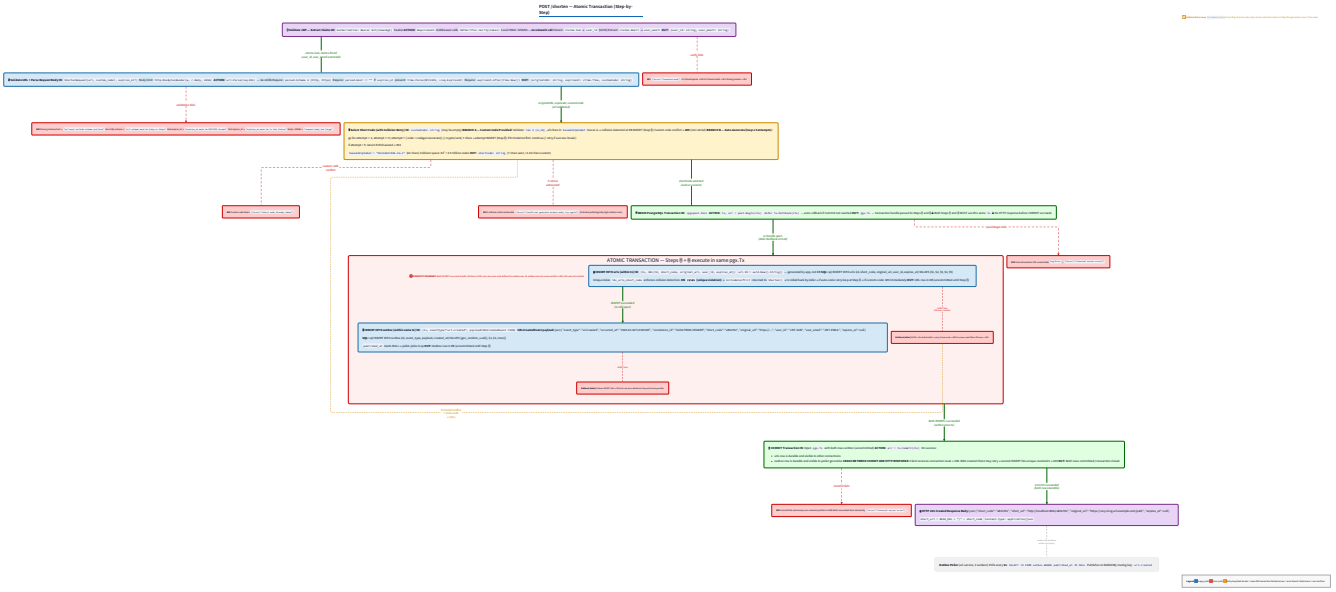
5.6 Cursor-Based Pagination for `GET /urls`

```
PROCEDURE URLService.ListURLs(ctx, userID, afterCursor, pageSize):
  pageSize = clamp(pageSize, 1, 100)  -- cap at 100; default 20
  fetchLimit = pageSize + 1           -- fetch one extra to detect next page
  rows, err = urlRepo.ListByUser(ctx, userID, afterCursor, fetchLimit)
  IF err → return nil, nil, wrapped error
  hasMore = len(rows) > pageSize
  IF hasMore:
    rows = rows[:pageSize]            -- trim the sentinel extra row
  var nextCursor *string
  IF hasMore:
    lastID = rows[len(rows)-1].ID
    nextCursor = &lastID             -- cursor is the UUID of the last returned row
  return rows, nextCursor, nil
END PROCEDURE
```

Why UUID cursor, not `created_at` cursor: `created_at` values can collide (two URLs created in the same microsecond). Using the UUID `id` as the keyset cursor — with the SQL subquery fetching that row's `(created_at, id)` — is both collision-safe and avoids transmitting timestamps in the cursor parameter.

5.7 Outbox Worker Pool

```
PROCEDURE StartOutboxPoller(ctx, pool, outboxRepo, publisher, log):
  jobChan = make(chan OutboxEntry, 50)  -- buffered; coordinator fills, workers drain
  -- Start 3 worker goroutines
  FOR i := 0; i < 3; i++:
    go outboxWorker(ctx, jobChan, outboxRepo, publisher, log)
  -- Coordinator goroutine (this goroutine IS the coordinator)
  FOR:
    SELECT:
    CASE <-ctx.Done():
      close(jobChan)
      return
    CASE <-time.After(2 * time.Second):
      entries, err = outboxRepo.FetchUnpublished(ctx, 50)
      IF err:
        log.Warn().Err(err).Msg("outbox fetch failed")
        continue
      FOR each entry in entries:
        SELECT:
        CASE jobChan <- entry:   -- send to worker
        CASE <-ctx.Done():
          close(jobChan)
          return
END PROCEDURE
PROCEDURE outboxWorker(ctx, jobChan, outboxRepo, publisher, log):
  FOR entry := range jobChan:
    err = publisher.Publish(ctx, entry.EventType, entry.Payload)
    IF err:
      log.Warn().Err(err).Str("id", entry.ID).Msg("outbox publish failed; will retry next cycle")
      -- Do NOT mark published. The coordinator will re-fetch this row next cycle.
      continue
    err = outboxRepo.MarkPublished(ctx, entry.ID)
    IF err:
      log.Error().Err(err).Str("id", entry.ID).Msg("outbox mark published failed; event may be re-published")
      -- Event was published to RabbitMQ but the DB update failed.
      -- On next poll, this row will be re-fetched and re-published.
      -- This is the at-least-once guarantee: consumers must be idempotent (M4).
END PROCEDURE
```



Channel capacity rationale: A buffered channel of 50 matches `FetchUnpublished(ctx, 50)`. If 3 workers are slower than the coordinator (e.g., RabbitMQ is slow), the channel fills and the coordinator blocks on the `jobChan <- entry` send. This provides natural backpressure: the coordinator does not enqueue faster than workers can drain.

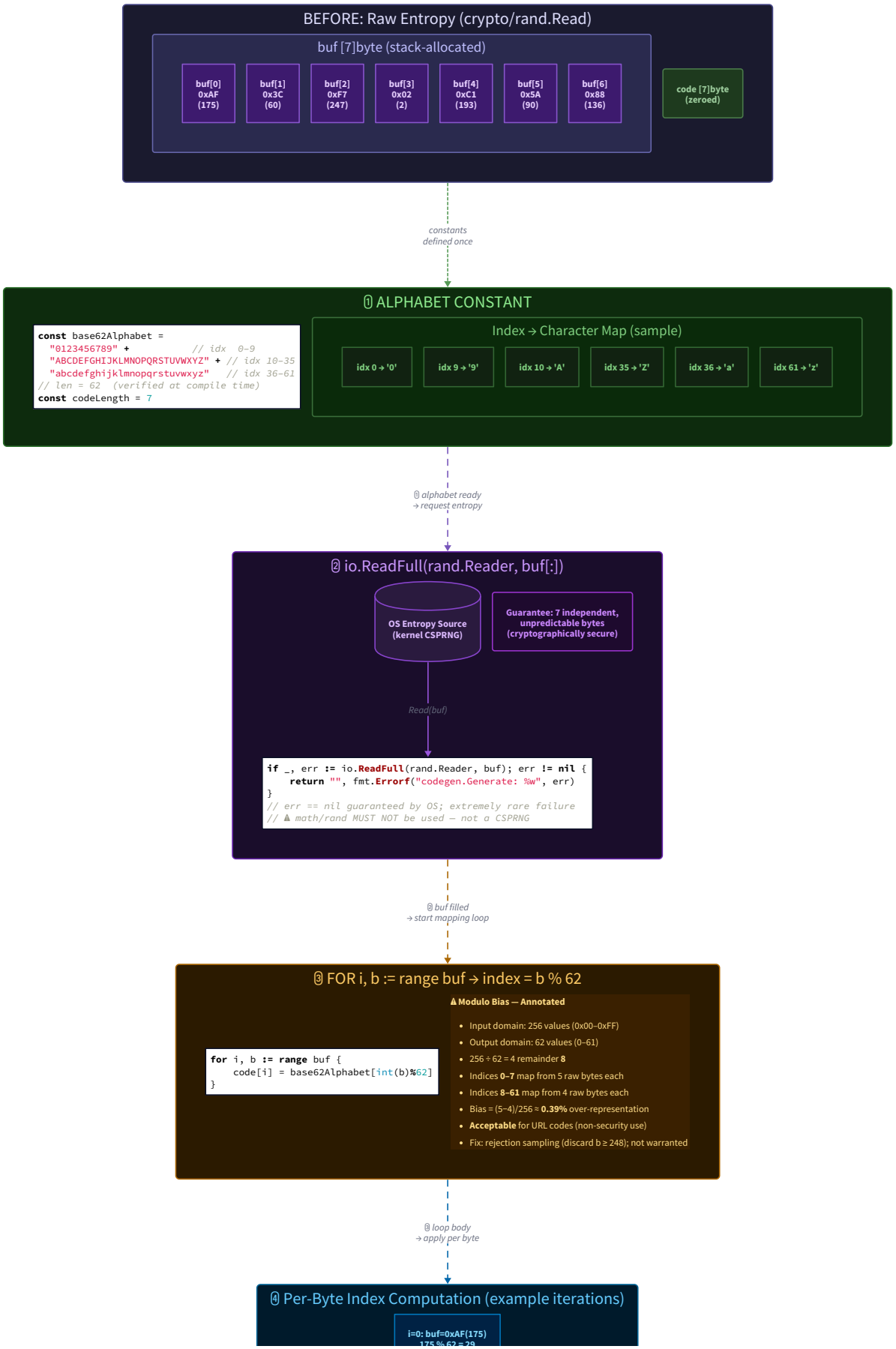
5.8 AMQPPublisher Implementation

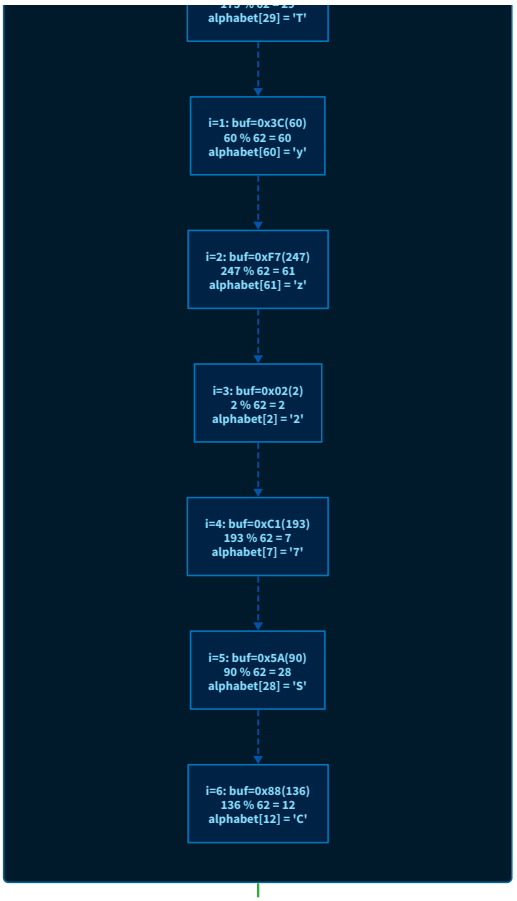
```go
// amqp/publisher.go

type amqpPublisher struct {
    ch  *amqp091.Channel
    mu  sync.Mutex // protects ch; channel is not goroutine-safe per amqp091 docs
    log zerolog.Logger
}

func (p *amqpPublisher) Publish(ctx context.Context, routingKey string, body []byte) error {
    p.mu.Lock()
    defer p.mu.Unlock()
    return p.ch.PublishWithContext(
        ctx,
        "url-shortener", // exchange
        routingKey,
        false,           // mandatory: false — drop if no queue bound (don't block)
        false,           // immediate: false — deprecated in RabbitMQ 3+
        amqp091.Publishing{
            ContentType:  "application/json",
            DeliveryMode: amqp091.Persistent, // survive broker restart
            Body:         body,
        },
    )
}
```

`amqp091.Channel` thread safety: The `amqp091` library's `Channel` is not safe for concurrent use. The `sync.Mutex` in `amqpPublisher` serializes concurrent `Publish` calls from the 3 worker goroutines. This is correct and intentional.

Base62 Code Generation Algorithm

`codegen.Generate()` — `crypto/rand` source, 7-char output





@ all 7 bytes mapped → result

AFTER: 7-char Base62 String

"Tyz27SC"
len = 7 ✓
all chars ∈ base62Alphabet ✓
crypto/rand source ✓

satisfies

Post-conditions verified by TestGenerate_*

- `len(code) == 7` always
- every `code[i] ∈ base62Alphabet`
- 1,000 consecutive calls → 0 duplicates
- `import "math/rand"` absent from source
- `crypto/rand.Reader` is the sole entropy source

returned to URLService.Shorten for DB insert

5 Collision Retry (caller: URLService.Shorten)

```go
for attempt := 1; attempt <= 5; attempt++ {
    code, _ := s.codegen.Generate()          // ← steps 0–6
    err := urlRepo.Create(ctx, tx, url{ShortCode: code, ...})
    if errors.Is(err, ErrCodeConflict) {
        log.Debug().Int("attempt", attempt).Msg("collision; retrying")
        continue                             // 62^7 = 3.52T codes; P(collision) ≈ 0
    }
    return code, nil                         // success
}
return "", ErrExhausted                      // 5 collisions: 503
```

P(collision | N existing codes) = N / 62^7
At N=1M: ≈ 0.000028%
Earth's storage saturates first

Custom code path:
Skip Generate()
Pass user string directly
DB UNIQUE → ErrCodeConflict → 409

Transformation Summary

Position	Raw	Decimal	%	Index	Char

Position	byte	Decimal	`62`	Index	Char
`code[0]`	`0xAF`	175	→	29	**'T'**
`code[1]`	`0x3C`	60	→	60	**'y'**
`code[2]`	`0xF7`	247	→	61	**'z'**
`code[3]`	`0x02`	2	→	2	**'2'**
`code[4]`	`0xC1`	193	→	7	**'7'**
`code[5]`	`0x5A`	90	→	28	**'S'**
`code[6]`	`0x88`	136	→	12	**'C'**
Result: `"Tyz27SC"` (7 chars, base62-safe)					

---

6. HTTP Handler Specification

6.1 `POST /shorten` Handler

```
PROCEDURE HandleShorten(w, r):
  -- Middleware: RequireAuth runs before this handler; claims are in context.
  claims, ok = middleware.ClaimsFromContext(r.Context())
  IF !ok → 401 (defensive; middleware should prevent this)
  -- Parse body
  http.MaxBytesReader(w, r.Body, 4096)
  Decode JSON into ShortenRequest
  IF decode error → 400 {"error":"invalid request body"}
  -- Validate original URL
  IF req.URL == "" → 400 {"error":"url is required"}
  parsed, err = url.Parse(req.URL)
  IF err != nil OR parsed.Scheme == "" OR parsed.Host == "" → 400 {"error":"url must include scheme and host"}
  IF parsed.Scheme not in ["http","https"] → 400 {"error":"url scheme must be http or https"}
  -- Parse optional expires_at
  var expiresAt *time.Time
  IF req.ExpiresAt != nil:
    t, err = time.Parse(time.RFC3339, *req.ExpiresAt)
    IF err → 400 {"error":"expires_at must be RFC3339 format"}
    IF t.Before(time.Now()) → 400 {"error":"expires_at must be in the future"}
    expiresAt = &t
  -- Extract correlation ID (set by gateway in M5; fallback for direct calls)
  correlationID = r.Header.Get("X-Correlation-ID")
  IF correlationID == "" → correlationID = uuid.New().String()
  -- Call service
  created, err = urlService.Shorten(
      r.Context(), req.URL, req.CustomCode, expiresAt,
      claims.Sub, claims.Email, correlationID,
  )
  IF errors.Is(err, ErrURLInvalid) → 400 {"error": err.Error()}
  IF errors.Is(err, repository.ErrCodeConflict) → 409 {"error":"short code already taken"}
  IF errors.Is(err, ErrExhausted) → 503 {"error":"could not generate unique code; try again"}
  IF err → 500 {"error":"internal server error"} + log.Error
  -- Build response
  shortURL = baseURL + "/" + created.ShortCode
  var expiresAtStr *string
  IF created.ExpiresAt != nil → expiresAtStr = ptr(created.ExpiresAt.Format(time.RFC3339))
  201 ShortenResponse{ShortCode, ShortURL, OriginalURL, ExpiresAt}
END PROCEDURE
```

6.2 `GET /:code` Redirect Handler

```
PROCEDURE HandleRedirect(w, r):
  shortCode = r.PathValue("code")
  IF shortCode == "" → 400 {"error":"short code is required"}
  -- Extract request metadata for click event
  rawIP = extractClientIP(r)  -- see § 6.2.1
  ipHash = hashIP(rawIP)      -- SHA-256(rawIP + CLICK_SALT env var), hex-encoded
  userAgent = r.Header.Get("User-Agent")
  referer = r.Header.Get("Referer")
  correlationID = r.Header.Get("X-Correlation-ID")
  IF correlationID == "" → correlationID = uuid.New().String()
  originalURL, err = urlService.Redirect(
      r.Context(), shortCode, ipHash, userAgent, referer, correlationID,
  )
  IF errors.Is(err, ErrURLNotFound) → 404 {"error":"not found"}
  IF errors.Is(err, ErrURLExpired) → 410 {"error":"this link has expired"}
  IF errors.Is(err, ErrURLInactive) → 410 {"error":"this link is no longer active"}
  IF err → 500 {"error":"internal server error"} + log.Error
  http.Redirect(w, r, originalURL, http.StatusMovedPermanently)  -- 301
END PROCEDURE
```

§ 6.2.1 Client IP Extraction:

GO

```go
func extractClientIP(r *http.Request) string {

    // Trust X-Forwarded-For only if set by a trusted proxy (the gateway).

    // In local dev, no proxy; use RemoteAddr.

    if xff := r.Header.Get("X-Forwarded-For"); xff != "" {

        // X-Forwarded-For can be "client, proxy1, proxy2"; take leftmost.

        parts := strings.Split(xff, ",")

        return strings.TrimSpace(parts[0])

    }

    host, _, err := net.SplitHostPort(r.RemoteAddr)

    if err != nil {

        return r.RemoteAddr // fallback: return as-is

    }

    return host

}

func hashIP(rawIP string) string {

    salt := os.Getenv("CLICK_SALT") // if unset, uses empty salt — document in .env.example

    h := sha256.Sum256([]byte(rawIP + salt))

    return hex.EncodeToString(h[:])

}
```

Why 301 and not 302: `301 Moved Permanently` allows browsers and CDNs to cache the redirect, reducing load on the service. If the URL is later deactivated, the Redis DEL and `is_active=false` handle server-side state, but cached 301s in client browsers will linger. This is an accepted tradeoff for redirect performance. If mutable redirect targets are a concern, use 302 — the spec mandates 301.

6.3 `GET /urls` List Handler

```
PROCEDURE HandleListURLs(w, r):
  claims, ok = middleware.ClaimsFromContext(r.Context())
  IF !ok → 401
  afterCursor = r.URL.Query().Get("after")  -- UUID string or empty
  pageSizeStr = r.URL.Query().Get("limit")
  pageSize = 20  -- default
  IF pageSizeStr != "":
    pageSize, err = strconv.Atoi(pageSizeStr)
    IF err OR pageSize < 1 → 400 {"error":"limit must be a positive integer"}
  urls, nextCursor, err = urlService.ListURLs(r.Context(), claims.Sub, afterCursor, pageSize)
  IF err → 500 + log.Error
  items = make([]URLItem, len(urls))
  FOR i, u := range urls:
    items[i] = URLItem{
        ShortCode:   u.ShortCode,
        OriginalURL: u.OriginalURL,
        CreatedAt:   u.CreatedAt.Format(time.RFC3339),
        IsActive:    u.IsActive,
    }
    IF u.ExpiresAt != nil → items[i].ExpiresAt = ptr(u.ExpiresAt.Format(time.RFC3339))
  var nextCursorStr *string
  IF nextCursor != nil → nextCursorStr = nextCursor
  200 URLListResponse{URLs: items, NextCursor: nextCursorStr}
END PROCEDURE
```

6.4 `DELETE /urls/:code` Handler

```
PROCEDURE HandleDeleteURL(w, r):
  claims, ok = middleware.ClaimsFromContext(r.Context())
  IF !ok → 401
  shortCode = r.PathValue("code")
  IF shortCode == "" → 400
  correlationID = r.Header.Get("X-Correlation-ID")
  IF correlationID == "" → correlationID = uuid.New().String()
  err = urlService.Delete(r.Context(), shortCode, claims.Sub, claims.Email, correlationID)
  IF errors.Is(err, ErrURLNotFound) → 404 {"error":"not found"}
  IF errors.Is(err, ErrNotOwner) → 403 {"error":"forbidden"}
  IF err → 500 + log.Error
  w.WriteHeader(http.StatusNoContent)  -- 204; no body
END PROCEDURE
```

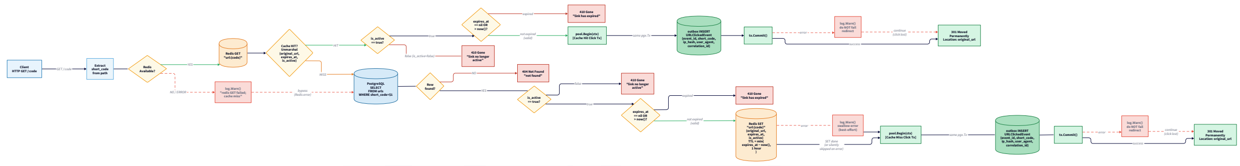
`URLService.Delete` logic:

```
PROCEDURE URLService.Delete(ctx, shortCode, userID, userEmail, correlationID):
  tx, err = pool.Begin(ctx)
  IF err → return wrapped
  DEFER tx.Rollback(ctx)
  err = urlRepo.SoftDelete(ctx, tx, shortCode, userID)
  IF ErrURLNotFound OR ErrNotOwner → return as-is (rollback via defer)
  IF err → return wrapped
  event = URLDeletedEvent{
      EventType:     EventTypeURLDeleted,
      OccurredAt:    time.Now().UTC(),
      CorrelationID: correlationID,
      ShortCode:     shortCode,
      UserID:        userID,
      UserEmail:     userEmail,
  }
  payload, _ = json.Marshal(event)  -- cannot fail for this struct
  err = outboxRepo.Insert(ctx, tx, event.EventType, payload)
  IF err → return wrapped (rollback via defer)
  err = tx.Commit(ctx)
  IF err → return wrapped
  -- Cache invalidation: best-effort, after commit.
  -- Do NOT do this inside the transaction — it is a different store.
  cache.Del(ctx, shortCode)  // error swallowed inside CacheClient.Del
  return nil
END PROCEDURE
```

GET /:code — Read-Through Cache Data Flow

Cache TTL Formula `TTL = min(expires_at - now(), 1h)`

- `expires_at == nil` → TTL = 1h
- `expires_at` within 1h → TTL = remaining
- already expired → skip SET; return 410



Outbox Atomicity Note The `URLClickedEvent` outbox INSERT executes in its own `pool.Begin()` tx, separate from the HTTP response path. A crash after `301` is sent but before `tx.Commit()` loses the click event. This is an acknowledged **at-least-once** tradeoff — the redirect always succeeds. `EventID` (UUID v4) enables consumer-side idempotency deduplication in analytics-service.

7. Error Handling Matrix

Error	Detected By	HTTP Status	Response Body	Logged?	Notes
`URL_INVALID` (missing scheme/host)	`url.Parse` in handler	400	`{"error":"url must include scheme and host"}`	No	Client mistake
`URL_INVALID` (non-http scheme)	handler validation	400	`{"error":"url scheme must be http or https"}`	No	
`EXPIRES_AT_INVALID` (parse fail)	handler	400	`{"error":"expires_at must be RFC3339 format"}`	No	
`EXPIRES_AT_IN_PAST`	handler	400	`{"error":"expires_at must be in the future"}`	No	
`CUSTOM_CODE_CONFLICT`	`urlRepo.Create` → `23505` + custom code set	409	`{"error":"short code already taken"}`	No	Expected
`CODE_EXHAUSTED`	`URLService.Shorten` after 5 retries	503	`{"error":"could not generate unique code; try again"}`	Warn	Indicates high collision rate
`CODE_NOT_FOUND`	`urlRepo.FindByCode` → `ErrURLNotFound`	404	`{"error":"not found"}`	No	Normal
`URL_EXPIRED`	`URLService.Redirect` expiry check	410	`{"error":"this link has expired"}`	No	
`URL_INACTIVE`	`URLService.Redirect` active check	410	`{"error":"this link is no longer active"}`	No	
`OWNERSHIP_VIOLATION`	`urlRepo.SoftDelete` → `ErrNotOwner`	403	`{"error":"forbidden"}`	Warn	Possible attack
`CACHE_ERROR` (Redis GET)	`redisCacheClient.Get`	—	Transparent; returns `ErrCacheMiss`	Warn internally	Never causes 5xx
`CACHE_ERROR` (Redis SET)	`redisCacheClient.Set`	—	Swallowed	Warn internally	
`CACHE_ERROR` (Redis DEL)	`redisCacheClient.Del`	—	Swallowed	Warn internally	Best-effort
`OUTBOX_PUBLISH_FAIL`	`amqpPublisher.Publish`	—	No HTTP effect	Warn in poller	Row stays unpublished; retried
`OUTBOX_MARK_FAIL`	`outboxRepo.MarkPublished`	—	No HTTP effect	Error in worker	Row re-published next cycle; consumers must be idempotent
`DB_ERROR` (unexpected)	any repo call	500	`{"error":"internal server error"}`	**Error**	Never expose details
`JWT_MISSING`	`RequireAuth` middleware	401	`{"error":"unauthorized"}`	No	
`JWT_INVALID`	`RequireAuth` middleware	401	`{"error":"unauthorized"}`	Warn	
`REQUEST_BODY_TOO_LARGE`	`http.MaxBytesReader`	400	`{"error":"request body too large"}`	No	
`AMQP_CHANNEL_CLOSED`	`amqpPublisher.Publish`	—	No HTTP effect	Warn	Poller retries next cycle
`PORT_BIND_FAIL`	`ListenAndServe`	—	Container exits	Fatal	
Consistent error response helper:					

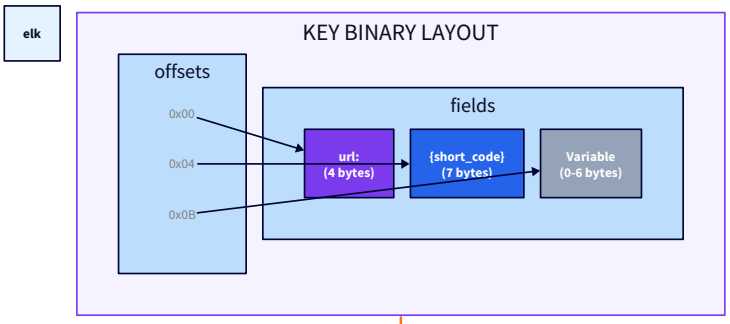
```go
// handler/helpers.go

func writeError(w http.ResponseWriter, status int, msg string) {

    writeJSON(w, status, ErrorResponse{Error: msg})

}

func writeJSON(w http.ResponseWriter, status int, v any) {

    w.Header().Set("Content-Type", "application/json")

    w.WriteHeader(status)

    _ = json.NewEncoder(w).Encode(v)

}
```

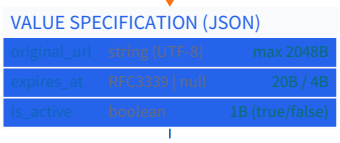
8. Route Registration in `main.go`

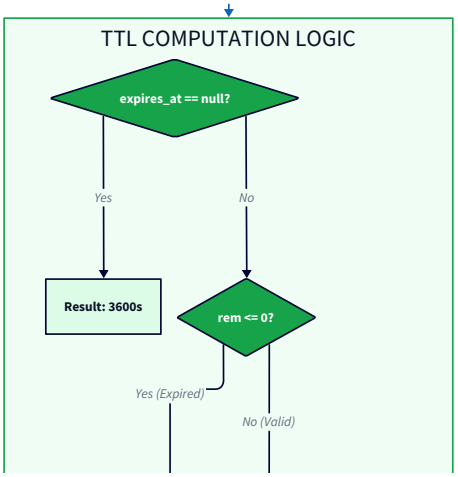
```go
// services/url-service/main.go (route wiring)

verifier := auth.NewJWTVerifier([]byte(mustGetEnv("JWT_SECRET")))

requireAuth := middleware.RequireAuth(verifier, log)

mux := http.NewServeMux()

// Public routes

mux.HandleFunc("GET /health",    handler.NewHealthHandler("url-service", log))

mux.HandleFunc("GET /{code}",    handler.NewRedirectHandler(urlSvc, log))

// Protected routes

mux.Handle("POST /shorten",       requireAuth(http.HandlerFunc(handler.NewShortenHandler(urlSvc, baseURL, log))))

mux.Handle("GET /urls",           requireAuth(http.HandlerFunc(handler.NewListURLsHandler(urlSvc, log))))

mux.Handle("DELETE /urls/{code}", requireAuth(http.HandlerFunc(handler.NewDeleteURLHandler(urlSvc, log))))
```

Pattern ordering note: `GET /{code}` is a wildcard catch-all. In Go 1.22+ `net/http`, more specific patterns take priority. `GET /health` and `GET /urls` are more specific and will match first. `GET /{code}` will only match paths that do not match any more specific pattern. Register more specific patterns before the wildcard.

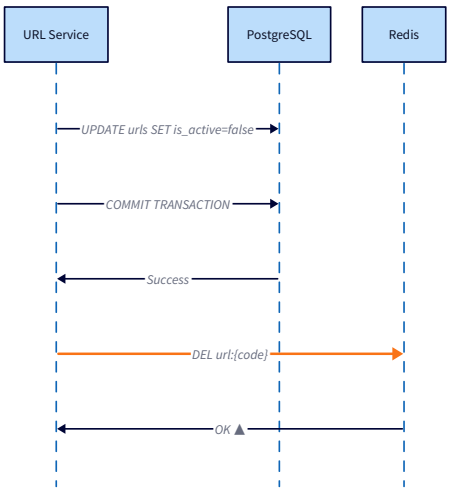
Redis Cache Key Layout — `url:{short_code}`

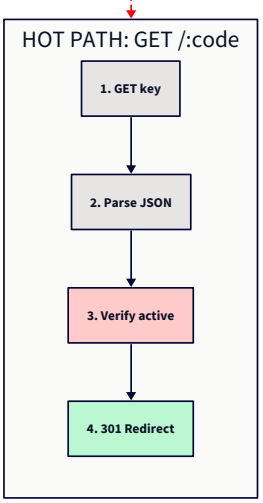


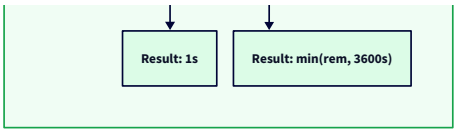




Cache Invalidation Flow







9. Concurrency Specification

9.1 HTTP Handler Goroutines

Each request runs in its own goroutine created by `net/http`. `URLService` is stateless (all state is in the PostgreSQL pool and Redis client, which are goroutine-safe). No shared mutable state exists in handlers or service layer. `pgxpool.Pool` is goroutine-safe per pgx documentation.

9.2 Outbox Worker Pool

Component	Goroutine Count	Shared State	Protection
Coordinator	1	`jobChan`, `outboxRepo`	Channel send is goroutine-safe; `outboxRepo` uses pool
Workers	3	`jobChan`, `outboxRepo`, `publisher`	Channel receive is goroutine-safe; `amqpPublisher` uses `sync.Mutex`
`amqpPublisher`	—	`*amqp091.Channel`	`sync.Mutex` serializes concurrent Publish calls
Goroutine lifecycle: All goroutines receive a `context.Context` derived from `context.WithCancel`. On `SIGTERM`, `main.go` calls the cancel function. The coordinator's `select` on `ctx.Done()` causes it to `close(jobChan)` and return. Workers drain the closed channel (range over closed channel yields remaining items, then exits). This provides a graceful drain: already-queued jobs are published before shutdown.			

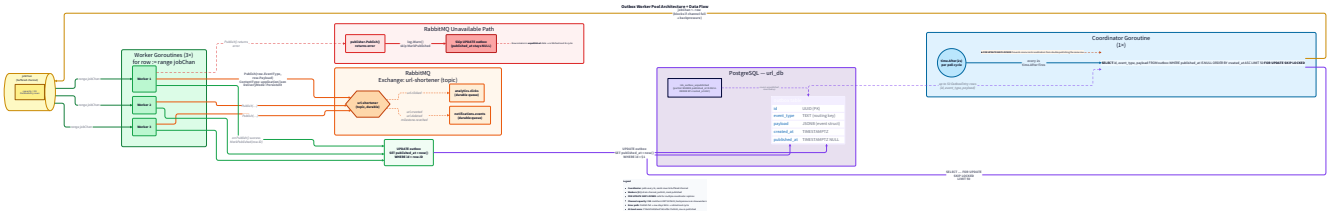
```go
// main.go shutdown sequence

ctx, cancel := context.WithCancel(context.Background())

defer cancel()

sigCh := make(chan os.Signal, 1)

signal.Notify(sigCh, syscall.SIGTERM, syscall.SIGINT)

go poller.Start(ctx) // starts coordinator + 3 workers

<-sigCh

log.Info().Msg("shutdown signal received")

cancel() // signals coordinator and workers to stop

// Give workers up to 10s to drain the channel and finish in-flight publishes.

shutdownCtx, shutdownCancel := context.WithTimeout(context.Background(), 10*time.Second)

defer shutdownCancel()

server.Shutdown(shutdownCtx)
```

9.3 Cache-Aside Read Safety

Multiple concurrent redirect handlers may simultaneously experience a cache miss for the same short code. Each will independently query PostgreSQL and then call `cache.Set`. This is safe:

- PostgreSQL reads are non-mutating (SELECT); no locking needed.

- Concurrent `cache.Set` calls for the same key result in idempotent SET operations (last writer wins, which is acceptable — all writers have the same data from the DB).
- No "thundering herd" mitigation is implemented; at the scale of this project it is unnecessary.



10. Implementation Sequence with Checkpoints

Phase 1 — Schema Migrations + Indexes (1–1.5 hr)

1. Write `migrations/001_create_urls.sql` and `migrations/002_create_outbox.sql`.
2. Apply migrations: `docker exec -i url_db psql -U url_user -d url_db < services/url-service/migrations/001_create_urls.sql`
3. Apply: `docker exec -i url_db psql -U url_user -d url_db < services/url-service/migrations/002_create_outbox.sql`
4. Verify schema: `docker exec url_db psql -U url_user -d url_db -c "\d urls"` and `"\d outbox"`.
5. Verify indexes: `docker exec url_db psql -U url_user -d url_db -c "\di"` — expect `idx_urls_short_code`, `idx_urls_user_id_created`, `idx_outbox_unpublished`. **Checkpoint:** Both tables exist with correct column types and NOT NULL constraints. Indexes are visible in `\di`. `EXPLAIN SELECT * FROM urls WHERE short_code = 'abc'` shows `Index Scan` on `idx_urls_short_code`.

Phase 2 — Base62 Code Generator (1–1.5 hr)

1. Add `github.com/google/uuid v1.6.0` to `go.mod` (already needed in M2; verify it's present).
2. Write `codegen/codegen.go` with `base62Alphabet`, `codeLength`, `base62Generator`, `Generate()`.
3. Write `codegen/codegen_test.go` (§ 11 tests).
4. Run `go test ./codegen/... -v -count=1`. **Checkpoint:** `TestGenerate_Length` passes (output is always 7 chars). `TestGenerate_Base62Alphabet` passes (all chars in `base62Alphabet`). `TestGenerate_Randomness` passes (1000 calls produce no duplicates with >99.999% probability). `TestGenerate_NoCryptoMathRand` passes — verify source code import: `grep "math/rand" codegen/codegen.go` returns nothing.

Phase 3 — URLRepository + OutboxRepository (1.5–2 hr)

1. Write `repository/url_repository.go`: `URL`, `URLSummary`, sentinel errors, `URLRepository` interface, `pgxURLRepository`, all four methods.
2. Write `repository/outbox_repository.go`: `OutboxEntry`, `OutboxRepository` interface, `pgxOutboxRepository`, three methods.
3. Write `repository/url_repository_test.go` and `repository/outbox_repository_test.go`.
4. Run `go test -tags integration ./repository/... -v`. **Checkpoint:** `TestCreate_HappyPath` inserts a URL and finds it by code. `TestCreate_CodeConflict` returns `ErrCodeConflict`. `TestSoftDelete_WrongOwner` returns `ErrNotOwner`. `TestListByUser_Cursor` returns second page when `afterID` is set. `EXPLAIN ANALYZE` on `ListByUser` SQL shows `Index Scan` on `idx_urls_user_id_created` (no sequential scan).

Phase 4 — Redis CacheClient (1–1.5 hr)

1. Write `cache/cache.go`: `CachedURL`, `RedisKey`, `CacheClient` interface, `redisCacheClient`, `Get` / `Set` / `Del`, nil-safety in every method.
2. Write `cache/cache_test.go`.
3. Run `go test ./cache/... -v` (unit tests use a mock or real Redis depending on tag). **Checkpoint:** `TestGet_CacheMiss_ReturnsErrCacheMiss` passes. `TestGet_NilClient_NoPanic` passes (nil `*redis.Client` returns `ErrCacheMiss`, no panic). `TestSet_TTLApplied` verifies the key has a TTL via `redis-cli TTL url:testcode`. `TestDel_KeyRemoved` verifies key is gone after `Del`. `TestComputeTTL_CappedAt1Hour` verifies `min(expires_at, 1h)` logic.

Phase 5 — POST /shorten + DELETE /urls/:code Handlers (1.5–2 hr)

1. Write `service/url_service.go`: `URLService` struct, `NewURLService`, `Shorten`, `Delete` (stubs for `Redirect` and `ListURLs` — return `nil, ErrURLNotFound` temporarily).
2. Write `handler/helpers.go`: `writeJSON`, `writeError`, `extractClientIP`, `hashIP`, `ptr` helper.
3. Write `handler/shorten.go`: `NewShortenHandler`.
4. Write `handler/delete.go`: `NewDeleteURLHandler`.
5. Wire in `main.go`: construct all dependencies, register routes.

6. Run `docker compose up --build url-service`. **Checkpoint:**

```bash
# Register a user first (user-service must be running)
TOKEN=$(curl -s -X POST http://localhost:8083/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"password123"}' | jq -r .token)

# Shorten
curl -s -X POST http://localhost:8081/shorten \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"url":"https://www.example.com/very/long/path"}' | jq .

# Expected: {"short_code":"<7chars>","short_url":"http://localhost:8081/<code>","original_url":"..."}

# Duplicate custom code
curl -s -o /dev/null -w "%{http_code}" -X POST http://localhost:8081/shorten \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"url":"https://example.org","custom_code":"<same_code>"}'

# Expected: 409

# Delete
curl -s -o /dev/null -w "%{http_code}" -X DELETE http://localhost:8081/urls/<code> \
  -H "Authorization: Bearer $TOKEN"

# Expected: 204
```

Phase 6 — GET /:code Redirect (1.5–2 hr)

1. Implement `URLService.Redirect` in `service/url_service.go` (§ 5.4).
2. Write `handler/redirect.go`: `NewRedirectHandler`.
3. Register `GET /{code}` in `main.go`.
4. Run `docker compose up --build url-service`. **Checkpoint:**

```bash
CODE=$(curl -s -X POST http://localhost:8081/shorten \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"url":"https://www.google.com"}' | jq -r .short_code)

# Redirect (follow → expect google.com)
curl -v "http://localhost:8081/$CODE" 2>&1 | grep "< HTTP\|< Location"

# Expected: HTTP/1.1 301, Location: https://www.google.com

# Cache hit (second request)
curl -v "http://localhost:8081/$CODE" 2>&1 | grep "< HTTP"

# Redis: redis-cli GET "url:$CODE" → returns JSON

# Non-existent code
curl -s -o /dev/null -w "%{http_code}" "http://localhost:8081/ZZZZZZZ"

# Expected: 404

# Run benchmark
go test -bench=BenchmarkRedirectCacheHit -benchtime=10s ./handler/...

# Target: < 5ms p99 at 100 concurrent (see § 9 for benchmark setup)
```

Phase 7 — GET /urls Cursor Pagination (1–1.5 hr)

1. Implement `URLService.ListURLs` in `service/url_service.go` (§ 5.6).
2. Write `handler/list.go`: `NewListURLsHandler`.
3. Register `GET /urls` in `main.go`. **Checkpoint:**

```bash
# Create 5 URLs
for i in {1..5}; do
  curl -s -X POST http://localhost:8081/shorten \
    -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
    -d "{\"url\":\"https://example.com/$i\"}" > /dev/null
done
# First page (limit=2)
RESP=$(curl -s "http://localhost:8081/urls?limit=2" -H "Authorization: Bearer $TOKEN")
echo $RESP | jq '.urls | length'  # Expected: 2
CURSOR=$(echo $RESP | jq -r '.next_cursor')  # non-null
# Second page
curl -s "http://localhost:8081/urls?limit=2&after=$CURSOR" \
  -H "Authorization: Bearer $TOKEN" | jq '.urls | length'  # Expected: 2
# Verify no seq scan
docker exec url_db psql -U url_user -d url_db -c \
  "EXPLAIN SELECT id, short_code FROM urls WHERE user_id = 'some-uuid' ORDER BY created_at DESC LIMIT 3;"
# Must show "Index Scan" not "Seq Scan"
```

Phase 8 — Outbox Worker Pool + AMQPPublisher (2–2.5 hr)

1. Write `amqp/publisher.go`: `AMQPPublisher` interface, `amqpPublisher` with `sync.Mutex`, `Publish`.
2. Write `outbox/poller.go`: `OutboxPoller` struct, `Start(ctx context.Context)` (launches coordinator + 3 workers).
3. Wire poller in `main.go`: construct `AMQPPublisher`, construct `OutboxPoller`, call `go poller.Start(ctx)`.
4. Run `docker compose up --build url-service`. **Checkpoint:**

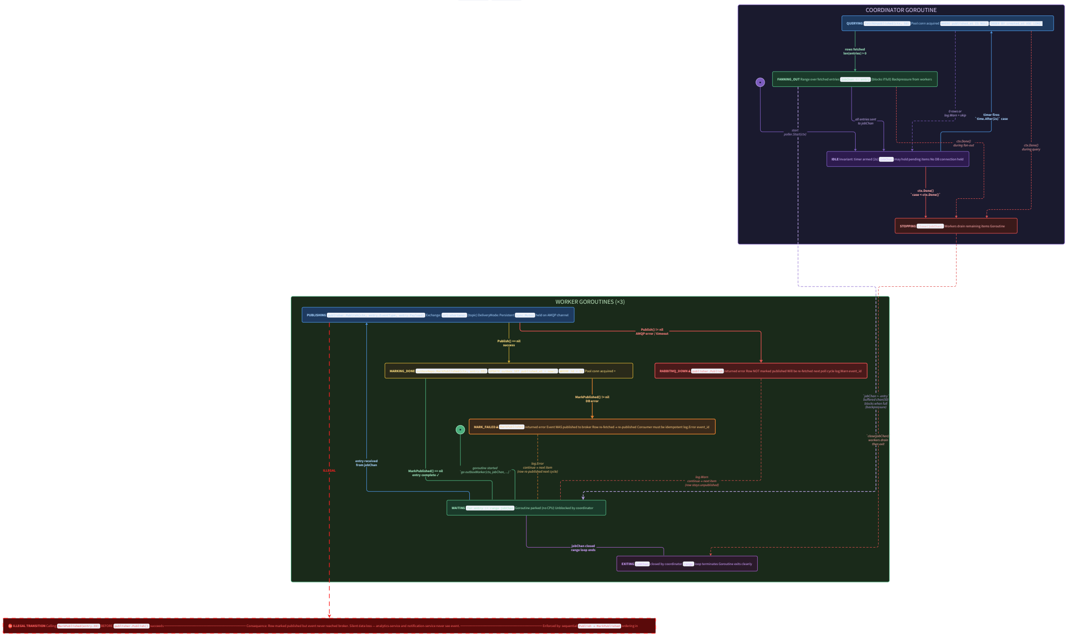
```bash
# Create a URL (writes outbox row)
curl -s -X POST http://localhost:8081/shorten \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"url":"https://www.rabbitmq-test.com"}'
# Wait 3 seconds for poller cycle
sleep 3
# Verify outbox row is marked published
docker exec url_db psql -U url_user -d url_db -c \
  "SELECT event_type, published_at FROM outbox ORDER BY created_at DESC LIMIT 1;"
# Expected: published_at IS NOT NULL
# Verify message in RabbitMQ management UI:
# http://localhost:15672 → Queues → notifications.events → Get Messages
# Should see URLCreatedEvent payload
# Test RabbitMQ down: stop broker, create URL, verify service stays healthy
docker compose stop rabbitmq
curl -s -o /dev/null -w "%{http_code}" -X POST http://localhost:8081/shorten \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"url":"https://resilience-test.com"}'
# Expected: 201 (not 503)
docker compose start rabbitmq
sleep 15  # reconnect + poll cycle
docker exec url_db psql -U url_user -d url_db -c \
  "SELECT count(*) FROM outbox WHERE published_at IS NULL;"
# Expected: 0
```

Phase 9 — Unit Tests + Benchmarks (1–1.5 hr)

1. Write all test files from § 11.
2. Run `go test ./... -v -count=1`.
3. Run benchmarks: `go test -bench=. -benchmem ./...`. **Checkpoint:** All tests pass. `BenchmarkRedirectCacheHit` shows p99 < 5ms at 100 concurrent. `BenchmarkRedirectCacheMiss` shows p99 < 20ms. `go vet ./...` clean. `go test -race ./...` clean (no race conditions).

Outbox Poller State Machine

Module: `url-shortener-m3` · `outbox/poller.go` Coordinator: 1 goroutine · Workers: 3 goroutines · Channel buffer: 50



Poll Cycle Timing

- Interval: 2s (`time.After`)
- Batch size: 50 rows per cycle
- Partial index: `WHERE published_at IS NULL`
- Oldest-first ordering ensures fairness
- Coordinator blocks on `jobChan <- entry` if all 3 workers are busy (backpressure)

Worker Concurrency Contract

- 3 workers share 1 `amqp091.Channel`
- `sync.Mutex` in `amqpPublisher.Publish` serializes concurrent AMQP writes
- Workers are independent: one RABBITMQ_DOWN does not block others
- Graceful shutdown: workers drain jobChan before exiting (range closes)

11. Test Specification

11.1 `codegen/codegen_test.go`

```go
// TestGenerate_Length: Generate() always returns exactly 7 characters.

// Run 1000 times; assert len(code) == 7 for all.

func TestGenerate_Length(t *testing.T)

// TestGenerate_Base62Alphabet: every character in generated code is in base62Alphabet.

// Run 1000 times; assert each char exists in the alphabet string.

func TestGenerate_Base62Alphabet(t *testing.T)

// TestGenerate_Randomness: 1000 consecutive calls produce no duplicates.

// Failure probability: 62^7 = 3.5 trillion; collision among 1000 is negligible.

func TestGenerate_Randomness(t *testing.T)

// TestGenerate_NotMathRand: verify crypto/rand is used.

// Indirect test: seed math/rand with a fixed seed; if codes match the seeded sequence,

// math/rand is being used (fail). This is a code review check, not runtime check —

// document in test comment that the import list must not contain "math/rand".

func TestGenerate_NoMathRandImport(t *testing.T) // parse AST or grep imports

// TestGenerate_CollisionRetryIn_URLService:

// Inject a mock CodeGenerator that returns the same code for the first 4 calls,

// then a unique code on the 5th. Verify URLService.Shorten succeeds with 5th code.

func TestShorten_CollisionRetry(t *testing.T)

// TestShorten_CollisionExhausted:

// Inject a mock CodeGenerator that always returns the same code (DB always rejects).

// Verify URLService.Shorten returns ErrExhausted after 5 attempts.

func TestShorten_CollisionExhausted(t *testing.T)

// BenchmarkGenerate: measure throughput.

// Expected: > 100,000 Generate() calls/sec (pure in-memory; should be fast).

func BenchmarkGenerate(b *testing.B)
```

11.2 `repository/url_repository_test.go` (integration)

```go
// Build tag: //go:build integration

// Requires: url_db running. DSN from TEST_DATABASE_DSN env var.

// TestCreate_HappyPath: create URL → FindByCode returns same data.

func TestCreate_HappyPath(t *testing.T)

// TestCreate_CodeConflict: create two URLs with same short_code → second returns ErrCodeConflict.

func TestCreate_CodeConflict(t *testing.T)

// TestCreate_InTransaction_RollbackOnError:

// Begin tx, Create URL, DON'T commit. In separate connection, FindByCode → ErrURLNotFound.

// Verifies that uncommitted tx is not visible.

func TestCreate_InTransaction_RollbackOnError(t *testing.T)

// TestFindByCode_NotFound: ErrURLNotFound for unknown code.

func TestFindByCode_NotFound(t *testing.T)

// TestSoftDelete_HappyPath: SoftDelete sets is_active=false; FindByCode returns is_active=false.

func TestSoftDelete_HappyPath(t *testing.T)

// TestSoftDelete_WrongOwner: SoftDelete with wrong userID returns ErrNotOwner.

func TestSoftDelete_WrongOwner(t *testing.T)

// TestSoftDelete_NotFound: SoftDelete on non-existent code returns ErrURLNotFound.

func TestSoftDelete_NotFound(t *testing.T)

// TestListByUser_EmptyCursor: first page returns newest-first order.

func TestListByUser_EmptyCursor(t *testing.T)

// TestListByUser_WithCursor: second page starts after cursor row.

func TestListByUser_WithCursor(t *testing.T)

// TestListByUser_UsesIndex: EXPLAIN on the query must not contain "Seq Scan".

// Use pgx to execute EXPLAIN and assert output.

func TestListByUser_UsesIndex(t *testing.T)
```

11.3 `repository/outbox_repository_test.go` (integration)

```go
// TestInsert_HappyPath: Insert outbox row in tx, commit, FetchUnpublished returns it.

func TestInsert_HappyPath(t *testing.T)

// TestInsert_RollsBackWithCaller: Insert in tx, rollback → FetchUnpublished returns empty.

func TestInsert_RollsBackWithCaller(t *testing.T)

// TestFetchUnpublished_Limit: Insert 60 rows, FetchUnpublished(50) returns exactly 50.

func TestFetchUnpublished_Limit(t *testing.T)

// TestMarkPublished_IdempotentOnDouble: call MarkPublished twice on same ID → no error.

func TestMarkPublished_IdempotentOnDouble(t *testing.T)

// TestFetchUnpublished_ExcludesPublished: published rows not returned.

func TestFetchUnpublished_ExcludesPublished(t *testing.T)
```

11.4 `cache/cache_test.go`

```go
// Unit tests using a real Redis (TEST_REDIS_ADDR) or mock; nil-client tests are pure unit.

// TestGet_NilClient_ReturnsMiss: nil *redis.Client → ErrCacheMiss, no panic.

func TestGet_NilClient_ReturnsMiss(t *testing.T)

// TestSet_NilClient_NoOp: nil *redis.Client → Set returns without panic.

func TestSet_NilClient_NoOp(t *testing.T)

// TestDel_NilClient_NoOp: nil *redis.Client → Del returns without panic.

func TestDel_NilClient_NoOp(t *testing.T)

// TestGet_Miss_ReturnsErrCacheMiss: key not in Redis → ErrCacheMiss.

func TestGet_Miss_ReturnsErrCacheMiss(t *testing.T)

// TestSetGet_RoundTrip: Set then Get returns same CachedURL.

func TestSetGet_RoundTrip(t *testing.T)

// TestSet_TTLApplied: after Set, redis-cli TTL shows positive value ≤ requested TTL.

func TestSet_TTLApplied(t *testing.T)

// TestDel_RemovesKey: Set then Del → Get returns ErrCacheMiss.

func TestDel_RemovesKey(t *testing.T)

// TestComputeTTL_NoExpiry: expiresAt=nil → returns 1h.

func TestComputeTTL_NoExpiry(t *testing.T)

// TestComputeTTL_ExpiryWithin1h: expiresAt=30min from now → returns ~30min.

func TestComputeTTL_ExpiryWithin1h(t *testing.T)

// TestComputeTTL_ExpiryBeyond1h: expiresAt=2h from now → returns 1h.

func TestComputeTTL_ExpiryBeyond1h(t *testing.T)

// TestComputeTTL_AlreadyExpired: expiresAt=past → returns 1s (defensive).

func TestComputeTTL_AlreadyExpired(t *testing.T)
```

11.5 `handler/handler_test.go` (integration + unit)

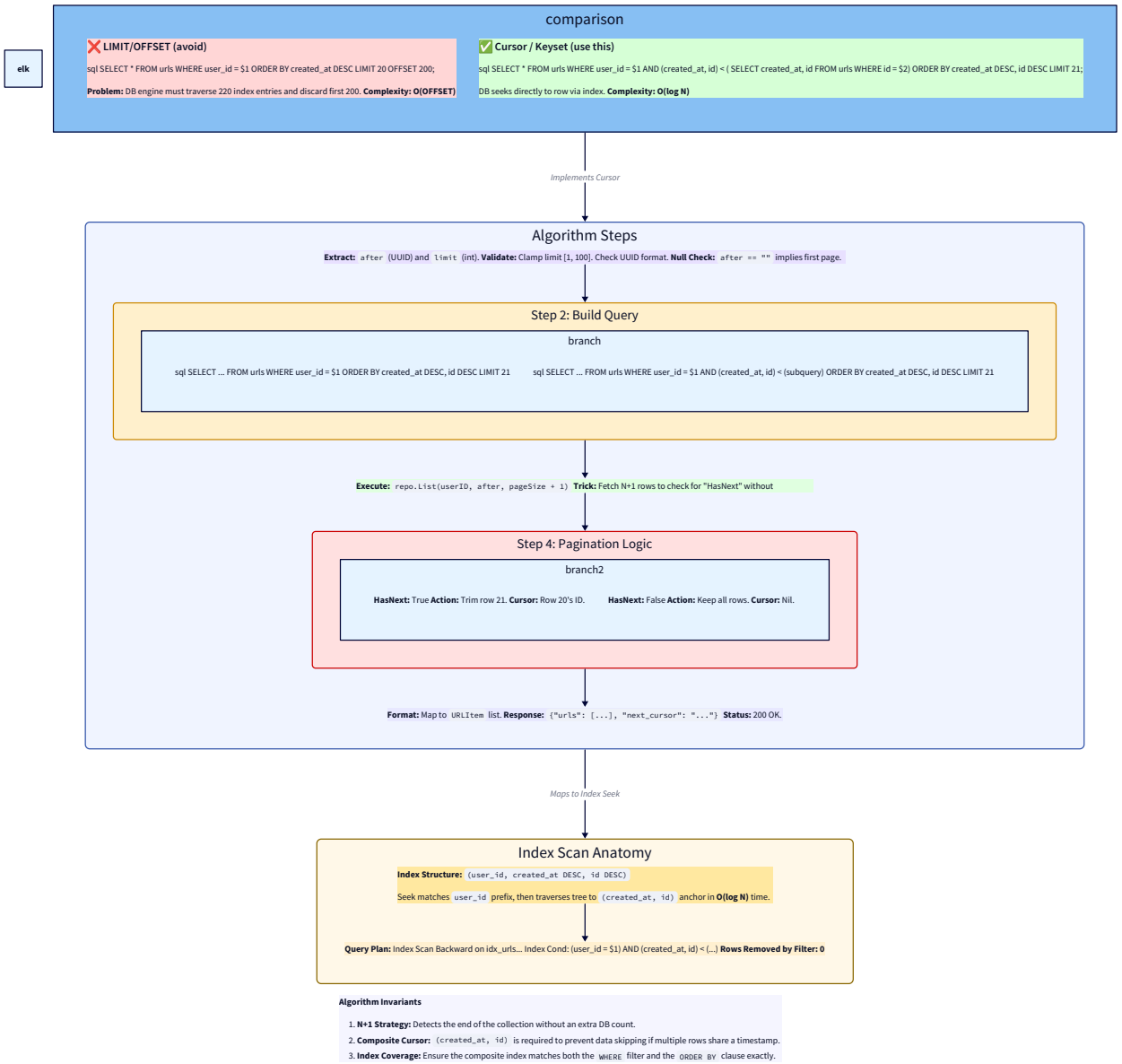
```go
// Unit tests use httptest.NewRecorder + httptest.NewRequest.

// Integration tests tag: //go:build integration

// TestShorten_HappyPath: valid JWT + valid URL → 201, short_code is 7-char base62.

func TestShorten_HappyPath(t *testing.T)

// TestShorten_NoAuth: missing Authorization → 401.

func TestShorten_NoAuth(t *testing.T)

// TestShorten_InvalidURL_MissingScheme: "example.com" (no https://) → 400.

func TestShorten_InvalidURL_MissingScheme(t *testing.T)

// TestShorten_ExpiresAtPast: expires_at in past → 400.

func TestShorten_ExpiresAtPast(t *testing.T)

// TestRedirect_CacheHit: set key in Redis, GET /:code → 301, Location matches.

func TestRedirect_CacheHit(t *testing.T)

// TestRedirect_CacheMiss_DBFallback: no Redis key, URL in DB → 301, Redis populated.

func TestRedirect_CacheMiss_DBFallback(t *testing.T)

// TestRedirect_NotFound: unknown code → 404.

func TestRedirect_NotFound(t *testing.T)

// TestRedirect_Expired: expires_at in past → 410.

func TestRedirect_Expired(t *testing.T)

// TestRedirect_Inactive: is_active=false → 410.

func TestRedirect_Inactive(t *testing.T)

// TestDelete_HappyPath: owner deletes URL → 204, Redis key gone.

func TestDelete_HappyPath(t *testing.T)

// TestDelete_WrongOwner: different JWT sub → 403.

func TestDelete_WrongOwner(t *testing.T)

// TestDelete_NotFound: non-existent code → 404.

func TestDelete_NotFound(t *testing.T)

// TestListURLs_Pagination: create 5, page by 2, verify cursor chain covers all 5.

func TestListURLs_Pagination(t *testing.T)

// BenchmarkRedirectCacheHit: serve redirect from Redis cache at 100 concurrent.

// Uses httptest.Server with a pre-seeded Redis entry.

// Assert: p99 latency < 5ms.

func BenchmarkRedirectCacheHit(b *testing.B)

// BenchmarkRedirectCacheMiss: serve redirect from PostgreSQL (Redis disabled).

// Assert: p99 latency < 20ms.

func BenchmarkRedirectCacheMiss(b *testing.B)
```

Benchmark implementation sketch:

```go
func BenchmarkRedirectCacheHit(b *testing.B) {

    // Setup: real Redis with seeded URL entry, real DB with URL row.

    // Use httptest.NewServer with the handler under test.

    srv := httptest.NewServer(handler.NewRedirectHandler(urlSvc, log))

    defer srv.Close()

    b.ResetTimer()

    b.SetParallelism(100)

    b.RunParallel(func(pb *testing.PB) {

        for pb.Next() {

            resp, err := http.Get(srv.URL + "/" + seededCode)

            if err != nil { b.Fatal(err) }

            resp.Body.Close()

            if resp.StatusCode != http.StatusMovedPermanently {

                b.Fatalf("unexpected status: %d", resp.StatusCode)

            }

        }

    })

    // Use b.ReportMetric to record p99 if using a custom histogram.

}
```

Cursor-Based Pagination Algorithm — GET /urls

Why not LIMIT/OFFSET? OFFSET N forces the DB to scan and discard N rows — O(N) full-index skip scan. Cursor uses `(created_at, id) <` predicate → O(log N) index range scan every time.



Algorithm Invariants

1. **N+1 Strategy:** Detects the end of the collection without an extra DB count.
2. **Composite Cursor:** `(created_at, id)` is required to prevent data skipping if multiple rows share a timestamp.
3. **Index Coverage:** Ensure the composite index matches both the `WHERE` filter and the `ORDER BY` clause exactly.

12. Environment Variable Reference

Variable	Required	Example	Description
`PORT`	Yes	`8081`	HTTP listen port
`SERVICE_NAME`	Yes	`url-service`	Embedded in log lines
`DATABASE_DSN`	Yes	`postgres://url_user:url_secret@url_db:5432/url_db?sslmode=disable`	PostgreSQL connection
`REDIS_ADDR`	Yes	`redis:6379`	Redis address; unavailability is non-fatal
`RABBITMQ_URL`	Yes	`amqp://admin:admin@rabbitmq:5672/`	AMQP broker
`JWT_SECRET`	Yes	`change-me-32-bytes-minimum!!!!!!`	JWT verification (not signing — url-service never issues tokens)
`BASE_URL`	Yes	`http://localhost:8081`	Prepended to short_code in API response `short_url`
`CLICK_SALT`	No	`random-salt-value`	Salt for IP hashing. Empty string if unset — document in `.env.example`
`BASE_URL` in production would be the public domain (e.g., `https://short.example.com`). In docker-compose it is the gateway's public URL. For this milestone, use the service's own URL; update to the gateway URL in M5.			
Add to `docker-compose.yml` for url-service:			

```yaml
environment:
  JWT_SECRET: "change-me-to-at-least-32-random-bytes"
  BASE_URL: "http://localhost:8081"
  CLICK_SALT: "replace-with-random-value"
```

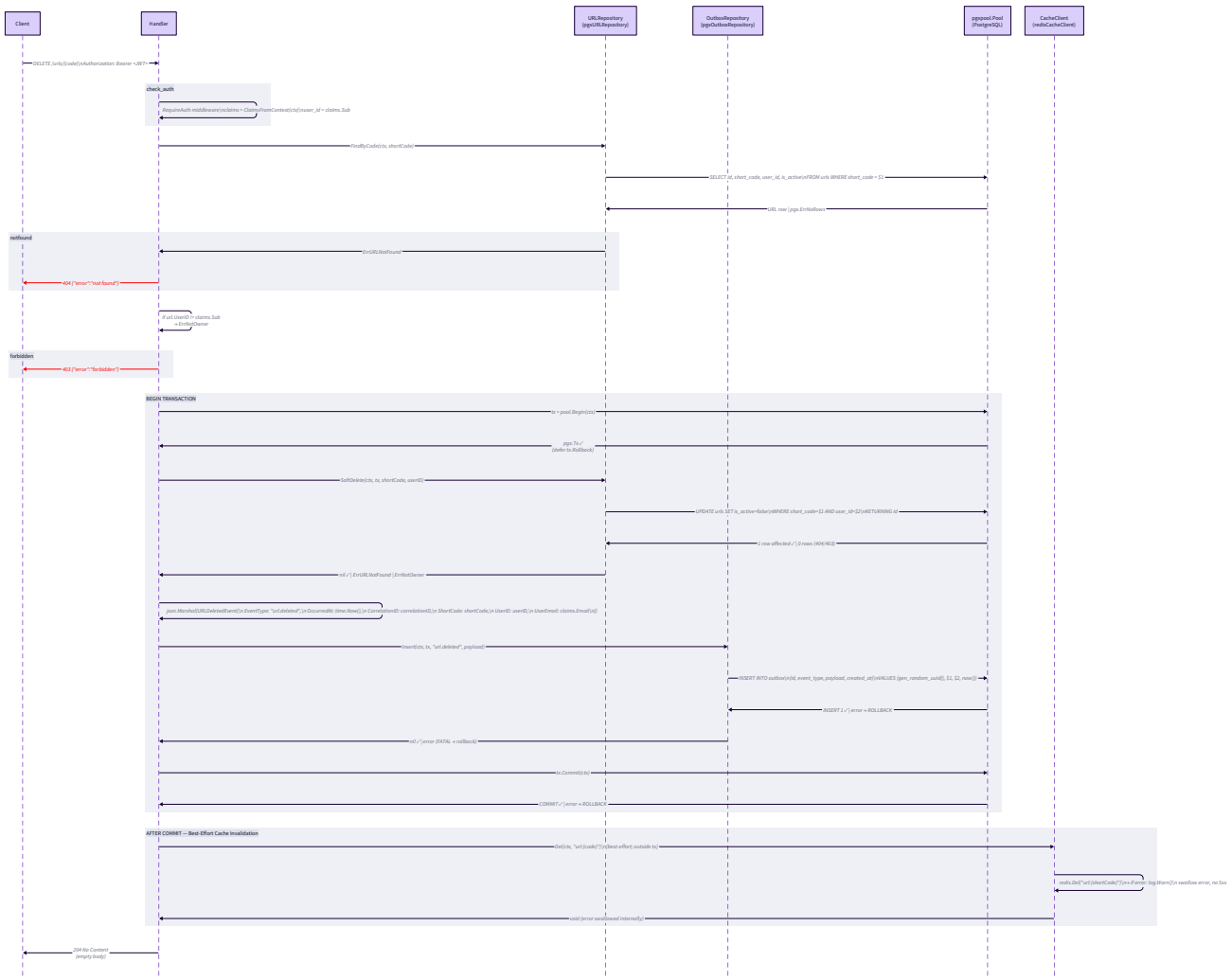
13. Performance Targets

Operation	Target	How to Measure
`GET /:code` cache hit p99	< 5ms	`go test -bench=BenchmarkRedirectCacheHit -benchtime=30s ./handler/...` at 100 parallel
`GET /:code` cache miss p99	< 20ms	`go test -bench=BenchmarkRedirectCacheMiss -benchtime=30s ./handler/...`
`POST /shorten` p99	< 50ms	`wrk -t4 -c20 -d30s -s shorten.lua http://localhost:8081/shorten`
Outbox poll lag (write → publish)	< 4s	Create URL, measure time until `published_at IS NOT NULL` in DB
`codegen.Generate()` throughput	> 100,000/sec	`go test -bench=BenchmarkGenerate ./codegen/...`
`ListURLs` cursor query	No seq scan	`EXPLAIN ANALYZE` on list query; assert `Index Scan` on `idx_urls_user_id_created`
`GET /urls` p99 (20-item page)	< 15ms	`wrk -t2 -c10 -d10s` with valid JWT
Container image size	< 20MB	`docker image ls url-service`
`go build ./...` incremental	< 10s	CI warm build
Outbox worker pool drain on shutdown	< 10s	Send SIGTERM; measure time until in-flight jobs complete

14. Anti-Pattern Guard Rail Checklist

Before completing this milestone, verify:

- [ ] **No synchronous call to analytics-service or notification-service.** `grep -r "analytics-service\|notification-service" services/url-service/` returns nothing except comments.
- [ ] **No shared database.** `services/url-service/` contains zero references to `analytics_db`, `user_db`, or `notification_db` connection strings.
- [ ] **`math/rand` is not imported.** `grep -r '"math/rand"' services/url-service/codegen/` returns nothing. Only `crypto/rand` is used.
- [ ] **No dual-write (URL insert and outbox insert in separate transactions).** The `createURLWithOutbox` function calls `pool.Begin` once and passes the same `pgx.Tx` to both `urlRepo.Create` and `outboxRepo.Insert`.
- [ ] **Redis error never causes 5xx.** `CacheClient.Get` returns `ErrCacheMiss` (not a real error) for all Redis failures. Handlers treat `ErrCacheMiss` as a DB-fallback signal.
- [ ] **`expires_at` TTL applied to Redis entries.** `cache.Set` is never called with `ttl=0` or a negative duration. `computeTTL` returns at minimum `1 * time.Second`.
- [ ] **Cache invalidated on DELETE.** `URLService.Delete` calls `cache.Del` after `tx.Commit`. Order is: commit → invalidate (never invalidate before commit; that would create a window where the DB has the old state and the cache is empty, causing a stale repopulation).
- [ ] **Ownership check on DELETE.** `urlRepo.SoftDelete` accepts `ownerID` and returns `ErrNotOwner` if mismatch. The handler maps this to 403.
- [ ] **Worker pool, not a single goroutine.** `outbox/poller.go` starts exactly 3 worker goroutines. Verify with `grep "go outboxWorker" outbox/poller.go` → 3 calls in a loop.
- [ ] **No outbox polling with `time.Sleep` in workers.** Workers block on `range jobChan`. Only the coordinator uses `time.After(2s)`. Workers have no sleep/timer.



Technical Design Specification

Module: Analytics Service — Click Ingestion + Stats API

Module ID: `url-shortener-m4`

1. Module Charter

The Analytics Service is a pure downstream consumer within the URL shortener system. It ingests `URLClickedEvent` messages from the RabbitMQ queue `analytics.clicks`, stores privacy-preserving click records in its private PostgreSQL database, detects click milestone thresholds (10, 100, 1000), and exposes a read-only HTTP stats API. It is the exclusive owner of the `analytics_db` PostgreSQL instance — no other service reads or writes to it. This module does **not** call the URL Service to validate short codes. It does **not** call the User Service to resolve user identities. It does **not** share any database schema with any other service. It does **not** issue JWTs or perform authentication — the stats API (`GET /stats/:code`) is intentionally public with no auth requirement per the project spec. It does **not** implement fan-out reads, materialized views, or any form of caching for stats queries (indexed PostgreSQL queries are sufficient at this scale). **Upstream dependencies:** M1 foundation (Docker Compose stack, `shared/events`, `shared/logger`); RabbitMQ exchange `url-shortener` (topic) declared by url-service (M3). The `analytics.clicks` queue is declared and bound by this service's own startup code — the consumer is responsible for its own queue declaration. The `MilestoneReachedEvent` struct is defined in `shared/events` (M1). **Downstream dependencies:** notification-service (M5) consumes `MilestoneReachedEvent` published by this service. No other service depends on analytics-service at runtime. **Invariants that must always hold after this milestone:**

- A `URLClickedEvent` with a given `EventID` is inserted into the `clicks` table at most once, regardless of how many times RabbitMQ delivers it. The `processed_events` deduplication table is the enforcement mechanism.
- Every click insertion, idempotency mark, and optional milestone insertion execute within a single PostgreSQL transaction. A crash between DB commit and RabbitMQ ACK causes re-delivery; the idempotency check makes re-processing a no-op.
- A poison message (malformed JSON, missing required fields) is always ACKed and never requeued. It is logged at `error` level. The consumer loop never panics.
- `GET /health` returns 200 even when the RabbitMQ consumer is paused or the connection is down. Consumer health is degraded, not fatal.
- Raw IP addresses are never stored. The `ip_hash` field on `URLClickedEvent` (computed by url-service) is stored as-is — analytics-service never sees or stores plaintext IPs.

2. File Structure

Create files in the numbered order below. All paths are relative to the monorepo root.

```
url-shortener/
│
└── services/analytics-service/
    ├── migrations/
    │   ├── 01  001_create_clicks.sql
    │   ├── 02  002_create_milestones.sql
    │   └── 03  003_create_processed_events.sql
    ├── repository/
    │   ├── 04  click_repository.go
    │   ├── 05  click_repository_test.go
    │   ├── 06  processed_event_repository.go
    │   ├── 07  processed_event_repository_test.go
    │   ├── 08  milestone_repository.go
    │   └── 09  milestone_repository_test.go
    ├── consumer/
    │   ├── 10  consumer.go             # AMQPConsumer interface + amqp091 implementation
    │   ├── 11  processor.go            # ClickProcessor: idempotency + insert + milestone
    │   └── 12  processor_test.go
    ├── milestone/
    │   ├── 13  detector.go             # MilestoneDetector: threshold logic
    │   └── 14  detector_test.go
    ├── publisher/
    │   ├── 15  publisher.go            # AMQPPublisher for MilestoneReachedEvent
    │   └── 16  publisher_test.go
    ├── handler/
    │   ├── 17  stats.go                # GET /stats/:code
    │   ├── 18  timeline.go            # GET /stats/:code/timeline
    │   ├── 19  health.go               # GET /health
    │   ├── 20  helpers.go              # writeJSON, writeError
    │   └── 21  handler_test.go
    └── 22  main.go
```

Total new files: 22 (excluding auto-generated `go.sum` changes).

3. Complete Data Model

3.1 PostgreSQL Schema — `migrations/001_create_clicks.sql`

```sql
-- migrations/001_create_clicks.sql

BEGIN;

CREATE TABLE IF NOT EXISTS clicks (

    id          UUID        PRIMARY KEY DEFAULT gen_random_uuid(),

    short_code  TEXT        NOT NULL,

    clicked_at  TIMESTAMPTZ NOT NULL,

    ip_hash     TEXT        NOT NULL,    -- SHA-256(ip+salt) from url-service; never raw IP

    user_agent  TEXT        NOT NULL DEFAULT '',

    referer     TEXT        NULL        -- NULL = no Referer header present

);

-- Primary analytics query: count/aggregate clicks for a short_code ordered by time.

-- Covers: total_clicks, clicks_last_24h, clicks_last_7d, timeline bucketing.

CREATE INDEX IF NOT EXISTS idx_clicks_short_code_time

    ON clicks(short_code, clicked_at DESC);

-- Referer stats query: top referers for a short_code where referer is not null.

-- Partial index keeps size small (many clicks have NULL referer).

CREATE INDEX IF NOT EXISTS idx_clicks_referer

    ON clicks(short_code, referer)

    WHERE referer IS NOT NULL;

COMMIT;
```

Field rationale:

- `short_code TEXT NOT NULL` : No FK to urls table — analytics-service owns no url data. Trusts the event payload. TEXT avoids the 10-char VARCHAR limit if future short codes change length.
- `clicked_at TIMESTAMPTZ NOT NULL` : The timestamp of the original click as embedded in `URLClickedEvent.OccurredAt` , not `now()` . Using event time (not ingestion time) gives accurate time-series analytics regardless of consumer lag.
- `ip_hash TEXT NOT NULL` : Already hashed by url-service. Analytics-service is never in possession of raw IPs. Storing the hash enables (future) per-IP deduplication without privacy violation.
- `referer TEXT NULL` : Absent `Referer` header is stored as NULL, not empty string. This allows the partial index `WHERE referer IS NOT NULL` to exclude the majority of rows from the referer-stats index.
- `user_agent TEXT NOT NULL DEFAULT ''` : Empty string when `User-Agent` header was absent. Default prevents NOT NULL violations on events from non-browser clients.

3.2 PostgreSQL Schema — `migrations/002_create_milestones.sql`

```sql
-- migrations/002_create_milestones.sql

BEGIN;

CREATE TABLE IF NOT EXISTS milestones (

    id           UUID        PRIMARY KEY DEFAULT gen_random_uuid(),

    short_code   TEXT        NOT NULL,

    milestone    INT         NOT NULL,       -- 10 | 100 | 1000

    triggered_at TIMESTAMPTZ NOT NULL DEFAULT now(),

    UNIQUE (short_code, milestone)           -- each threshold crossed at most once per code

);

-- Query: "what is the highest milestone already triggered for this code?"

CREATE INDEX IF NOT EXISTS idx_milestones_short_code

    ON milestones(short_code, milestone DESC);

COMMIT;
```

Field rationale:

- `UNIQUE (short_code, milestone)`: Prevents duplicate milestone rows from concurrent processing or re-delivery. If two consumers race to detect the same milestone, the second `INSERT` fails on the unique constraint — the caller catches this and treats it as a no-op (milestone already recorded).
- `milestone INT`: Stored as plain integer (10, 100, 1000) for query simplicity. Not stored as an enum because the set of thresholds may expand (10,000 etc.) without schema changes.

3.3 PostgreSQL Schema — `migrations/003_create_processed_events.sql`

```sql
-- migrations/003_create_processed_events.sql

BEGIN;

CREATE TABLE IF NOT EXISTS processed_events (

    event_id    UUID        PRIMARY KEY,     -- = URLClickedEvent.EventID

    processed_at TIMESTAMPTZ NOT NULL DEFAULT now()

);

-- The PRIMARY KEY on event_id is itself the index used by IsProcessed SELECT.

-- No additional index needed.

COMMIT;
```

Field rationale:

- `event_id UUID PRIMARY KEY`: The `EventID` from `URLClickedEvent` is the deduplication key. UUID PKs have B-tree index by default — `SELECT 1 FROM processed_events WHERE event_id = $1` uses an index scan in O(log n). No separate index is needed.
- `processed_at TIMESTAMPTZ`: Audit field. Useful for debugging and for a future cleanup job that purges old dedup entries (not implemented in this milestone).
- **No FK to clicks.id**: Intentional. The idempotency check is independent of the click row — if `MarkProcessed` succeeds but `Insert` fails, the event is marked processed but not inserted. This edge case is handled by inserting the click *before* marking processed within the same transaction (see § 5.2).

3.4 Go Structs

```go
// repository/click_repository.go

package repository

import "time"

// Click is a single recorded redirect event stored in the clicks table.

// Fields map 1:1 to table columns; no derived fields.

type Click struct {

    ID        string    // UUID v4; generated by DB DEFAULT

    ShortCode string    // matches URLClickedEvent.ShortCode

    ClickedAt time.Time // from URLClickedEvent.OccurredAt (event time, not ingestion time)

    IPHash    string    // from URLClickedEvent.IPHash; never the raw IP

    UserAgent string    // from URLClickedEvent.UserAgent; empty string if absent

    Referer   string    // from URLClickedEvent.Referer; empty string means NULL in DB

}

// RefererCount is a single row returned by TopReferers.

type RefererCount struct {

    Referer string

    Count   int64

}

// PeriodCount is a single row returned by Timeline.

type PeriodCount struct {

    Period string // ISO 8601 formatted truncated timestamp; "2026-03-02T14:00:00Z" for hour, "2026-03-02" for day

    Clicks int64

}
```

```go
// repository/milestone_repository.go

package repository

import "time"

// Milestone records a click count threshold crossing for a short_code.

type Milestone struct {

    ID          string

    ShortCode   string

    Milestone   int       // 10 | 100 | 1000

    TriggeredAt time.Time

}
```

```go
// repository/processed_event_repository.go

package repository

import "time"

// ProcessedEvent is a deduplication record.

type ProcessedEvent struct {

    EventID     string    // UUID string; matches URLClickedEvent.EventID

    ProcessedAt time.Time

}
```

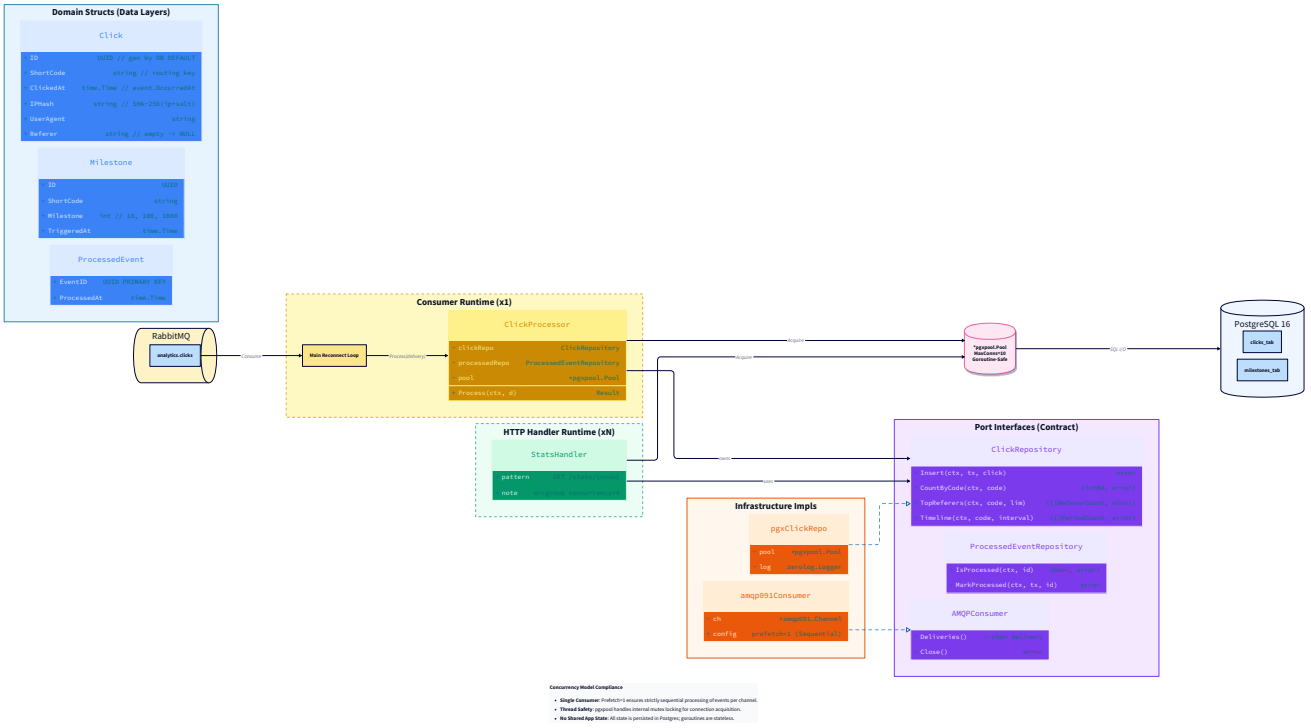
```go
// consumer/processor.go — internal processing types

// ProcessResult describes the outcome of processing one AMQP delivery.

type ProcessResult int

const (

    ProcessResultInserted    ProcessResult = iota // new event; click row inserted

    ProcessResultDuplicate                        // event_id already in processed_events; skipped

    ProcessResultPoisoned                         // malformed message; ACKed, not inserted

    ProcessResultError                            // transient DB/infra error; NACKed with requeue

)
```

GO

```go
// handler/ — API response types

// StatsResponse is the JSON body returned by GET /stats/:code.
type StatsResponse struct {
    ShortCode      string         `json:"short_code"`
    TotalClicks    int64          `json:"total_clicks"`
    ClicksLast24h  int64          `json:"clicks_last_24h"`
    ClicksLast7d   int64          `json:"clicks_last_7d"`
    TopReferers    []RefererItem  `json:"top_referers"`    // up to 5 entries; empty array if none
}

// RefererItem is a single entry in StatsResponse.TopReferers.
type RefererItem struct {
    Referer string `json:"referer"`
    Count   int64  `json:"count"`
}

// TimelineResponse is the JSON body returned by GET /stats/:code/timeline.
type TimelineResponse struct {
    ShortCode string          `json:"short_code"`
    Interval  string          `json:"interval"` // "day" | "hour"
    Points    []TimelinePoint `json:"points"`   // ordered oldest-first
}

// TimelinePoint is a single bucket in the timeline.
type TimelinePoint struct {
    Period string `json:"period"` // ISO 8601 string
    Clicks int64  `json:"clicks"`
}

// ErrorResponse is the JSON body for all 4xx/5xx responses.
type ErrorResponse struct {
    Error string `json:"error"`
}
```

Analytics Service Module Architecture

`sizedfpool` shared across goroutine boundaries — goroutine-safe by pgxpool design



Concurrency Model Compliance

- **Single Consumer**: Prefetch=1 ensures strictly sequential processing of events per channel.
- **Thread Safety**: pgxpool handles internal mutex locking for connection acquisition.
- **No Shared App State**: All state is persisted in Postgres; goroutines are stateless.

4. Interface Contracts

4.1 `ClickRepository`

```go
// repository/click_repository.go

// ClickRepository defines persistence operations for click records.

// Implementations must be safe for concurrent use.

type ClickRepository interface {

    // Insert writes a new click row within the provided transaction tx.

    // tx must be non-nil; Insert is always called inside the idempotency transaction.

    // click.Referer == "" is stored as NULL (the implementation converts empty → NULL).

    // Returns a wrapped error on DB failure; no sentinel errors defined.

    Insert(ctx context.Context, tx pgx.Tx, click Click) error

    // CountByCode returns the total number of clicks for shortCode.

    // Uses the pool (no transaction); reads committed data.

    // Returns 0, nil if no rows exist (not an error).

    CountByCode(ctx context.Context, shortCode string) (int64, error)

    // CountSince returns the number of clicks for shortCode with

    // clicked_at >= since (inclusive). Uses idx_clicks_short_code_time.

    // Returns 0, nil if no rows match.

    CountSince(ctx context.Context, shortCode string, since time.Time) (int64, error)

    // TopReferers returns up to limit referers with their click counts,

    // ordered by count DESC, for the given shortCode.

    // Only rows where referer IS NOT NULL are considered.

    // Returns an empty slice (not nil) if no referer rows exist.

    TopReferers(ctx context.Context, shortCode string, limit int) ([]RefererCount, error)

    // Timeline returns aggregated click counts bucketed by interval.

    // interval must be "day" or "hour" — callers validate before calling.

    // Returns rows ordered by period ASC (oldest first).

    // Uses PostgreSQL date_trunc(interval, clicked_at AT TIME ZONE 'UTC').

    // Returns an empty slice (not nil) if no rows exist.

    Timeline(ctx context.Context, shortCode string, interval string) ([]PeriodCount, error)

}
```

SQL queries:

```sql
-- Insert

-- Note: empty string referer → NULL via pgx nullability handling (see implementation note below)

INSERT INTO clicks (short_code, clicked_at, ip_hash, user_agent, referer)

VALUES ($1, $2, $3, $4, $5)

-- CountByCode

SELECT COUNT(*) FROM clicks WHERE short_code = $1

-- CountSince

SELECT COUNT(*) FROM clicks

WHERE short_code = $1 AND clicked_at >= $2

-- TopReferers

SELECT referer, COUNT(*) AS cnt

FROM clicks

WHERE short_code = $1

  AND referer IS NOT NULL

GROUP BY referer

ORDER BY cnt DESC

LIMIT $2

-- Timeline (interval validated to "day" or "hour" before interpolation)

-- SAFE: interval is never raw user input in the SQL string; it is validated and

-- only one of two literals is ever used. Use a CASE or two separate queries.

SELECT

    date_trunc($2, clicked_at AT TIME ZONE 'UTC') AS period,

    COUNT(*) AS clicks

FROM clicks

WHERE short_code = $1

GROUP BY period

ORDER BY period ASC
```

Implementation note on NULL referer:

```go
// pgx handles Go's typed nil via pgx.NullString or *string.

// Use *string for the referer column: nil → NULL in DB.

var referer *string

if click.Referer != "" {

    referer = &click.Referer

}

_, err = tx.Exec(ctx,

    `INSERT INTO clicks (short_code, clicked_at, ip_hash, user_agent, referer)

     VALUES ($1, $2, $3, $4, $5)`,

    click.ShortCode, click.ClickedAt, click.IPHash, click.UserAgent, referer,

)
```

`Timeline` SQL safety note: `date_trunc` takes the interval as a string literal, not a column reference. Since interval is validated to exactly `"day"` or `"hour"` before reaching the repository, it is safe to interpolate directly into the query string (not as a `$N` parameter — PostgreSQL does not accept `date_trunc($2, ...)` where `$2` is a string bind parameter for the granularity argument in all driver versions). Implementation:

```go
func (r *pgxClickRepository) Timeline(ctx context.Context, shortCode string, interval string) ([]PeriodCount, error) {
    // interval is pre-validated to "day" | "hour" by the handler.
    // Safe string interpolation: only two possible values.
    q := fmt.Sprintf(`
        SELECT date_trunc('%s', clicked_at AT TIME ZONE 'UTC') AS period, COUNT(*) AS clicks
        FROM clicks
        WHERE short_code = $1
        GROUP BY period
        ORDER BY period ASC`, interval)
    rows, err := r.pool.Query(ctx, q, shortCode)
    // ...scan rows into []PeriodCount...
}
```

4.2 `ProcessedEventRepository`

```go
// repository/processed_event_repository.go
// ProcessedEventRepository manages the deduplication table.
type ProcessedEventRepository interface {
    // IsProcessed checks whether eventID is already in processed_events.
    // Returns true if found, false if not found.
    // Returns false on DB error (logged internally); the processor will
    // attempt the INSERT and rely on the UNIQUE constraint as a safety net.
    // This is a read-only operation; uses the pool (no transaction).
    IsProcessed(ctx context.Context, eventID string) (bool, error)
    // MarkProcessed inserts eventID into processed_events within tx.
    // tx must be the same transaction as the Click INSERT — atomicity is the contract.
    // Returns error on DB failure; the constraint violation (duplicate) on
    // processed_events.event_id PRIMARY KEY is treated as a success (idempotent).
    MarkProcessed(ctx context.Context, tx pgx.Tx, eventID string) error
}
```

SQL queries:

```sql
-- IsProcessed
SELECT 1 FROM processed_events WHERE event_id = $1
-- MarkProcessed
INSERT INTO processed_events (event_id) VALUES ($1)
ON CONFLICT (event_id) DO NOTHING
```

Why `ON CONFLICT DO NOTHING` on `MarkProcessed`: The outer idempotency check (`IsProcessed`) prevents most re-processing. However, between the `IsProcessed` read and the transaction commit, a concurrent consumer (future scenario) could process the same event. The `ON CONFLICT DO NOTHING` makes `MarkProcessed` idempotent at the DB level as a second guard. In the current single-consumer design this is defensive; it costs nothing.

4.3 `MilestoneRepository`

```go
// repository/milestone_repository.go

// MilestoneRepository manages milestone threshold records.
type MilestoneRepository interface {
    // GetLatestMilestone returns the highest milestone value already recorded
    // for shortCode. Returns 0, nil if no milestone row exists.
    // Uses the pool (no transaction); reads committed data.
    GetLatestMilestone(ctx context.Context, shortCode string) (int, error)

    // InsertMilestone records a new milestone crossing within tx.
    // tx must be the same transaction as the click insert (§ 5.2).
    // On UNIQUE constraint violation (short_code, milestone already recorded):
    // returns nil (treated as no-op — milestone was concurrently recorded).
    // On any other DB error: returns wrapped error.
    InsertMilestone(ctx context.Context, tx pgx.Tx, shortCode string, milestone int) error
}
```

SQL queries:

```sql
-- GetLatestMilestone
SELECT COALESCE(MAX(milestone), 0)
FROM milestones
WHERE short_code = $1

-- InsertMilestone
INSERT INTO milestones (short_code, milestone)
VALUES ($1, $2)
ON CONFLICT (short_code, milestone) DO NOTHING
```

4.4 `AMQPConsumer`

```go
// consumer/consumer.go

// AMQPConsumer wraps the amqp091 channel consumption setup.

// It declares the queue and binding, then starts consuming.

type AMQPConsumer interface {

    // Deliveries returns the channel of incoming AMQP messages.

    // The channel is closed when the AMQP connection drops.

    // Callers must re-initialize AMQPConsumer on channel close.

    Deliveries() <-chan amqp091.Delivery

    // Close cancels the consumer tag and closes the AMQP channel.

    // Called during graceful shutdown or before reconnect.

    Close() error

}

// NewAMQPConsumer constructs a consumer, declaring the queue and binding,

// and starts the AMQP Basic.Consume. Returns error if any AMQP operation fails.

// queueName: "analytics.clicks"

// exchangeName: "url-shortener"

// routingKey: "url.clicked"

func NewAMQPConsumer(ch *amqp091.Channel, log zerolog.Logger) (AMQPConsumer, error)
```

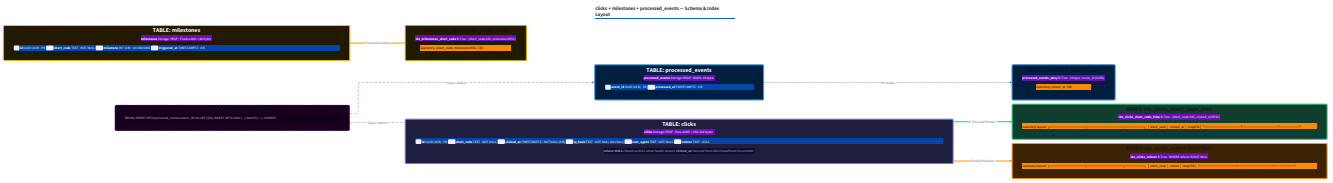
Consumer setup inside `NewAMQPConsumer`:

```go
// 1. Declare exchange (idempotent — url-service already declared it)

err = ch.ExchangeDeclare("url-shortener", "topic", true, false, false, false, nil)

// 2. Declare queue

q, err := ch.QueueDeclare("analytics.clicks", true, false, false, false, nil)

// 3. Bind queue to exchange

err = ch.QueueBind(q.Name, "url.clicked", "url-shortener", false, nil)

// 4. Set prefetch count: process one message at a time

err = ch.Qos(1, 0, false)  // prefetchCount=1, prefetchSize=0, global=false

// 5. Start consuming

deliveries, err := ch.Consume(

    q.Name,

    "analytics-consumer",  // consumer tag

    false,                 // autoAck=false — explicit ACK after processing

    false,                 // exclusive=false

    false,                 // noLocal=false

    false,                 // noWait=false

    nil,

)
```

Why `Qos(1, 0, false)` (prefetch=1): The consumer processes one message at a time to simplify idempotency logic. With `prefetch=1`, RabbitMQ delivers the next message only after the current one is ACKed or NACKed. This eliminates concurrent processing scenarios where two goroutines might race on `IsProcessed → INSERT`. A future optimization could increase prefetch and use a worker pool, but it would require distributed locking or serializable transactions for idempotency.

4.5 `ClickProcessor`

```go
// consumer/processor.go

// ClickProcessor orchestrates the full processing of a single URLClickedEvent.

// It is called by the consumer goroutine for each delivery.

type ClickProcessor struct {

    clickRepo     repository.ClickRepository

    processedRepo repository.ProcessedEventRepository

    milestoneRepo repository.MilestoneRepository

    detector      milestone.Detector

    publisher     publisher.AMQPPublisher

    pool          *pgxpool.Pool

    log           zerolog.Logger

}

func NewClickProcessor(

    pool *pgxpool.Pool,

    clickRepo repository.ClickRepository,

    processedRepo repository.ProcessedEventRepository,

    milestoneRepo repository.MilestoneRepository,

    detector milestone.Detector,

    pub publisher.AMQPPublisher,

    log zerolog.Logger,

) *ClickProcessor

// Process handles a single AMQP delivery.

// Returns ProcessResult indicating the outcome.

// The caller (consumer goroutine) uses the result to ACK or NACK.

// Process never panics — all panics are recovered internally and logged.

func (p *ClickProcessor) Process(ctx context.Context, d amqp091.Delivery) ProcessResult
```



4.6 `MilestoneDetector`

```go
// milestone/detector.go

// Thresholds is the ordered list of click counts at which milestones are triggered.

// Must be sorted ascending. Checked in order during detection.

var Thresholds = []int{10, 100, 1000}

// Detector determines which milestone (if any) was just crossed.

type Detector interface {

    // Detect returns the milestone value crossed by newTotal, given that

    // the highest previously recorded milestone is latestMilestone.

    // Returns 0 if no new milestone was crossed.

    // Example: latestMilestone=10, newTotal=100 → returns 100.

    // Example: latestMilestone=100, newTotal=101 → returns 0 (100 already recorded).

    // Example: latestMilestone=0, newTotal=10 → returns 10.

    // Example: latestMilestone=0, newTotal=15 → returns 10 (lowest uncrossed threshold).

    Detect(latestMilestone int, newTotal int64) int

}
```

`Detect` implementation:

```go
func (d *thresholdDetector) Detect(latestMilestone int, newTotal int64) int {

    for _, t := range Thresholds {

        if t > latestMilestone && int64(t) <= newTotal {

            return t // return the LOWEST uncrossed threshold that newTotal has reached

        }

    }

    return 0

}
```

Why return the lowest uncrossed threshold: If a code jumps from 0 to 150 clicks in a burst (e.g., due to consumer lag), `latestMilestone=0` and `newTotal=150`. The detector returns `10` (the first uncrossed threshold). After that milestone is recorded, `GetLatestMilestone` returns `10`, and the next call returns `100`. This ensures no threshold is skipped silently — each is recorded in sequence. In practice, with a running consumer this situation is rare.

4.7 `AMQPPublisher` (analytics-service scoped)

```go
// publisher/publisher.go

// AMQPPublisher publishes MilestoneReachedEvent to the RabbitMQ exchange.

// Interface is identical in structure to url-service's publisher (§ 4.5 of M3),

// but is independently defined in this package to avoid cross-service imports.

type AMQPPublisher interface {

    // Publish sends body to exchange "url-shortener" with routingKey "milestone.reached".

    // Returns error if the AMQP channel is closed or broker unreachable.

    // Callers log the error and continue — milestone publish failure does NOT

    // roll back the DB transaction (milestone row is still saved).

    Publish(ctx context.Context, routingKey string, body []byte) error

}

func NewAMQPPublisher(ch *amqp091.Channel, log zerolog.Logger) AMQPPublisher
```

5. Algorithm Specification

5.1 Consumer Goroutine Lifecycle

```
PROCEDURE RunConsumer(ctx context.Context, amqpURL string, processor *ClickProcessor, log):
  FOR:
    -- Attempt to connect and consume; reconnect on failure.
    conn, ch, err = connectAMQP(ctx, amqpURL)  -- exponential backoff, same as M1 § 4.5
    IF err:
      log.Warn().Err(err).Msg("consumer amqp connect failed; retrying in 10s")
      SELECT:
      CASE <-ctx.Done(): return
      CASE <-time.After(10 * time.Second): continue
    consumer, err = NewAMQPConsumer(ch, log)
    IF err:
      log.Warn().Err(err).Msg("consumer setup failed; retrying")
      ch.Close(); conn.Close()
      continue
    log.Info().Msg("amqp consumer started; waiting for messages")
    -- Inner loop: process deliveries until channel closes.
    FOR:
      SELECT:
      CASE <-ctx.Done():
        consumer.Close()
        conn.Close()
        return
      CASE d, ok := <-consumer.Deliveries():
        IF !ok:
          -- Channel closed (broker disconnect).
          log.Warn().Msg("amqp delivery channel closed; reconnecting")
          conn.Close()
          BREAK inner loop → reconnect
        -- Run processor in a supervised call (panic recovery inside Process).
        result = processor.Process(ctx, d)
        SWITCH result:
        CASE ProcessResultInserted:
          d.Ack(false)
        CASE ProcessResultDuplicate:
          d.Ack(false)      -- duplicate: safe to ACK, no DB side effect
        CASE ProcessResultPoisoned:
          d.Ack(false)      -- poison: ACK to remove from queue; logged as error
        CASE ProcessResultError:
          d.Nack(false, true) -- transient error: NACK with requeue=true
END PROCEDURE
```

Supervisor wrapper in `main.go`:

```go
go func() {
    for {
        func() {
            defer func() {
                if r := recover(); r != nil {
                    log.Error().Interface("panic", r).Msg("consumer goroutine panicked; restarting")
                }
            }()
            RunConsumer(ctx, amqpURL, processor, log)
        }()
        select {
        case <-ctx.Done():
            return
        case <-time.After(5 * time.Second):
            log.Info().Msg("restarting consumer goroutine")
        }
    }
}()
```

The outer `for` loop with `recover()` ensures the consumer goroutine is always running while the service is alive, even after a panic in the processor.

5.2 `ClickProcessor.Process` — Full Algorithm

```
PROCEDURE ClickProcessor.Process(ctx, d amqp091.Delivery) ProcessResult:
  -- Step 0: Panic recovery (defer at top of function)
  DEFER recover() → log error, return ProcessResultError
  -- Step 1: Parse event
  var event events.URLClickedEvent
  err = json.Unmarshal(d.Body, &event)
  IF err:
    log.Error().Err(err).
        Str("body", truncate(string(d.Body), 200)).
        Msg("malformed URLClickedEvent; ACKing poison message")
    return ProcessResultPoisoned
  -- Step 2: Validate required fields
  IF event.EventID == "" OR event.ShortCode == "":
    log.Error().
        Str("event_id", event.EventID).
        Str("short_code", event.ShortCode).
        Msg("URLClickedEvent missing required fields; ACKing")
    return ProcessResultPoisoned
  -- Step 3: Set correlation ID on logger for this message
  msgLog = log.With().
      Str("correlation_id", event.CorrelationID).
      Str("event_id", event.EventID).
      Str("short_code", event.ShortCode).
      Logger()
  -- Step 4: Fast idempotency pre-check (outside transaction for performance)
  already, err = processedRepo.IsProcessed(ctx, event.EventID)
  IF err:
    -- DB error on check: log, attempt processing anyway (DB constraint is the real guard)
    msgLog.Warn().Err(err).Msg("idempotency pre-check DB error; proceeding with insert attempt")
  IF already:
    msgLog.Debug().Msg("duplicate event; skipping")
    return ProcessResultDuplicate
  -- Step 5: Begin transaction
  tx, err = pool.Begin(ctx)
  IF err:
    msgLog.Error().Err(err).Msg("begin tx failed")
    return ProcessResultError
  DEFER tx.Rollback(ctx)  -- no-op if Commit succeeds
  -- Step 6: Mark processed (inside transaction, before click insert)
  -- Order: MarkProcessed FIRST, then Insert.
  -- Rationale: if Insert fails after MarkProcessed, the tx rolls back — both
  -- operations are undone. The event will be redelivered and reprocessed.
  -- If we inserted first and marked second, a crash between them would leave
  -- a click row with no idempotency record — the event is redelivered and
  -- double-counted. Inserting the mark first is the safe order.
  err = processedRepo.MarkProcessed(ctx, tx, event.EventID)
  IF err:
    msgLog.Error().Err(err).Msg("mark processed failed")
    return ProcessResultError
  -- Step 7: Build Click struct
  click = repository.Click{
      ShortCode: event.ShortCode,
      ClickedAt: event.OccurredAt,    -- event time, not now()
      IPHash:    event.IPHash,
      UserAgent: event.UserAgent,
      Referer:   event.Referer,       -- may be "" → stored as NULL
  }
  -- Step 8: Insert click
  err = clickRepo.Insert(ctx, tx, click)
  IF err:
    msgLog.Error().Err(err).Msg("click insert failed")
    return ProcessResultError
  -- Step 9: Commit click + idempotency mark
  err = tx.Commit(ctx)
  IF err:
    msgLog.Error().Err(err).Msg("commit failed")
    return ProcessResultError
  -- Step 10: Milestone detection (AFTER commit — reads committed data)
  -- This runs outside the click transaction to avoid long-held locks.
  p.detectAndRecordMilestone(ctx, event, msgLog)
  -- Step 11: Return success
  return ProcessResultInserted
END PROCEDURE
```

Why milestone detection is outside the click transaction: `CountByCode` reads the total click count for the short_code. If this query runs inside the click transaction, it reads the uncommitted inserted row (own-transaction visibility in PostgreSQL). However, running milestone detection inside the transaction means:

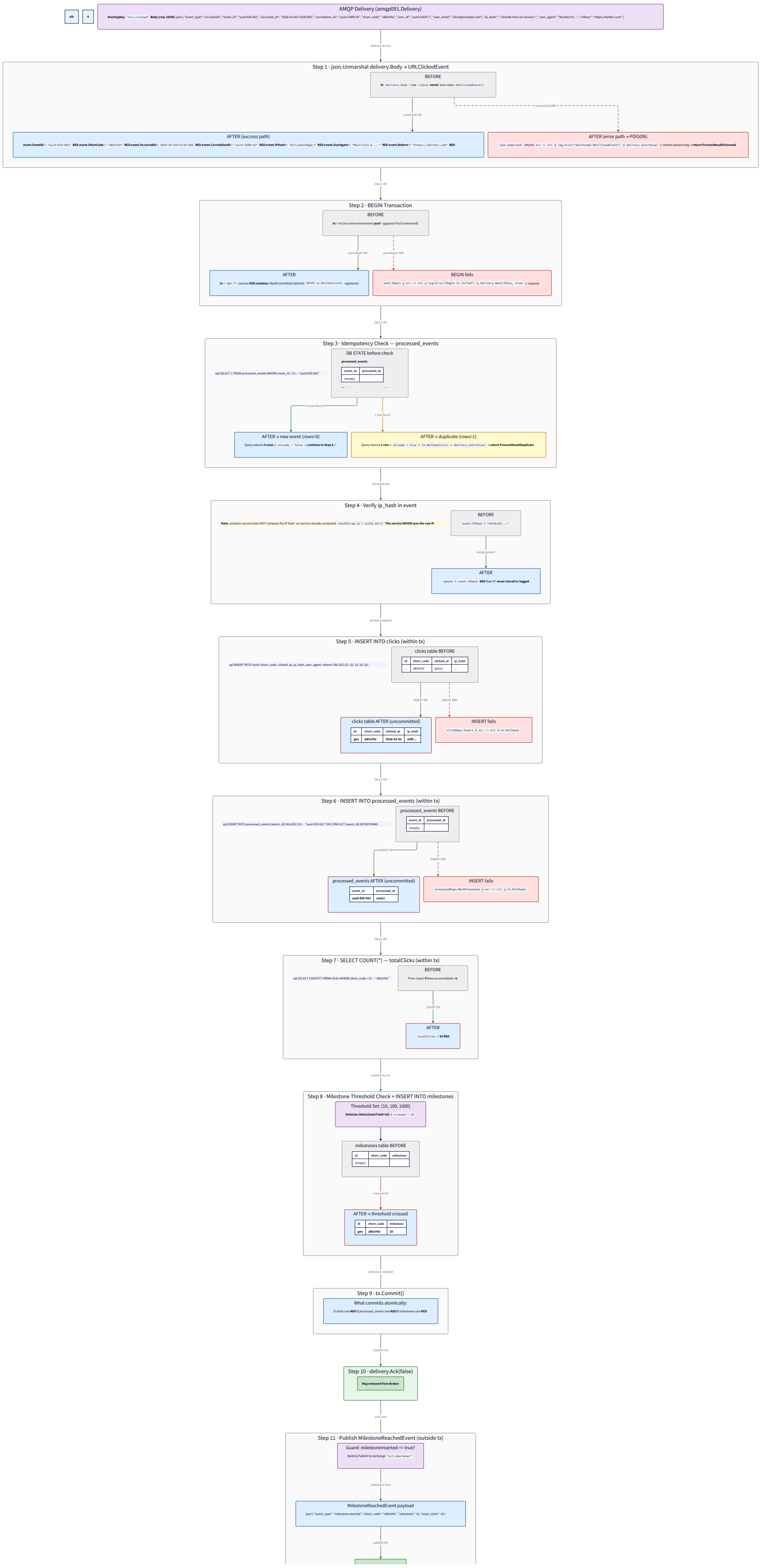
1. The transaction holds locks longer (milestone INSERT + possible publisher call).
2. A milestone publish failure would roll back the click insert. The spec (error category `MILESTONE_PUBLISH_FAIL`) explicitly states: "log warning, do not fail the transaction (milestone row still saved)." Therefore, milestone detection runs after the click transaction commits, with a fresh pool connection.

5.3 `detectAndRecordMilestone` — Milestone Detection Algorithm

```
PROCEDURE detectAndRecordMilestone(ctx, event URLClickedEvent, log):
  -- Step 1: Get current total click count (post-commit)
  total, err = clickRepo.CountByCode(ctx, event.ShortCode)
  IF err:
    log.Error().Err(err).Msg("milestone: count query failed")
    return  -- milestone detection skipped; not fatal
  -- Step 2: Get highest recorded milestone for this code
  latest, err = milestoneRepo.GetLatestMilestone(ctx, event.ShortCode)
  IF err:
    log.Error().Err(err).Msg("milestone: get latest failed")
    return
  -- Step 3: Detect if a threshold was crossed
  crossed = detector.Detect(latest, total)
  IF crossed == 0:
    return  -- no milestone crossed
  log.Info().
      Str("short_code", event.ShortCode).
      Int("milestone", crossed).
      Int64("total", total).
      Msg("milestone crossed")
  -- Step 4: Record milestone in its own transaction
  tx, err = pool.Begin(ctx)
  IF err:
    log.Error().Err(err).Msg("milestone: begin tx failed")
    return
  DEFER tx.Rollback(ctx)
  err = milestoneRepo.InsertMilestone(ctx, tx, event.ShortCode, crossed)
  IF err:
    log.Error().Err(err).Msg("milestone: insert failed")
    return  -- tx rollback via defer
  err = tx.Commit(ctx)
  IF err:
    log.Error().Err(err).Msg("milestone: commit failed")
    return
  -- Step 5: Publish MilestoneReachedEvent
  milestoneEvent = events.MilestoneReachedEvent{
      EventType:     events.EventTypeMilestoneReached,
      OccurredAt:    time.Now().UTC(),
      CorrelationID: event.CorrelationID,
      ShortCode:     event.ShortCode,
      UserID:        event.UserID,
      UserEmail:     event.UserEmail,  -- from URLClickedEvent (owner info)
      Milestone:     crossed,
      TotalClicks:   total,
  }
  payload, _ = json.Marshal(milestoneEvent)
  err = publisher.Publish(ctx, events.EventTypeMilestoneReached, payload)
  IF err:
    -- Log warning; milestone row already committed → notification-service will
    -- not be notified this cycle but the milestone is durably recorded.
    log.Warn().Err(err).
        Str("short_code", event.ShortCode).
        Int("milestone", crossed).
        Msg("milestone event publish failed; milestone saved but notification may be missed")
    return
  log.Info().Str("short_code", event.ShortCode).Int("milestone", crossed).
      Msg("MilestoneReachedEvent published")
END PROCEDURE
```

Milestone deduplication note: `milestones` has `UNIQUE (short_code, milestone)`. If the service crashes between Step 3 and Step 4 and the event is redelivered, `IsProcessed` returns true (the click was committed in Step 9 of `Process`), so the whole `Process` function returns `ProcessResultDuplicate` — milestone detection is not re-run. This is correct: the click count hasn't changed, so the milestone would be detected again. The `ON CONFLICT DO NOTHING` on `InsertMilestone` provides an additional safety net if the dedup table is somehow bypassed.

URLClickedEvent Ingestion — Idempotent Consumer Algorithm



5.4 `GET /stats/:code` Handler Algorithm

```
PROCEDURE HandleStats(w http.ResponseWriter, r *http.Request):
  shortCode = r.PathValue("code")
  IF shortCode == "":
    writeError(w, 400, "short code is required")
    return
  now = time.Now().UTC()
  -- Run all four queries concurrently using errgroup.
  -- All are read-only; no transaction needed.
  var (
      total   int64
      last24h int64
      last7d  int64
      refs    []repository.RefererCount
  )
  g, gctx = errgroup.WithContext(r.Context())
  g.Go(func() error {
      var err error
      total, err = clickRepo.CountByCode(gctx, shortCode)
      return err
  })
  g.Go(func() error {
      var err error
      last24h, err = clickRepo.CountSince(gctx, shortCode, now.Add(-24*time.Hour))
      return err
  })
  g.Go(func() error {
      var err error
      last7d, err = clickRepo.CountSince(gctx, shortCode, now.Add(-7*24*time.Hour))
      return err
  })
  g.Go(func() error {
      var err error
      refs, err = clickRepo.TopReferers(gctx, shortCode, 5)
      return err
  })
  IF err = g.Wait(); err != nil:
    log.Error().Err(err).Str("short_code", shortCode).Msg("stats query failed")
    writeError(w, 500, "internal server error")
    return
  -- Build response (always 200 even if total=0; short_code may have no clicks yet)
  items = make([]RefererItem, len(refs))
  FOR i, r := range refs:
    items[i] = RefererItem{Referer: r.Referer, Count: r.Count}
  writeJSON(w, 200, StatsResponse{
      ShortCode:     shortCode,
      TotalClicks:   total,
      ClicksLast24h: last24h,
      ClicksLast7d:  last7d,
      TopReferers:   items,  -- empty slice, never null
  })
END PROCEDURE
```

Concurrency note: Using `golang.org/x/sync/errgroup` to run the four queries concurrently reduces p99 latency from `4 × query_time` to approximately `1 × slowest_query_time`. All four queries are SELECT-only and use the shared pgxpool — each acquires its own connection. This is safe and the pool has `MaxConns=10` (M1), sufficient for 4 concurrent reads per request plus consumer activity.

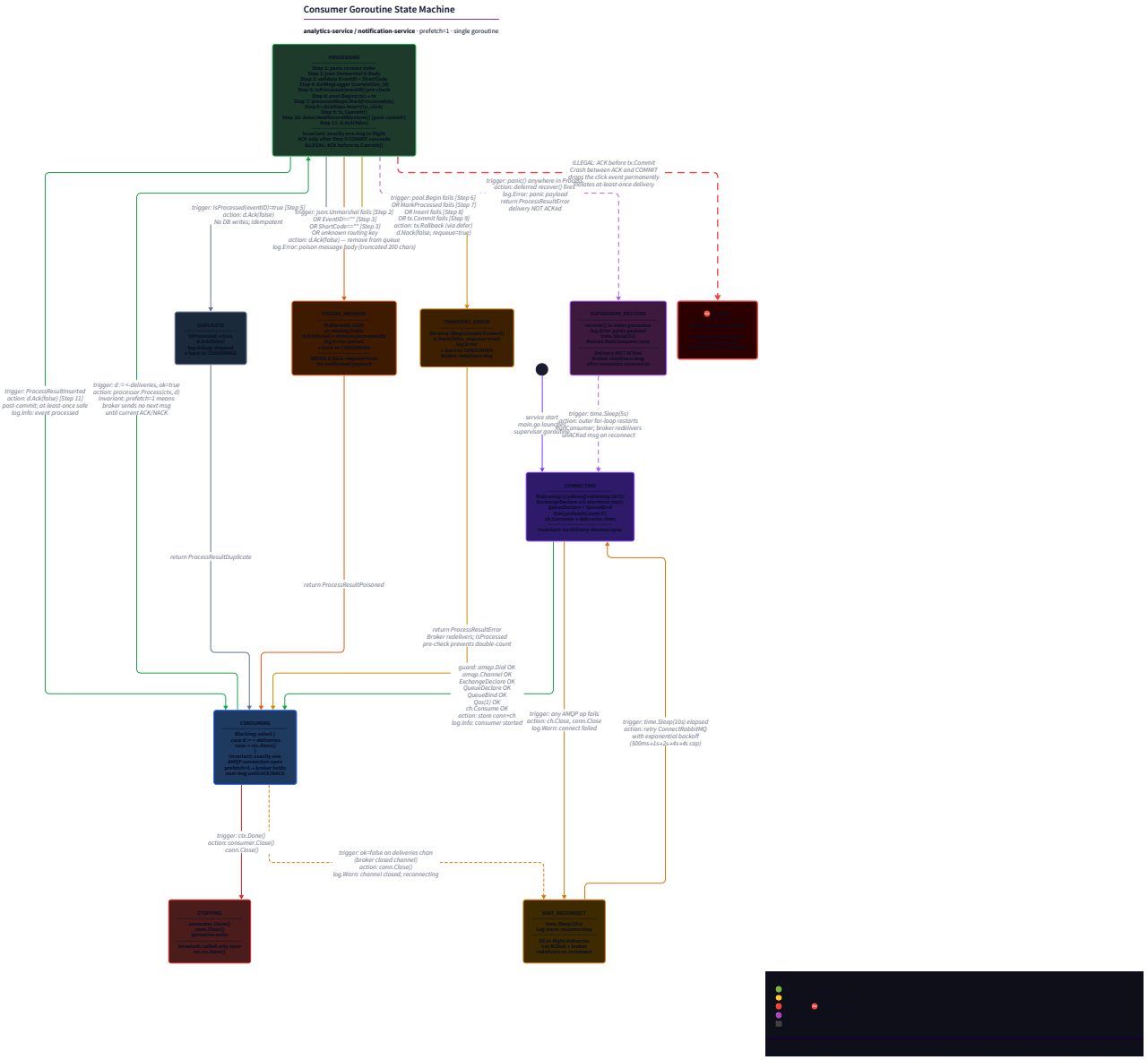
5.5 `GET /stats/:code/timeline` Handler Algorithm

```
PROCEDURE HandleTimeline(w http.ResponseWriter, r *http.Request):
  shortCode = r.PathValue("code")
  IF shortCode == "":
    writeError(w, 400, "short code is required")
    return
  interval = r.URL.Query().Get("interval")
  IF interval == "":
    interval = "day"  -- default
  IF interval != "day" AND interval != "hour":
    writeError(w, 400, "interval must be 'day' or 'hour'")
    return
  points, err = clickRepo.Timeline(r.Context(), shortCode, interval)
  IF err:
    log.Error().Err(err).Str("short_code", shortCode).Msg("timeline query failed")
    writeError(w, 500, "internal server error")
    return
  -- Format period timestamps as ISO 8601 strings.
  -- For interval="day": "2026-03-02" (date only).
  -- For interval="hour": "2026-03-02T14:00:00Z" (full RFC3339).
  formattedPoints = make([]TimelinePoint, len(points))
  FOR i, p := range points:
    formattedPoints[i] = TimelinePoint{
        Period: formatPeriod(p.Period, interval),
        Clicks: p.Clicks,
    }
  writeJSON(w, 200, TimelineResponse{
      ShortCode: shortCode,
      Interval:  interval,
      Points:    formattedPoints,
  })
END PROCEDURE
FUNCTION formatPeriod(period string, interval string) string:
  -- period is a UTC timestamp string returned by PostgreSQL date_trunc.
  -- Parse as time.Time, then format based on interval.
  t, _ = time.Parse(time.RFC3339, period)  -- pgx scans TIMESTAMPTZ as time.Time directly
  IF interval == "day":
    return t.UTC().Format("2006-01-02")
  return t.UTC().Format(time.RFC3339)
```

Implementation note: pgx scans `TIMESTAMPTZ` columns directly into `time.Time`. The `PeriodCount.Period` field should be `time.Time` (not `string`) in the repository layer; the handler converts to the appropriate string format. Update the `PeriodCount` struct:

```go
type PeriodCount struct {

    Period time.Time // scanned directly from PostgreSQL TIMESTAMPTZ

    Clicks int64

}
```

The `TimelinePoint.Period` in the API response is a string (formatted by the handler).



6. State Machine: Consumer Goroutine

```
States: DISCONNECTED → CONNECTING → CONSUMING → DISCONNECTED
        CONSUMING → STOPPING (on ctx.Done)
        STOPPING → (goroutine exits)
Transitions:
  DISCONNECTED →[connect attempt success]→ CONSUMING
  DISCONNECTED →[connect attempt fail]→ DISCONNECTED (wait 10s, retry)
  CONSUMING →[delivery channel closed]→ DISCONNECTED
  CONSUMING →[ctx.Done]→ STOPPING
  STOPPING →[consumer.Close + conn.Close]→ (exit)
Illegal transitions (must never occur):
  CONSUMING → CONSUMING (no nested consumer start)
  STOPPING → CONNECTING (no reconnect after shutdown signal)
```

```
States: CONSUMING → processing one message at a time
  Per-message substates:
    IDLE → PARSING → CHECKING_DUP → IN_TRANSACTION → COMMITTED → MILESTONE_DETECTION → IDLE
```

The `Qos(1, 0, false)` (prefetch=1) enforces that the broker does not deliver a second message until the first is

ACKed or NACKed, making the per-message substates truly sequential.

7. Error Handling Matrix

Error	Detected By	Recovery	ACK/NACK	Logged?	Notes
`MALFORMED_JSON`	`json.Unmarshal`	Log error, ACK	ACK	Error	Poison message; requeue would loop forever
`MISSING_REQUIRED_FIELDS`	field check in `Process`	Log error, ACK	ACK	Error	EventID or ShortCode empty
`DUPLICATE_EVENT`	`processedRepo.IsProcessed` → true	Log debug, ACK	ACK	Debug	At-least-once guarantee; normal occurrence
`IDEMPOTENCY_CHECK_DB_ERROR`	`IsProcessed` returns error	Log warn, proceed to INSERT	—	Warn	DB constraint is the real guard
`BEGIN_TX_FAIL`	`pool.Begin`	Log error, NACK requeue	NACK	Error	Transient; retried on redelivery
`MARK_PROCESSED_FAIL`	`processedRepo.MarkProcessed`	Log error, rollback, NACK	NACK	Error	
`CLICK_INSERT_FAIL`	`clickRepo.Insert`	Log error, rollback, NACK	NACK	Error	
`TX_COMMIT_FAIL`	`tx.Commit`	Log error, NACK	NACK	Error	Row not inserted; safe to retry
`MILESTONE_COUNT_FAIL`	`clickRepo.CountByCode`	Log error, skip milestone, return	ACK (click already committed)	Error	Click inserted; milestone detection skipped
`MILESTONE_INSERT_FAIL`	`milestoneRepo.InsertMilestone`	Log error, skip publish, return	ACK	Error	Click inserted; milestone not recorded this cycle
`MILESTONE_PUBLISH_FAIL`	`publisher.Publish`	Log warn, return	ACK	Warn	Milestone saved; notification may be missed
`CONSUMER_PANIC`	`recover()` in supervisor	Log error, restart goroutine after 5s	N/A	Error	Panic in Process means delivery not ACKed; broker redelivers after consumer reconnects
`AMQP_DISCONNECTED`	Delivery channel closed	Log warn, reconnect loop	N/A	Warn	Consumer pauses; /health still 200
`STATS_QUERY_FAIL`	`errgroup.Wait` error	Log error, 500 response	N/A	Error	
`TIMELINE_INVALID_INTERVAL`	handler validation	400 response	N/A	No	User input error

Error	Detected By	Recovery	ACK/NACK	Logged?	Notes
`DB_POOL_EXHAUSTED`	pgx `pool.Begin` or `pool.Query`	Same as `BEGIN_TX_FAIL`	NACK	Error	Indicates pool sizing issue
Dead-letter queue consideration: The spec does not require a DLQ in this milestone. Poison messages (malformed JSON) are ACKed to remove them from the queue. For production, a DLQ binding on `analytics.clicks` would capture nacked messages with `requeue=false`. Document in README as a future improvement.					

8. Concurrency Specification

8.1 Consumer Goroutine

Component	Goroutine Count	Shared Mutable State	Protection
Consumer supervisor	1	None	N/A
`RunConsumer`	1	`amqp091.Connection`, `amqp091.Channel`	Owned by single goroutine
`ClickProcessor.Process`	Called from 1 goroutine	`pgxpool.Pool` (goroutine-safe)	Pool internal locking
`detectAndRecordMilestone`	Called from 1 goroutine	Same pool	Pool internal locking
`AMQPPublisher`	Called from 1 goroutine	`amqp091.Channel`	No concurrent access in single-consumer design
Note: Because the consumer is single-goroutine (enforced by prefetch=1 + sequential processing), `AMQPPublisher` in analytics-service does NOT need the `sync.Mutex` that url-service's publisher uses (where 3 worker goroutines share one channel). If the design ever moves to concurrent processing, the mutex must be added.			

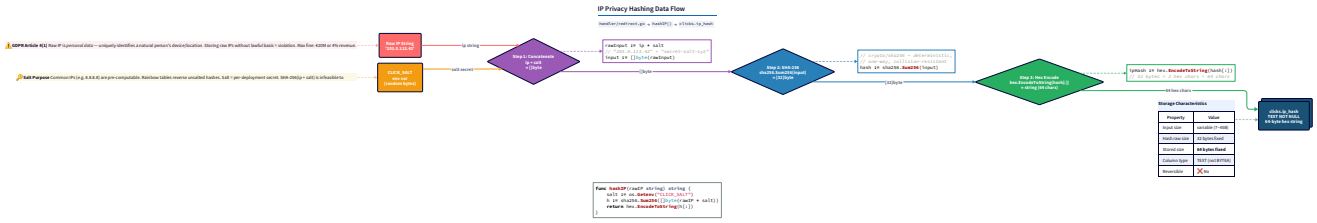
8.2 Stats API Handlers

Stats handlers run concurrently (one goroutine per request, via `net/http`). They are read-only; all state is in PostgreSQL. `pgxpool.Pool` is goroutine-safe. The `errgroup`-based concurrent queries within `HandleStats` are bounded to 4 goroutines per request — with `MaxConns=10`, up to 2 concurrent stat requests can run fully parallel before pool pressure begins.

8.3 Consumer vs. Stats API Isolation

The consumer goroutine and HTTP handler goroutines share the `*pgxpool.Pool`. Pool slots are acquired dynamically. With `MinConns=2, MaxConns=10`:

- Consumer always needs at most 2 connections simultaneously (one for the main click tx, one for milestone detection).
- Stats API handler needs up to 4 connections (errgroup concurrent queries).
- Total peak: 6 connections; well within `MaxConns=10`.



9. Route Registration in `main.go`

```go
// services/analytics-service/main.go (route wiring)

// No JWT auth on stats endpoints — stats are public per spec.

mux := http.NewServeMux()

mux.HandleFunc("GET /health",                    handler.NewHealthHandler("analytics-service", log))

mux.HandleFunc("GET /stats/{code}",              handler.NewStatsHandler(clickRepo, log))

mux.HandleFunc("GET /stats/{code}/timeline",     handler.NewTimelineHandler(clickRepo, log))
```

Pattern disambiguation: `GET /stats/{code}/timeline` is more specific than `GET /stats/{code}` — Go 1.22+ `net/http` routing correctly resolves this: `/stats/abc/timeline` matches the timeline handler, `/stats/abc` matches the stats handler. Register in any order; specificity takes precedence. No `/analytics/` prefix: The gateway (M5) routes `GET /api/stats/:code` → `http://analytics-service:8082/stats/:code`. The service itself uses `/stats/` without the `/api/` prefix — the gateway strips the path prefix before forwarding. Verify the gateway routing table in M5.

10. Implementation Sequence with Checkpoints

Phase 1 — Schema Migrations + Indexes (1–1.5 hr)

1. Write `migrations/001_create_clicks.sql`, `002_create_milestones.sql`, `003_create_processed_events.sql`.
2. Apply migrations:

```bash
docker exec -i analytics_db psql -U analytics_user -d analytics_db \
  < services/analytics-service/migrations/001_create_clicks.sql
docker exec -i analytics_db psql -U analytics_user -d analytics_db \
  < services/analytics-service/migrations/002_create_milestones.sql
docker exec -i analytics_db psql -U analytics_user -d analytics_db \
  < services/analytics-service/migrations/003_create_processed_events.sql
```

3. Verify schema and indexes:

```bash
docker exec analytics_db psql -U analytics_user -d analytics_db -c "\d clicks"
docker exec analytics_db psql -U analytics_user -d analytics_db -c "\di"
```

Checkpoint: Three tables exist. `\di` shows `idx_clicks_short_code_time`, `idx_clicks_referer` (partial), `idx_milestones_short_code`. `milestones` has unique constraint on `(short_code, milestone)`. `processed_events` has UUID primary key. Run:

```sql
EXPLAIN SELECT COUNT(*) FROM clicks WHERE short_code = 'abc' AND clicked_at >= now() - interval '24h';
```

Must show `Index Scan` on `idx_clicks_short_code_time`, not `Seq Scan`.

Phase 2 — Repository Layer (1.5–2 hr)

1. Add dependencies to `go.mod`: `go get github.com/jackc/pgx/v5 github.com/rs/zerolog golang.org/x/sync`.
2. Write `repository/click_repository.go`: `Click`, `RefererCount`, `PeriodCount`, `ClickRepository` interface, `pgxClickRepository`, all five methods.
3. Write `repository/processed_event_repository.go`: `ProcessedEvent`, `ProcessedEventRepository` interface, `pgxProcessedEventRepository`, `IsProcessed`, `MarkProcessed`.
4. Write `repository/milestone_repository.go`: `Milestone`, `MilestoneRepository` interface, `pgxMilestoneRepository`, `GetLatestMilestone`, `InsertMilestone`.
5. Write all three `_test.go` files with integration build tag.

6. Run: `go test -tags integration ./repository/... -v` **Checkpoint:**

- `TestClickInsert_HappyPath`: inserts click, `CountByCode` returns 1.
- `TestMarkProcessed_Idempotent`: `MarkProcessed` twice with same event_id → no error.
- `TestIsProcessed_AfterMark`: `IsProcessed` returns true after `MarkProcessed` + commit.
- `TestGetLatestMilestone_NoRows`: returns 0.
- `TestInsertMilestone_Conflict`: second insert of same `(short_code, milestone)` → returns nil (ON CONFLICT DO NOTHING).
- `TestTopReferers_NullsExcluded`: click with empty referer not returned in top referers.
- `TestTimeline_DayBucket`: two clicks on same day return one period row.

Phase 3 — RabbitMQ Consumer + Idempotency (2–2.5 hr)

1. Write `consumer/consumer.go`: `AMQPConsumer` interface, `amqp091Consumer` struct, `NewAMQPConsumer` (exchange declare, queue declare, bind, Qos, Consume).
2. Write `consumer/processor.go`: `ClickProcessor` struct, `NewClickProcessor`, `Process` following § 5.2 exactly.
3. Write `consumer/processor_test.go`: unit tests with mock repositories (§ 11.3).
4. Write `main.go` startup: DB connect, construct repositories, construct processor, launch `RunConsumer` supervisor goroutine.
5. Run `docker compose up --build analytics-service`. **Checkpoint:**

```bash
# From url-service (M3), shorten a URL with a valid JWT and make a redirect:

TOKEN=$(curl -s -X POST http://localhost:8083/login \

  -H "Content-Type: application/json" \

  -d '{"email":"alice@example.com","password":"password123"}' | jq -r .token)

CODE=$(curl -s -X POST http://localhost:8081/shorten \

  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \

  -d '{"url":"https://www.example.com"}' | jq -r .short_code)

curl -s "http://localhost:8081/$CODE" -L > /dev/null

# Wait for outbox poller (url-service) + consumer (analytics-service):

sleep 5

# Verify click was inserted:

docker exec analytics_db psql -U analytics_user -d analytics_db \

  -c "SELECT short_code, clicked_at FROM clicks ORDER BY clicked_at DESC LIMIT 1;"

# Verify processed_events has the event:

docker exec analytics_db psql -U analytics_user -d analytics_db \

  -c "SELECT event_id, processed_at FROM processed_events ORDER BY processed_at DESC LIMIT 1;"
```

Phase 4 — Milestone Detection + MilestoneReachedEvent Publish (1–1.5 hr)

1. Write `milestone/detector.go`: `Thresholds`, `Detector` interface, `thresholdDetector`, `Detect`.
2. Write `milestone/detector_test.go` (§ 11.4).
3. Write `publisher/publisher.go`: `AMQPPublisher` interface, `amqp091Publisher`.
4. Update `consumer/processor.go`: inject `Detector` and `AMQPPublisher`; implement `detectAndRecordMilestone` (§ 5.3).
5. Run `go test ./milestone/... -v`. **Checkpoint:**

```bash
# Generate 10 redirect events for the same short_code to cross the first milestone.
for i in {1..10}; do curl -s "http://localhost:8081/$CODE" -L > /dev/null; done
sleep 6
# Verify milestone row:
docker exec analytics_db psql -U analytics_user -d analytics_db \
  -c "SELECT short_code, milestone, triggered_at FROM milestones;"
# Expected: 1 row with milestone=10
# Verify MilestoneReachedEvent in RabbitMQ management UI:
# http://localhost:15672 → Queues → notifications.events → Get Messages
# Should show event with event_type="milestone.reached", milestone=10
```

Phase 5 — Stats API Handlers (1.5–2 hr)

1. Write `handler/helpers.go`: `writeJSON`, `writeError`.
2. Write `handler/stats.go`: `NewStatsHandler` with errgroup concurrent queries (§ 5.4).
3. Write `handler/timeline.go`: `NewTimelineHandler` (§ 5.5).
4. Write `handler/health.go`: `NewHealthHandler("analytics-service", log)`.
5. Wire routes in `main.go`.
6. Write `handler/handler_test.go` (§ 11.5).
7. Run `docker compose up --build analytics-service`. **Checkpoint:**

```bash
curl -s "http://localhost:8082/stats/$CODE" | jq .
# Expected:
# {
#   "short_code": "<CODE>",
#   "total_clicks": 10,
#   "clicks_last_24h": 10,
#   "clicks_last_7d": 10,
#   "top_referers": []
# }
curl -s "http://localhost:8082/stats/$CODE/timeline?interval=hour" | jq .
# Expected: {"short_code":"...", "interval":"hour", "points":[{"period":"2026-...T...Z","clicks":10}]}
curl -s "http://localhost:8082/stats/$CODE/timeline?interval=invalid"
# Expected: 400 {"error":"interval must be 'day' or 'hour'"}
curl -s "http://localhost:8082/health" | jq .
# Expected: {"status":"ok","service":"analytics-service"}
```

Phase 6 — Idempotency Test (0.5–1 hr)

1. Write `consumer/processor_test.go` integration test for duplicate delivery (§ 11.3 `TestDuplicateEvent_ClickCount1`).
2. Run with live stack: `go test -tags integration ./consumer/... -v -run TestDuplicateEvent`. **Checkpoint:**

```bash
# Direct integration test (described in § 11.3):

# Deliver the same URLClickedEvent JSON twice to analytics.clicks queue.

# After both are processed, clicks table has exactly 1 row for that event_id.

go test -tags integration -v -run TestDuplicateEvent ./consumer/...

# Expected: PASS
```

11. Test Specification

11.1 `repository/click_repository_test.go` (integration)

```go
// Build tag: //go:build integration

// Requires: analytics_db running. DSN from TEST_DATABASE_DSN env var.

// TestClickInsert_HappyPath: Insert click in tx, commit, CountByCode returns 1.

func TestClickInsert_HappyPath(t *testing.T)

// TestClickInsert_NullReferer: Insert with empty referer → NULL in DB.

// Verify: SELECT referer FROM clicks WHERE ... IS NULL.

func TestClickInsert_NullReferer(t *testing.T)

// TestClickInsert_NonNullReferer: Insert with non-empty referer → stored correctly.

func TestClickInsert_NonNullReferer(t *testing.T)

// TestCountSince_24h: Insert 3 clicks: 2 within 24h, 1 older. CountSince(24h) returns 2.

func TestCountSince_24h(t *testing.T)

// TestTopReferers_TopFive: Insert clicks with 6 distinct referers.

// TopReferers(code, 5) returns 5 entries, highest count first.

func TestTopReferers_TopFive(t *testing.T)

// TestTopReferers_NullsExcluded: Insert 3 clicks with null referer, 1 with referer.

// TopReferers returns only 1 entry.

func TestTopReferers_NullsExcluded(t *testing.T)

// TestTimeline_DayInterval: Insert 3 clicks on same day, 2 on next day.

// Timeline(code, "day") returns 2 period rows with correct counts.

func TestTimeline_DayInterval(t *testing.T)

// TestTimeline_HourInterval: Insert 2 clicks in hour 14, 1 in hour 15.

// Timeline(code, "hour") returns 2 rows.

func TestTimeline_HourInterval(t *testing.T)

// TestTimeline_EmptyResult: Timeline for unknown code returns empty slice, not nil.

func TestTimeline_EmptyResult(t *testing.T)

// TestCountByCode_NoRows: CountByCode for unknown code returns 0, nil (not error).

func TestCountByCode_NoRows(t *testing.T)
```

11.2 `repository/processed_event_repository_test.go` (integration)

```go
// TestIsProcessed_NotFound: unknown event_id returns false, nil.

func TestIsProcessed_NotFound(t *testing.T)

// TestIsProcessed_AfterMark: MarkProcessed in tx, commit, IsProcessed returns true.

func TestIsProcessed_AfterMark(t *testing.T)

// TestMarkProcessed_Idempotent: MarkProcessed called twice with same event_id.

// Second call returns nil (ON CONFLICT DO NOTHING).

func TestMarkProcessed_Idempotent(t *testing.T)

// TestMarkProcessed_RollsBackWithTx: MarkProcessed in tx, rollback → IsProcessed returns false.

func TestMarkProcessed_RollsBackWithTx(t *testing.T)
```

11.3 `consumer/processor_test.go` (unit + integration)

Unit tests use mock implementations of `ClickRepository`, `ProcessedEventRepository`, `MilestoneRepository`, `Detector`, `AMQPPublisher`. Define interfaces in their respective packages so mocks implement them.

```go
// Unit tests (no build tag; mock dependencies):

// TestProcess_HappyPath: valid URLClickedEvent JSON → ProcessResultInserted returned.

// Mock: IsProcessed=false, Insert=nil, MarkProcessed=nil, CountByCode=1, GetLatestMilestone=0, Detect=0.

func TestProcess_HappyPath(t *testing.T)

// TestProcess_MalformedJSON: invalid JSON body → ProcessResultPoisoned.

// Verify: no DB calls made.

func TestProcess_MalformedJSON(t *testing.T)

// TestProcess_EmptyEventID: EventID="" → ProcessResultPoisoned.

func TestProcess_EmptyEventID(t *testing.T)

// TestProcess_DuplicateEvent: IsProcessed=true → ProcessResultDuplicate.

// Verify: Insert NOT called.

func TestProcess_DuplicateEvent(t *testing.T)

// TestProcess_InsertFails: Insert returns error → ProcessResultError.

// Verify: MarkProcessed not committed.

func TestProcess_InsertFails(t *testing.T)

// TestProcess_MilestoneDetected: Detect returns 10 → InsertMilestone and Publish called.

func TestProcess_MilestoneDetected(t *testing.T)

// TestProcess_MilestonePublishFails: Publish returns error → ProcessResultInserted (not error).

// Click was committed; milestone saved; publish failure is non-fatal.

func TestProcess_MilestonePublishFails(t *testing.T)

// TestProcess_PanicRecovery: mock ClickRepo.Insert panics → Process returns ProcessResultError.

// Verify: no panic propagates to caller.

func TestProcess_PanicRecovery(t *testing.T)
```

```go
// Integration test (//go:build integration):

// Requires full stack running: analytics_db, rabbitmq.

// TestDuplicateEvent_ClickCount1:

// 1. Construct a URLClickedEvent with a fixed EventID UUID.

// 2. Marshal to JSON.

// 3. Directly call processor.Process twice with identical amqp091.Delivery.

// 4. Assert: first call returns ProcessResultInserted.

// 5. Assert: second call returns ProcessResultDuplicate.

// 6. Assert: SELECT COUNT(*) FROM clicks WHERE ip_hash = event.IPHash → 1 (not 2).

// 7. Assert: SELECT COUNT(*) FROM processed_events WHERE event_id = eventID → 1.

func TestDuplicateEvent_ClickCount1(t *testing.T)
```

11.4 `milestone/detector_test.go`

```go
// TestDetect_NoThresholdCrossed: latestMilestone=0, newTotal=5 → returns 0.

func TestDetect_NoThresholdCrossed(t *testing.T)

// TestDetect_FirstThreshold: latestMilestone=0, newTotal=10 → returns 10.

func TestDetect_FirstThreshold(t *testing.T)

// TestDetect_FirstThresholdExceeded: latestMilestone=0, newTotal=15 → returns 10.

// (lowest uncrossed threshold, not the exact count)

func TestDetect_FirstThresholdExceeded(t *testing.T)

// TestDetect_AlreadyAtThreshold: latestMilestone=10, newTotal=10 → returns 0.

// (10 is already recorded)

func TestDetect_AlreadyAtThreshold(t *testing.T)

// TestDetect_SkipsToHigher: latestMilestone=10, newTotal=150 → returns 100.

func TestDetect_SkipsToHigher(t *testing.T)

// TestDetect_AllThresholdsCrossed: latestMilestone=0, newTotal=1500 → returns 10.

// Only the lowest uncrossed is returned per call.

func TestDetect_AllThresholdsCrossedReturnsLowest(t *testing.T)

// TestDetect_BeyondAllThresholds: latestMilestone=1000, newTotal=50000 → returns 0.

func TestDetect_BeyondAllThresholds(t *testing.T)

// Table-driven test covering all threshold transitions:

func TestDetect_AllTransitions(t *testing.T) // uses t.Run subtests
```

11.5 `handler/handler_test.go`

```go
// All tests use httptest.NewRecorder + httptest.NewRequest.

// Mock ClickRepository injected via constructor.

// TestStats_HappyPath: mock returns total=42, last24h=5, last7d=20, refs=[].

// GET /stats/abc → 200, JSON fields match.

func TestStats_HappyPath(t *testing.T)

// TestStats_WithReferers: mock TopReferers returns 3 entries.

// Response top_referers has 3 items in count-DESC order.

func TestStats_WithReferers(t *testing.T)

// TestStats_ZeroClicks: all counts return 0. Response is valid JSON with 0s, not null.

func TestStats_ZeroClicks(t *testing.T)

// TestStats_DBError: mock CountByCode returns error → 500.

func TestStats_DBError(t *testing.T)

// TestStats_TopReferers_NeverNull: even with no referers, top_referers is [] not null.

func TestStats_TopReferers_NeverNull(t *testing.T)

// TestTimeline_DayInterval: mock returns 3 period rows.

// Response points formatted as "2026-03-02" strings.

func TestTimeline_DayInterval(t *testing.T)

// TestTimeline_HourInterval: mock returns 2 period rows.

// Response points formatted as RFC3339 strings.

func TestTimeline_HourInterval(t *testing.T)

// TestTimeline_InvalidInterval: interval="week" → 400.

func TestTimeline_InvalidInterval(t *testing.T)

// TestTimeline_MissingInterval: no ?interval param → defaults to "day", 200.

func TestTimeline_MissingInterval(t *testing.T)

// TestTimeline_EmptyPoints: mock returns empty slice → points:[] not null.

func TestTimeline_EmptyPoints(t *testing.T)

// TestHealth_Returns200: GET /health → 200 {"status":"ok","service":"analytics-service"}.

func TestHealth_Returns200(t *testing.T)

// BenchmarkStats: mock-backed, measure p99 of /stats/:code handler.

// Target: < 30ms with real DB; < 1ms with mocks.

func BenchmarkStats(b *testing.B)

// BenchmarkTimeline: mock-backed, measure p99 of /stats/:code/timeline handler.

func BenchmarkTimeline(b *testing.B)
```

12. Performance Targets

Operation	Target	How to Measure
Click ingestion lag (publish → DB insert)	< 500ms p99	Timestamp `URLClickedEvent.OccurredAt` vs `clicks.clicked_at` insert time; measure with 100 sequential clicks
`GET /stats/:code` p99 (live DB)	< 30ms	`wrk -t4 -c20 -d30s http://localhost:8082/stats/<code>` with 1000+ click rows
`GET /stats/:code/timeline?interval=day` p99	< 50ms	`wrk -t2 -c10 -d30s` with 30 days of data
`IsProcessed` (indexed SELECT by UUID PK)	< 5ms	`go test -bench=BenchmarkIsProcessed ./repository/...`
`CountByCode` (index scan)	< 10ms	`go test -bench=BenchmarkCountByCode ./repository/...` with 10,000 click rows
`Timeline` aggregation (1000 rows, day bucket)	< 20ms	`EXPLAIN ANALYZE` on timeline query; benchmark at scale
`GET /health` p99	< 10ms	(inherited from M1)
`go test ./...` (unit, no DB)	< 10s	CI warm build
`docker build` image size	< 20MB	`docker image ls analytics-service`
Milestone detection + publish end-to-end	< 2s from 10th click commit	Observe `triggered_at` in milestones vs click timestamps
Errgroup concurrency speedup for /stats	4x vs sequential	Benchmark with 10ms mock query delay × 4 queries
Index validation commands:		

```bash
# Confirm index scan on clicks_short_code_time for CountByCode

docker exec analytics_db psql -U analytics_user -d analytics_db -c \

  "EXPLAIN ANALYZE SELECT COUNT(*) FROM clicks WHERE short_code = 'abc' AND clicked_at >= now() - interval '24h';"

# Must show: "Index Scan using idx_clicks_short_code_time"

# Confirm partial index for TopReferers

docker exec analytics_db psql -U analytics_user -d analytics_db -c \

  "EXPLAIN ANALYZE SELECT referer, COUNT(*) FROM clicks WHERE short_code='abc' AND referer IS NOT NULL GROUP BY referer ORDER BY 2 DESC
LIMIT 5;"

# Must show: "Index Scan using idx_clicks_referer"
```

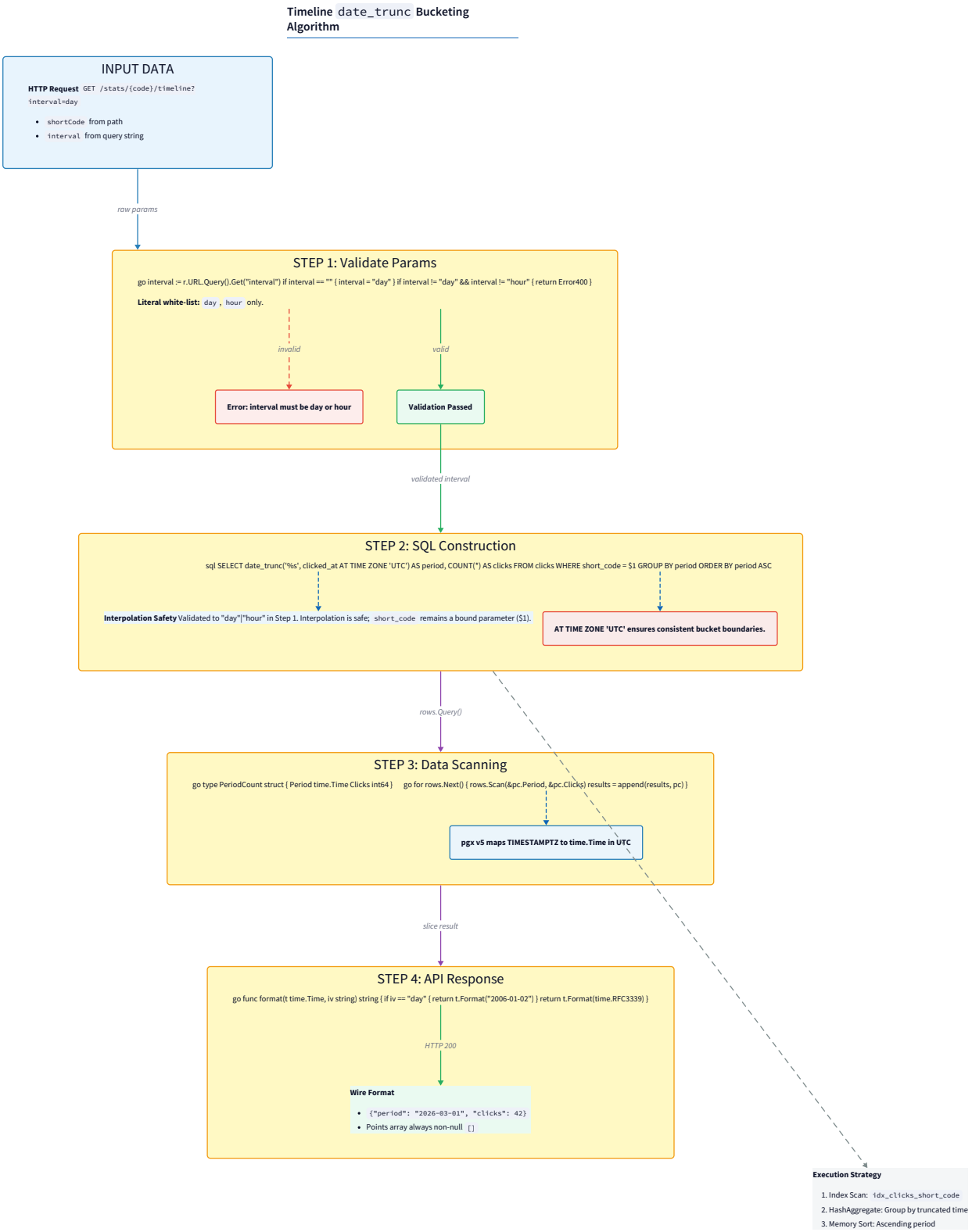
13. Environment Variable Reference

Variable	Required	Example	Description
`PORT`	Yes	`8082`	HTTP listen port
`SERVICE_NAME`	Yes	`analytics-service`	Embedded in log lines
`DATABASE_DSN`	Yes	`postgres://analytics_user:analytics_secret@analytics_db:5432/analytics_db?sslmode=disable`	analytics_db connection
`RABBITMQ_URL`	Yes	`amqp://admin:admin@rabbitmq:5672/`	AMQP broker
No `JWT_SECRET` for this milestone — stats endpoints are public. `JWT_SECRET` is added in M5 if the gateway requires it for proxied requests (gateway validates; analytics-service itself does not validate tokens per spec).			
No `REDIS_ADDR` — analytics-service has no cache. It does not share Redis with url-service or the gateway.			

14. Anti-Pattern Guard Rail Checklist

Before completing this milestone, verify:

- [ ] **No cross-service DB access.** `grep -r "url_db\|user_db\|notification_db" services/analytics-service/` returns nothing.
- [ ] **No call to url-service on event ingestion.** `grep -r "url-service\|8081" services/analytics-service/` returns nothing except comments. Trust the event; do not validate `short_code` against the URL Service.
- [ ] **No raw IP storage.** `grep -r "RemoteAddr\|X-Forwarded-For\|ip_address" services/analytics-service/` returns nothing. Only `ip_hash` (pre-computed by url-service) is stored.
- [ ] **Idempotency implemented.** `processed_events` table exists with UUID PK. `MarkProcessed` is called inside the same transaction as `Insert`. `IsProcessed` is checked before the transaction begins.
- [ ] **Poison message ACKed, not NACKed.** `grep "Ack" services/analytics-service/consumer/` shows `ProcessResultPoisoned` maps to `Ack(false)`, not `Nack`.
- [ ] **`/health` returns 200 when RabbitMQ is down.** The health handler has no dependency on the AMQP connection. Consumer pauses; service stays up.
- [ ] **Milestone publish failure is non-fatal.** `detectAndRecordMilestone` logs a warning and returns on publish error. The click transaction is already committed before milestone detection runs.
- [ ] **`date_trunc` interval is validated before use.** Handler validates `interval ∈ {"day", "hour"}` before passing to `Timeline`. No raw user input in SQL.
- [ ] **`top_referers` is never null in response.** `StatsResponse.TopReferers` is initialized with `make([]RefererItem, 0)` when `refs` is empty — JSON encodes as `[]`, not `null`.
- [ ] **`timeline.points` is never null in response.** Same initialization pattern.



Technical Design Specification

Module: Notification Service + API Gateway + Circuit Breaker

Module ID: `url-shortener-m5`

1. Module Charter

This module delivers the final two infrastructure components that complete the URL shortener system. The **Notification Service** is a pure event consumer: it subscribes to `url.created`, `url.deleted`, and `milestone.reached` events from the `notifications.events` RabbitMQ queue, persists a notification record per event, logs a mock email send, and exposes a JWT-protected `GET /notifications` endpoint for paginated retrieval. The **API Gateway** is the sole client-facing entry point: it validates JWTs locally (no user-service call per request), propagates `X-Correlation-ID`, enforces per-IP token-bucket rate limits via Redis, implements a three-state circuit breaker for the url-service proxy path, and routes all requests to the correct downstream service without containing any domain business logic. The `shared/logger` package is finalized here and retrofitted into all services with structured JSON fields including `correlation_id`. This module does **not** implement email delivery (only log output), password reset, push notifications, WebSocket subscriptions, server-sent events, any form of aggregation or analytics within the gateway, load balancing across multiple replicas of a downstream service, OAuth, API keys, or mTLS. The gateway does not call user-service to verify tokens at request time — it uses the shared `JWT_SECRET` for local HMAC-SHA256 verification identically to every other service. The gateway does not inspect request bodies; it proxies them verbatim. **Upstream dependencies:** M1 (Docker Compose stack, `shared/events`, `shared/logger`, RabbitMQ, Redis), M2 (`shared/auth` JWTVerifier, `shared/middleware` RequireAuth), M3 (url-service on port 8081), M4 (analytics-service on port 8082). The `notifications.events` queue was declared in M1's startup but the consumer goroutine is first implemented here. **Downstream dependencies:** None. This is the terminal milestone. No subsequent service depends on notification-service or the gateway at the code level. **Invariants that must always hold after this milestone:**

- The gateway is the only service with a port exposed to clients in production; all downstream services communicate only on the Docker internal `backend` network.
- A JWT that fails HMAC-SHA256 verification at the gateway never reaches any downstream service.
- The circuit breaker for url-service transitions through exactly three states (Closed → Open → HalfOpen → Closed); a retry loop without state is not a circuit breaker.
- Rate limit counters for `/api/shorten` and `/r/:code` are stored in Redis (shared across hypothetical replicas), never in process-local memory.
- If Redis is unavailable when a rate-limit check is attempted, the request is **allowed** (fail-open) and a warning is logged; no traffic is blocked due to a Redis outage.
- Every notification event consumed from RabbitMQ results in exactly one `notifications` row and one log line, regardless of how many times the event is delivered.
- `GET /notifications` returns only notifications belonging to the authenticated user's `sub` claim; cross-user access is impossible.

2. File Structure

Create files in the numbered order below. All paths are relative to the monorepo root.

```
url-shortener/
│
├── services/notification-service/
│   ├── migrations/
│   │   └── 01  001_create_notifications.sql
│   ├── repository/
│   │   ├── 02  notification_repository.go
│   │   └── 03  notification_repository_test.go
│   ├── consumer/
│   │   ├── 04  consumer.go             # AMQPConsumer setup (reuse M4 pattern)
│   │   ├── 05  processor.go            # NotificationProcessor: parse + insert + log
│   │   └── 06  processor_test.go
│   ├── handler/
│   │   ├── 07  notifications.go        # GET /notifications (auth required)
│   │   ├── 08  health.go
│   │   ├── 09  helpers.go
│   │   └── 10  handler_test.go
│   └── 11  main.go
│
├── gateway/
│   ├── circuitbreaker/
│   │   ├── 12  circuitbreaker.go       # CircuitBreaker state machine
│   │   └── 13  circuitbreaker_test.go
│   ├── ratelimit/
│   │   ├── 14  ratelimit.go            # RateLimiter: Redis INCR + EXPIRE
│   │   └── 15  ratelimit_test.go
│   ├── proxy/
│   │   ├── 16  proxy.go                # ReverseProxy wrapper
│   │   └── 17  proxy_test.go
│   ├── middleware/
│   │   ├── 18  correlation.go          # X-Correlation-ID inject/propagate
│   │   ├── 19  jwt.go                  # Gateway JWT middleware (wraps shared/middleware)
│   │   ├── 20  logging.go             # Request/response structured logging middleware
│   │   └── 21  middleware_test.go
│   ├── handler/
│   │   ├── 22  health.go
│   │   └── 23  handler_test.go
│   ├── router/
│   │   ├── 24  router.go              # Route table + registration
│   │   └── 25  router_test.go
│   └── 26  main.go
│
└── shared/
    └── logger/
        └── 27  logger.go              # Finalized zerolog wrapper (update from M1 stub)
```

Total new files: 27 (excluding auto-generated `go.sum` changes and files already written in prior milestones).

3. Complete Data Model

3.1 PostgreSQL Schema — `migrations/001_create_notifications.sql`

```sql
-- migrations/001_create_notifications.sql

BEGIN;

CREATE TABLE IF NOT EXISTS notifications (

    id           UUID        PRIMARY KEY DEFAULT gen_random_uuid(),

    user_id      UUID        NOT NULL,          -- JWT sub of the URL owner

    event_type   TEXT        NOT NULL,          -- "url.created" | "url.deleted" | "milestone.reached"

    payload      JSONB       NOT NULL,          -- raw event struct for debugging / future use

    status       TEXT        NOT NULL DEFAULT 'sent',  -- 'pending' | 'sent' | 'failed'

    created_at   TIMESTAMPTZ NOT NULL DEFAULT now(),

    sent_at      TIMESTAMPTZ NULL               -- set when mock send completes (= created_at in M5)

);

-- Primary list query: user's notifications, newest first, cursor-paginated.

CREATE INDEX IF NOT EXISTS idx_notifications_user_id_created

    ON notifications(user_id, created_at DESC);

-- Deduplication: each event_id is processed at most once.

-- Stores the RabbitMQ message's correlation_id + event_type as dedup key.

CREATE TABLE IF NOT EXISTS processed_notifications (

    event_key    TEXT        PRIMARY KEY,   -- "<event_type>:<correlation_id>" composite key

    processed_at TIMESTAMPTZ NOT NULL DEFAULT now()

);

COMMIT;
```

Field rationale:

- `user_id UUID NOT NULL`: Extracted from `URLCreatedEvent.UserID`, `URLDeletedEvent.UserID`, or `MilestoneReachedEvent.UserID`. This is what makes `GET /notifications` user-scoped. Stored as UUID even though the column type is UUID — validated during event parse.
- `event_type TEXT NOT NULL`: Matches the RabbitMQ routing key. Stored verbatim for client filtering (future).
- `payload JSONB NOT NULL`: Full event struct persisted for auditability and future re-processing. Not used in M5 API responses but present for operational debugging.
- `status TEXT NOT NULL DEFAULT 'sent'`: In M5, all notifications transition immediately from implicit `pending` to `sent` synchronously during processing. The column is present for forward-compatibility with async email queues.
- `sent_at TIMESTAMPTZ NULL`: Set to `now()` when the mock send log line is emitted. Null if processing fails before the log line.
- `processed_notifications.event_key TEXT PRIMARY KEY`: Composite string `"<event_type>:<correlation_id>"`. Using `correlation_id` as the dedup discriminator means the same logical notification from the same request is deduplicated. If two genuinely different events share the same `correlation_id` (pathological case), the second is dropped — acceptable given the low probability and the mock-only email implementation.

3.2 Go Structs — Notification Service

```go
// repository/notification_repository.go

package repository

import "time"

// Notification is a single persisted notification record.

type Notification struct {

    ID        string    // UUID v4, primary key; set by DB DEFAULT

    UserID    string    // UUID string; JWT sub of the URL owner

    EventType string    // routing key: "url.created" | "url.deleted" | "milestone.reached"

    Payload   []byte    // raw JSON of the event struct

    Status    string    // "pending" | "sent" | "failed"

    CreatedAt time.Time // set by DB DEFAULT

    SentAt    *time.Time // nil until mock send completes

}

// NotificationSummary is the projection returned by ListByUser.

// Omits the raw payload from API responses.

type NotificationSummary struct {

    ID        string    // used as cursor value

    UserID    string

    EventType string

    Status    string

    CreatedAt time.Time

    SentAt    *time.Time

}

// Sentinel errors

var (

    ErrNotificationNotFound = errors.New("notification not found")

)
```

```go
// handler/notifications.go — API response types

package handler

// NotificationListResponse is the JSON body returned by GET /notifications.

type NotificationListResponse struct {

    Notifications []NotificationItem `json:"notifications"`

    NextCursor    *string            `json:"next_cursor,omitempty"`

}

// NotificationItem is a single entry in the list response.

type NotificationItem struct {

    ID        string  `json:"id"`

    EventType string  `json:"event_type"`

    Status    string  `json:"status"`

    CreatedAt string  `json:"created_at"` // RFC3339

    SentAt    *string `json:"sent_at,omitempty"`

}
```

3.3 Go Structs — Circuit Breaker

```go
// gateway/circuitbreaker/circuitbreaker.go

package circuitbreaker

import (

    "sync"

    "sync/atomic"

    "time"

)

// State represents the circuit breaker's current operating mode.

type State int32

const (

    StateClosed   State = iota // Normal operation; requests flow through.

    StateOpen                  // Failure threshold exceeded; requests rejected immediately.

    StateHalfOpen              // Probe mode; one request allowed through to test recovery.

)

// Config holds the circuit breaker's tuning parameters.

type Config struct {

    // FailureThreshold is the number of consecutive failures within Window

    // required to trip the circuit from Closed → Open.

    FailureThreshold int

    // Window is the observation period for counting failures.

    // Failures older than Window do not count toward the threshold.

    Window time.Duration

    // OpenDuration is how long the circuit stays Open before transitioning

    // to HalfOpen and allowing a single probe request.

    OpenDuration time.Duration

}

// DefaultConfig returns the spec-required settings.

//   FailureThreshold: 5 consecutive failures

//   Window:          10 seconds

//   OpenDuration:    30 seconds

func DefaultConfig() Config {

    return Config{

        FailureThreshold: 5,

        Window:           10 * time.Second,

        OpenDuration:     30 * time.Second,

    }

}

// CircuitBreaker implements the three-state pattern for a single downstream target.

// All exported methods are goroutine-safe.
```

```go
type CircuitBreaker struct {

    cfg            Config

    mu             sync.Mutex    // protects state, failureCount, lastFailureAt, openedAt

    state          State

    failureCount   int           // consecutive failures in the current window

    lastFailureAt  time.Time     // time of the most recent failure; used for window reset

    openedAt       time.Time     // time the circuit transitioned to Open

}

// NewCircuitBreaker constructs a CircuitBreaker in the Closed state.

func NewCircuitBreaker(cfg Config) *CircuitBreaker
```

3.4 Go Structs — Rate Limiter

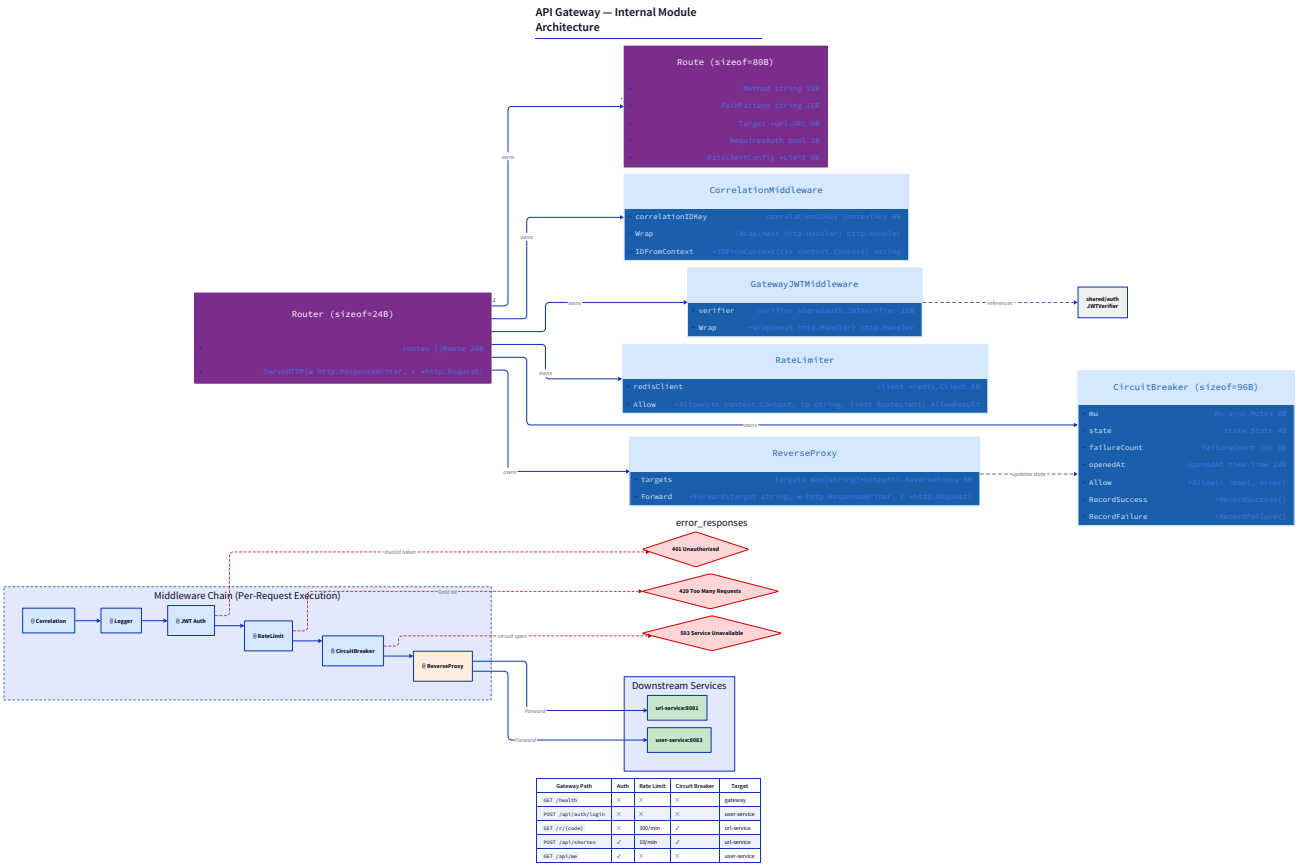
```go
// gateway/ratelimit/ratelimit.go

package ratelimit

import (

    "context"

    "time"

    "github.com/redis/go-redis/v9"

    "github.com/rs/zerolog"

)

// RateLimiter implements per-IP token-bucket rate limiting using Redis INCR + EXPIRE.

// Each IP+route combination has an independent counter with a fixed window.

type RateLimiter struct {

    client *redis.Client  // may be nil (fail-open on Redis unavailability)

    log    zerolog.Logger

}

// NewRateLimiter constructs a RateLimiter.

// client may be nil — the RateLimiter fails open when client is nil.

func NewRateLimiter(client *redis.Client, log zerolog.Logger) *RateLimiter

// AllowResult is the outcome of a rate limit check.

type AllowResult struct {

    Allowed    bool

    RetryAfter time.Duration // > 0 when Allowed=false; seconds to wait

    Remaining  int64         // requests remaining in the current window (informational)

}

// RouteLimit defines the rate limit policy for one route class.

type RouteLimit struct {

    Key    string        // route identifier; used as part of Redis key prefix

    Limit  int64         // max requests per Window per IP

    Window time.Duration // sliding-window duration

}

// Pre-defined route limits (from spec):

var (

    LimitShorten = RouteLimit{Key: "shorten", Limit: 10, Window: time.Minute}

    LimitRedirect = RouteLimit{Key: "redirect", Limit: 300, Window: time.Minute}

)
```

3.5 Go Structs — Gateway Proxy

```go
// gateway/proxy/proxy.go

package proxy

import (

    "net/http"

    "net/http/httputil"

    "net/url"

    "time"

    "github.com/rs/zerolog"

)

// DownstreamTarget represents a single upstream service the gateway routes to.

type DownstreamTarget struct {

    Name    string   // service name for logging: "url-service", etc.

    BaseURL *url.URL // parsed base URL; e.g. http://url-service:8081

}

// ReverseProxy wraps httputil.ReverseProxy with error handling and logging.

type ReverseProxy struct {

    targets map[string]*httputil.ReverseProxy // keyed by target.Name

    log     zerolog.Logger

    // HTTP client shared across all proxies; controls timeouts.

    client  *http.Client

}

// ProxyConfig configures the reverse proxy behavior.

type ProxyConfig struct {

    DialTimeout           time.Duration // default: 5s

    ResponseHeaderTimeout time.Duration // default: 30s (spec: 30s timeout counts as failure)

    MaxIdleConnsPerHost   int           // default: 10

}
```

3.6 Structured Log Entry (finalized schema)

```go
// shared/logger/logger.go

// LogFields documents the canonical set of structured fields on every log line.

// This is documentation only; actual emission uses zerolog's fluent API.

// All services MUST produce logs conforming to this schema.

type LogFields struct {

    Level         string `json:"level"`          // "debug"|"info"|"warn"|"error"

    Time          int64  `json:"time"`           // Unix milliseconds (zerolog.TimeFormatUnixMs)

    Service       string `json:"service"`        // injected at logger construction

    CorrelationID string `json:"correlation_id"` // from X-Correlation-ID; "" if not available

    Method        string `json:"method"`         // HTTP method; "" for non-request logs

    Path          string `json:"path"`           // HTTP path; "" for non-request logs

    Status        int    `json:"status"`         // HTTP response status; 0 for non-request logs

    DurationMS    int64  `json:"duration_ms"`    // handler duration; 0 for non-request logs

    Msg           string `json:"msg"`            // human-readable message

}
```



4. Interface Contracts

4.1 `NotificationRepository`

```go
// repository/notification_repository.go

// NotificationRepository defines persistence for notification records.

// All implementations must be safe for concurrent use.

type NotificationRepository interface {

    // Insert writes a new notification row and sets sent_at = now() atomically.

    // Uses the pool directly (no external transaction; the notification service

    // has no dual-write requirement — there is no outbox here).

    // Returns a wrapped error on DB failure.

    Insert(ctx context.Context, n Notification) error

    // ListByUser returns a cursor-paginated list of notifications for userID,

    // ordered by created_at DESC (newest first).

    // afterID: if non-empty, returns rows with (created_at, id) strictly less

    //          than the anchor row identified by afterID (same cursor logic as M3).

    // limit: maximum rows to return; caller passes pageSize+1 to detect next page.

    // Returns empty slice (not nil) when no rows match.

    ListByUser(ctx context.Context, userID string, afterID string, limit int) ([]NotificationSummary, error)

    // IsProcessed checks whether eventKey exists in processed_notifications.

    // Returns true if found, false on not-found or DB error (fail-open for idempotency).

    IsProcessed(ctx context.Context, eventKey string) (bool, error)

    // MarkProcessed inserts eventKey into processed_notifications.

    // ON CONFLICT DO NOTHING makes this idempotent.

    // Uses pool directly (same connection as Insert is not required — see § 5.2).

    MarkProcessed(ctx context.Context, eventKey string) error

}
```

SQL queries:

```sql
-- Insert
INSERT INTO notifications (user_id, event_type, payload, status, sent_at)
VALUES ($1, $2, $3, 'sent', now())
RETURNING id, created_at

-- ListByUser (with cursor)
SELECT id, user_id, event_type, status, created_at, sent_at
FROM notifications
WHERE user_id = $1
  AND ($2::uuid IS NULL OR (created_at, id) < (
      SELECT created_at, id FROM notifications WHERE id = $2::uuid
  ))
ORDER BY created_at DESC, id DESC
LIMIT $3

-- IsProcessed
SELECT 1 FROM processed_notifications WHERE event_key = $1

-- MarkProcessed
INSERT INTO processed_notifications (event_key) VALUES ($1)
ON CONFLICT (event_key) DO NOTHING
```

4.2 `CircuitBreaker`

```go
// gateway/circuitbreaker/circuitbreaker.go

// Allow checks whether a request should be forwarded to the downstream service.

// Returns (true, nil) in Closed or HalfOpen state (probe allowed).

// Returns (false, ErrCircuitOpen) in Open state — caller must return 503.

// Returns (false, ErrCircuitOpen) in HalfOpen state if a probe is already in-flight

//   (HalfOpen allows exactly ONE concurrent probe; subsequent calls are rejected).

//

// State transition rules (enforced inside Allow with mu held):

//   Closed:   always returns true.

//   Open:     if time.Since(openedAt) >= cfg.OpenDuration → transition to HalfOpen.

//             Still in cooldown → return false.

//   HalfOpen: if no probe in-flight → mark probe in-flight, return true.

//             Probe already in-flight → return false (treat as Open).

func (cb *CircuitBreaker) Allow() (bool, error)

// RecordSuccess notifies the circuit breaker that the last proxied request succeeded.

// State transition rules:

//   Closed:   reset failureCount to 0.

//   HalfOpen: transition to Closed, clear probe in-flight flag, reset failureCount.

//   Open:     no-op (should not receive success in Open state; log warning).

func (cb *CircuitBreaker) RecordSuccess()

// RecordFailure notifies the circuit breaker that the last proxied request failed.

// A failure is defined as: HTTP 5xx response from downstream, or a timeout/connection error.

// State transition rules:

//   Closed:   increment failureCount; if failureCount >= cfg.FailureThreshold

//             AND time.Since(lastFailureAt) <= cfg.Window → trip to Open; record openedAt.

//             If time.Since(lastFailureAt) > cfg.Window → reset failureCount to 1 (new window).

//   HalfOpen: probe failed → transition back to Open; record new openedAt; clear probe flag.

//   Open:     no-op (should not receive failure in Open state; log warning).

func (cb *CircuitBreaker) RecordFailure()

// State returns the current state without acquiring the mutex (uses atomic read).

// Callers use this for observability/logging only — do not make routing decisions

// based on this value; use Allow() for that (which holds the mutex).

func (cb *CircuitBreaker) State() State

// ErrCircuitOpen is returned by Allow when the circuit is in Open or HalfOpen-no-probe state.

var ErrCircuitOpen = errors.New("circuit breaker open")
```

State machine invariants:

- `failureCount` is always 0 in `StateOpen` and `StateHalfOpen`.
- `openedAt.IsZero()` is true only in `StateClosed`.
- The probe in-flight flag is a `bool` field `probeInFlight` protected by `mu`. It is set to `true` when `Allow()` transitions HalfOpen → allows probe, and cleared by `RecordSuccess()` or `RecordFailure()` from HalfOpen.

4.3 `RateLimiter`

```go
// gateway/ratelimit/ratelimit.go

// Allow checks whether the request from ip should be permitted for the given route policy.

// Redis key format: "ratelimit:{route.Key}:{ip}"

// Example: "ratelimit:shorten:192.168.1.1"

//

// Algorithm (atomic at Redis server level):

//   1. INCR "ratelimit:{route.Key}:{ip}" → current count

//   2. If current count == 1: EXPIRE key route.Window (sets TTL on first request in window)

//   3. If current count > route.Limit: return AllowResult{Allowed: false, RetryAfter: TTL of key}

//   4. Else: return AllowResult{Allowed: true, Remaining: route.Limit - current}

//

// If Redis client is nil or any Redis error occurs:

//   Log warning with error details.

//   Return AllowResult{Allowed: true} — fail-open; do NOT block traffic.

//

// RetryAfter is obtained via TTL command after the INCR reveals the limit is exceeded.

// TTL returns the remaining seconds; if TTL returns -1 (no expiry), use route.Window as fallback.

//

// Parameters:

//   ctx:   request context; used for Redis call timeout

//   ip:    client IP string extracted from X-Forwarded-For or RemoteAddr

//   route: RouteLimit policy for this endpoint

//

// Returns AllowResult. Never returns an error — all errors result in fail-open.

func (rl *RateLimiter) Allow(ctx context.Context, ip string, route RouteLimit) AllowResult
```

Redis command sequence (INCR + conditional EXPIRE):

```go
// Implementation inside Allow:

pipe := rl.client.Pipeline()

incrCmd := pipe.Incr(ctx, key)

// Conditional EXPIRE only on first request avoids resetting TTL on every request.

// Use a Lua script for atomic INCR+EXPIRE-if-new:

const luaIncrExpire = `

local current = redis.call('INCR', KEYS[1])

if current == 1 then

    redis.call('EXPIRE', KEYS[1], ARGV[1])

end

return current

`

// Execute as a single atomic script — no pipeline needed:

result, err := rl.client.Eval(ctx, luaIncrExpire, []string{key},

    int(route.Window.Seconds())).Int64()
```

Why Lua script over pipeline: A pipeline of INCR + EXPIRE is not atomic — two concurrent requests could both INCR and both see `current == 1`, causing double EXPIRE. The Lua script executes atomically at the Redis server, making this race impossible.

4.4 `ReverseProxy.Forward`

```go
// gateway/proxy/proxy.go

// Forward proxies the incoming request to the named target service.

// targetName must match a key in rp.targets (registered at construction).

// Injects X-Correlation-ID into the forwarded request if not already present.

// Sets X-Forwarded-For with the client IP.

//

// On success: streams the downstream response to w verbatim (headers + body).

// On target not found: writes 502 {"error":"unknown upstream"}.

// On proxy error (connection refused, timeout, circuit open):

//   The circuitBreaker parameter (passed at construction per-target) handles

//   RecordFailure() before this function returns; caller must not double-record.

//

// Timeout: ResponseHeaderTimeout=30s is enforced by the underlying http.Client.

// A timeout error is treated as a downstream failure for circuit breaker purposes.

func (rp *ReverseProxy) Forward(targetName string, w http.ResponseWriter, r *http.Request)
```

`httputil.ReverseProxy` error hook:

```go
// Set on the httputil.ReverseProxy at construction:

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

    log.Error().Err(err).Str("target", targetName).Msg("proxy error")

    cb.RecordFailure()

    writeJSON(w, http.StatusBadGateway, ErrorResponse{Error: "upstream error"})

}

// ModifyResponse is used to intercept 5xx from downstream:

proxy.ModifyResponse = func(resp *http.Response) error {

    if resp.StatusCode >= 500 {

        cb.RecordFailure()

    } else {

        cb.RecordSuccess()

    }

    return nil

}
```

4.5 Correlation Middleware

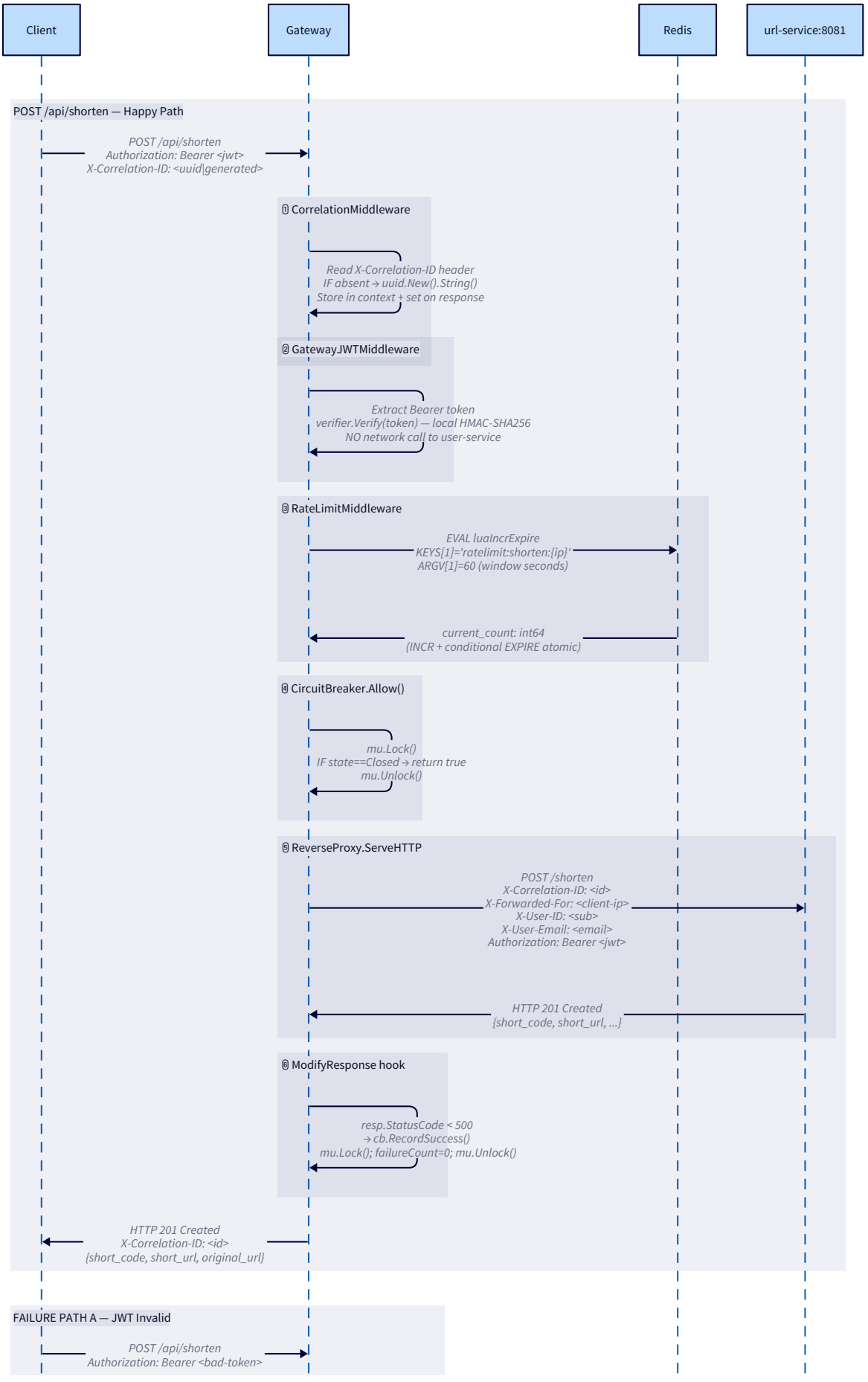
```go
// gateway/middleware/correlation.go

// CorrelationMiddleware is an http.Handler wrapper that:

//  1. Reads X-Correlation-ID from the incoming request header.

//  2. If absent or empty, generates a new UUID v4.

//  3. Sets X-Correlation-ID on the request before passing to next.

//  4. Sets X-Correlation-ID on the response.

//  5. Stores the correlation ID in the request context under correlationIDKey.

//

// All downstream forwarded requests carry this header.

// All log lines within the request's context use this ID.

func CorrelationMiddleware(next http.Handler) http.Handler

// CorrelationIDFromContext retrieves the correlation ID stored by CorrelationMiddleware.

// Returns empty string if not present (should not happen after middleware runs).

func CorrelationIDFromContext(ctx context.Context) string

// contextKey type is unexported to prevent collisions.

type contextKey string

const correlationIDKey contextKey = "correlation_id"
```

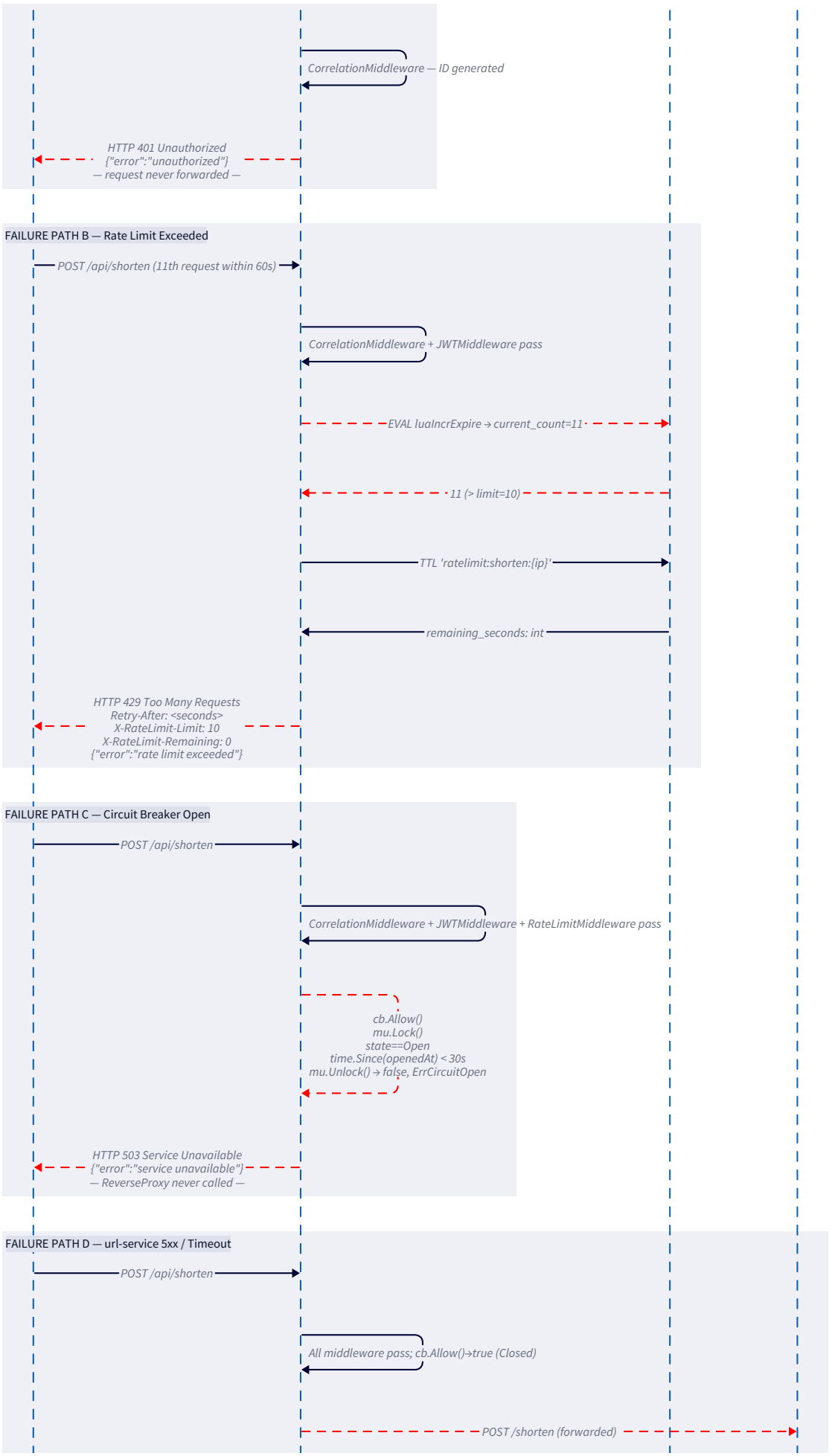
4.6 Gateway JWT Middleware

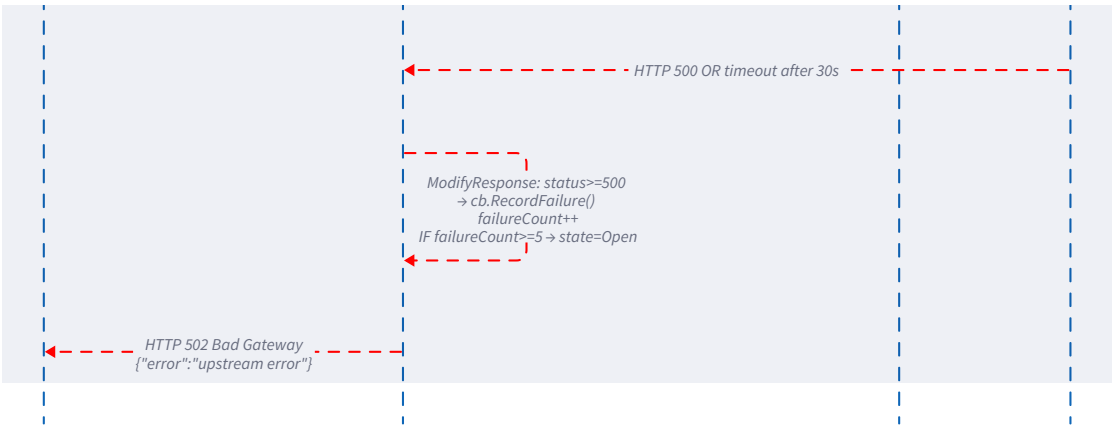
```go
// gateway/middleware/jwt.go

// GatewayJWTMiddleware returns an http.Handler wrapper that:
//  1. Reads Authorization: Bearer <token> from the incoming request.
//  2. Verifies the token using the shared JWTVerifier (local HMAC; no network call).
//  3. On success: forwards the request to next, adding X-User-ID and X-User-Email
//     headers with values from the verified claims (so downstream services can
//     optionally trust these without re-verifying).
//  4. On failure: returns 401 {"error":"unauthorized"} immediately;
//     the downstream service never receives the request.
//
// Routes exempt from JWT validation: POST /api/auth/register, POST /api/auth/login.
// These are matched by path prefix "/api/auth/"; all other /api/* routes require JWT.
// GET /r/:code does NOT require JWT (public redirect).
// GET /health does NOT require JWT.
//
// verifier: a shared/auth.JWTVerifier constructed with JWT_SECRET from environment.
// log: structured logger with "service":"gateway" pre-set.
func GatewayJWTMiddleware(verifier sharedauth.JWTVerifier, log zerolog.Logger) func(http.Handler) http.Handler
```

4.7 `shared/logger.New`

```go
// shared/logger/logger.go

// New returns a zerolog.Logger configured for structured JSON output.

// serviceName is embedded in every log line as "service".

// w is the output writer; pass os.Stdout in production.

//

// Configuration applied:

//   zerolog.TimeFieldFormat = zerolog.TimeFormatUnixMs

//   Level: zerolog.InfoLevel (overridable via LOG_LEVEL env var: debug|info|warn|error)

//   CallerSkipFrameCount: 2 (so Caller() reports the actual call site, not the logger wrapper)

//

// The returned Logger is value-safe to copy and store in structs.

// No global zerolog.GlobalLevel is set — each service sets its own level.

func New(serviceName string, w io.Writer) zerolog.Logger

// WithCorrelationID returns a child logger with correlation_id field added.

// Call at the start of each HTTP request handler with the ID from context.

// Usage: log := logger.WithCorrelationID(baseLog, correlationID)

func WithCorrelationID(log zerolog.Logger, correlationID string) zerolog.Logger

// RequestLogger returns an http.Handler middleware that logs each request/response.

// Emits one log line per request with all LogFields populated.

// Extracts correlation_id from context (set by CorrelationMiddleware).

func RequestLogger(log zerolog.Logger) func(http.Handler) http.Handler
```


Client
Gateway
Redis
url-service:8081
POST /api/shorten — Happy Path
POST /api/shorten
Authorization: Bearer <jwt>
X-Correlation-ID: <uuid|generated>
0 CorrelationMiddleware
Read X-Correlation-ID header
IF absent → uuid.New().String()
Store in context + set on response
2 GatewayJWTMiddleware
Extract Bearer token
verifier.Verify(token) — local HMAC-SHA256
NO network call to user-service
3 RateLimitMiddleware
EVAL luaIncrExpire
KEYS[1]='ratelimit:shorten:{ip}'
ARGV[1]=60 (window seconds)
current_count: int64
(INCR + conditional EXPIRE atomic)
4 CircuitBreaker.Allow()
mu.Lock()
IF state==Closed → return true
mu.Unlock()
5 ReverseProxy.ServeHTTP
POST /shorten
X-Correlation-ID: <id>
X-Forwarded-For: <client-ip>
X-User-ID: <sub>
X-User-Email: <email>
Authorization: Bearer <jwt>
HTTP 201 Created
{short_code, short_url, ...}
6 ModifyResponse hook
resp.StatusCode < 500
→ cb.RecordSuccess()
mu.Lock(); failureCount=0; mu.Unlock()
HTTP 201 Created
X-Correlation-ID: <id>
{short_code, short_url, original_url}
FAILURE PATH A — JWT Invalid
POST /api/shorten
Authorization: Bearer <bad-token>


CorrelationMiddleware — ID generated
HTTP 401 Unauthorized
{"error":"unauthorized"}
— request never forwarded —
FAILURE PATH B — Rate Limit Exceeded
POST /api/shorten (11th request within 60s)
CorrelationMiddleware + JWTMiddleware pass
EVAL luaIncrExpire → current_count=11
11 (> limit=10)
TTL 'ratelimit:shorten:{ip}'
remaining_seconds: int
HTTP 429 Too Many Requests
Retry-After: <seconds>
X-RateLimit-Limit: 10
X-RateLimit-Remaining: 0
{"error":"rate limit exceeded"}
FAILURE PATH C — Circuit Breaker Open
POST /api/shorten
CorrelationMiddleware + JWTMiddleware + RateLimitMiddleware pass
cb.Allow()
mu.Lock()
state==Open
time.Since(openedAt) < 30s
mu.Unlock() → false, ErrCircuitOpen
HTTP 503 Service Unavailable
{"error":"service unavailable"}
— ReverseProxy never called —
FAILURE PATH D — url-service 5xx / Timeout
POST /api/shorten
All middleware pass; cb.Allow()→true (Closed)
POST /shorten (forwarded)


HTTP 500 OR timeout after 30s
ModifyResponse: status>=500
→ cb.RecordFailure()
failureCount++
IF failureCount>=5 → state=Open
HTTP 502 Bad Gateway
{"error":"upstream error"}

5. Algorithm Specification

5.1 Notification Processor — Full Algorithm

```
PROCEDURE NotificationProcessor.Process(ctx, d amqp091.Delivery) ProcessResult:
  -- Step 0: Panic recovery (defer at top)
  DEFER recover() → log.Error, return ProcessResultError
  -- Step 1: Determine event type from routing key or payload
  routingKey = d.RoutingKey  -- "url.created" | "url.deleted" | "milestone.reached"
  -- Step 2: Parse into the correct event struct based on routing key
  SWITCH routingKey:
  CASE "url.created":
    var event events.URLCreatedEvent
    IF json.Unmarshal(d.Body, &event) fails → log.Error, return ProcessResultPoisoned
    userID = event.UserID
    userEmail = event.UserEmail
    correlationID = event.CorrelationID
    message = fmt.Sprintf("New short URL created: %s → %s", event.ShortCode, event.OriginalURL)
  CASE "url.deleted":
    var event events.URLDeletedEvent
    IF json.Unmarshal(d.Body, &event) fails → log.Error, return ProcessResultPoisoned
    userID = event.UserID
    userEmail = event.UserEmail
    correlationID = event.CorrelationID
    message = fmt.Sprintf("Short URL deleted: %s", event.ShortCode)
  CASE "milestone.reached":
    var event events.MilestoneReachedEvent
    IF json.Unmarshal(d.Body, &event) fails → log.Error, return ProcessResultPoisoned
    userID = event.UserID
    userEmail = event.UserEmail
    correlationID = event.CorrelationID
    message = fmt.Sprintf("Milestone reached: %d clicks on %s", event.Milestone, event.ShortCode)
  DEFAULT:
    log.Warn().Str("routing_key", routingKey).Msg("unknown routing key; ACKing")
    return ProcessResultPoisoned
  -- Step 3: Validate required fields
  IF userID == "" OR correlationID == "":
    log.Error().Str("routing_key", routingKey).Msg("missing user_id or correlation_id; ACKing")
    return ProcessResultPoisoned
  -- Step 4: Build dedup key
  eventKey = routingKey + ":" + correlationID
  -- Step 5: Idempotency pre-check
  already, _ = repo.IsProcessed(ctx, eventKey)  -- errors treated as false (fail-open)
  IF already:
    log.Debug().Str("event_key", eventKey).Msg("duplicate notification; skipping")
    return ProcessResultDuplicate
  -- Step 6: Insert notification row
  n = Notification{
      UserID:    userID,
      EventType: routingKey,
      Payload:   d.Body,
      Status:    "sent",
  }
  IF repo.Insert(ctx, n) fails:
    log.Error().Err(err).Msg("notification insert failed")
    return ProcessResultError
  -- Step 7: Mock send — log the notification
  log.Info().
      Str("correlation_id", correlationID).
      Str("event_type", routingKey).
      Str("user_email", userEmail).
      Str("message", message).
      Msg("would send email to user")
  -- Step 8: Mark processed (after successful insert + log)
  IF repo.MarkProcessed(ctx, eventKey) fails:
    -- Insert succeeded; log error but don't fail the message.
    -- On redelivery, Insert will create a second row (acceptable: dedup pre-check
    -- returned false, so re-run will create a duplicate notification but not crash).
    -- Accept this rare edge case; it requires a service crash between step 6 and step 8.
    log.Error().Err(err).Str("event_key", eventKey).Msg("mark processed failed; duplicate possible on redeliver")
  return ProcessResultInserted
END PROCEDURE
```

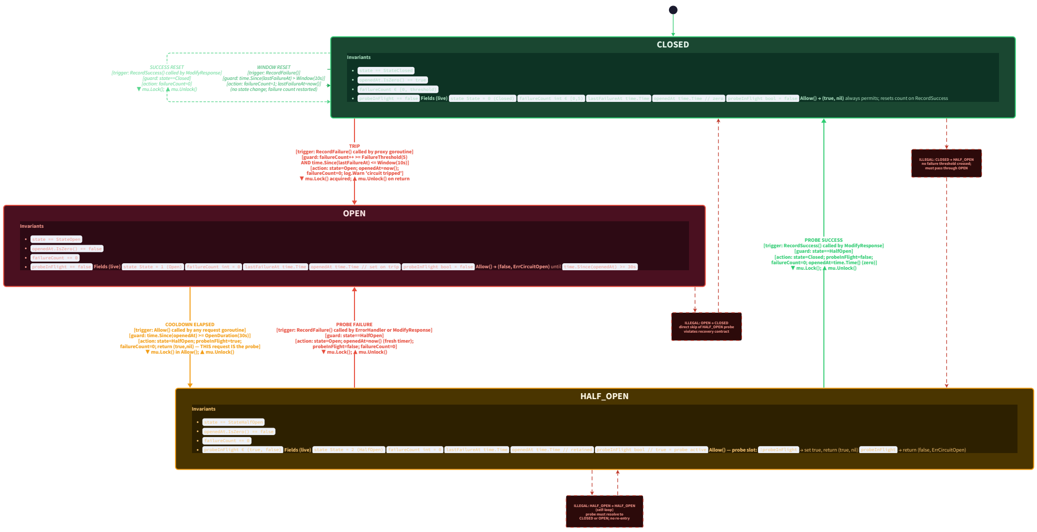
Why MarkProcessed is NOT in the same transaction as Insert: The notification service has no dual-write requirement (no outbox, no RabbitMQ publish from within the notification handler). Using two sequential DB operations (Insert, then MarkProcessed) simplifies the code. The rare crash-between-operations case creates a duplicate notification row — acceptable given mock email delivery in M5.

5.2 Circuit Breaker State Machine — Detailed Transitions

```
PROCEDURE CircuitBreaker.Allow() (bool, error):
  cb.mu.Lock()
  DEFER cb.mu.Unlock()
  SWITCH cb.state:
  CASE StateClosed:
    return true, nil
  CASE StateOpen:
    IF time.Since(cb.openedAt) >= cb.cfg.OpenDuration:
      -- Cooldown elapsed; transition to HalfOpen for probe
      cb.state = StateHalfOpen
      cb.probeInFlight = true
      cb.failureCount = 0
      return true, nil   -- this request IS the probe
    ELSE:
      return false, ErrCircuitOpen
  CASE StateHalfOpen:
    IF cb.probeInFlight:
      -- A probe is already in-flight; reject this request
      return false, ErrCircuitOpen
    ELSE:
      -- Edge case: HalfOpen but no probe in-flight (should not happen; defensive)
      cb.probeInFlight = true
      return true, nil
  END SWITCH
END PROCEDURE
PROCEDURE CircuitBreaker.RecordSuccess():
  cb.mu.Lock()
  DEFER cb.mu.Unlock()
  SWITCH cb.state:
  CASE StateClosed:
    cb.failureCount = 0    -- reset window counter
  CASE StateHalfOpen:
    -- Probe succeeded; circuit healed
    cb.state = StateClosed
    cb.probeInFlight = false
    cb.failureCount = 0
    cb.openedAt = time.Time{}   -- zero value signals Closed state
  CASE StateOpen:
    cb.log.Warn().Msg("circuit breaker: RecordSuccess called in Open state (unexpected)")
  END SWITCH
END PROCEDURE
PROCEDURE CircuitBreaker.RecordFailure():
  cb.mu.Lock()
  DEFER cb.mu.Unlock()
  now = time.Now()
  SWITCH cb.state:
  CASE StateClosed:
    -- Check if this failure is within the current window
    IF cb.failureCount == 0 OR now.Sub(cb.lastFailureAt) > cb.cfg.Window:
      -- New window; start fresh
      cb.failureCount = 1
    ELSE:
      cb.failureCount++
    cb.lastFailureAt = now
    IF cb.failureCount >= cb.cfg.FailureThreshold:
      -- Trip to Open
      cb.state = StateOpen
      cb.openedAt = now
      cb.failureCount = 0
      cb.log.Warn().
          Int("threshold", cb.cfg.FailureThreshold).
          Dur("window", cb.cfg.Window).
          Msg("circuit breaker tripped to Open")
  CASE StateHalfOpen:
    -- Probe failed; return to Open with fresh openedAt
    cb.state = StateOpen
    cb.openedAt = now
    cb.probeInFlight = false
    cb.failureCount = 0
    cb.log.Warn().Msg("circuit breaker half-open probe failed; returning to Open")
  CASE StateOpen:
    cb.log.Warn().Msg("circuit breaker: RecordFailure called in Open state (unexpected)")
  END SWITCH
END PROCEDURE
```

Circuit Breaker State Machine

url-shortener-m5 · `gateway/circuitbreaker/circuitbreaker.go` Mutex (`mu sync.Mutex`) held on ALL state reads/writes except `State()` (atomic)



CLOSED
Invariants
state == StateClosed
openedAt.IsZero() == true
failureCount ∈ [0, threshold)
probeInFlight == false
Fields (live)
state State = 0 (Closed)
failureCount int ∈ [0,5)
lastFailureAt time.Time
openedAt time.Time // zero
probeInFlight bool = false
Allow() → (true, nil) always permits; resets count on RecordSuccess
SUCCESS RESET
[trigger: RecordSuccess() called by ModifyResponse]
[guard: state==Closed]
[action: failureCount=0]
▼ mu.Lock(); ▲ mu.Unlock()
WINDOW RESET
[trigger: RecordFailure()]
[guard: time.Since(lastFailureAt) > Window(10s)]
[action: failureCount=1; lastFailureAt=now()]
(no state change; failure count restarted)
TRIP
[trigger: RecordFailure() called by proxy goroutine]
[guard: failureCount++ >= FailureThreshold(5)
AND time.Since(lastFailureAt) <= Window(10s)]
[action: state=Open; openedAt=now();
failureCount=0; log.Warn 'circuit tripped']
▼ mu.Lock() acquired; ▲ mu.Unlock() on return
ILLEGAL: CLOSED → HALF_OPEN
no failure threshold crossed;
must pass through OPEN
OPEN
Invariants
state == StateOpen
openedAt.IsZero() == false
failureCount == 0
probeInFlight == false
Fields (live)
state State = 1 (Open)
failureCount int = 0
lastFailureAt time.Time
openedAt time.Time // set on trip
probeInFlight bool = false
Allow() → (false, ErrCircuitOpen) until
time.Since(openedAt) >= 30s
PROBE SUCCESS
[trigger: RecordSuccess() called by ModifyResponse]
[guard: state==HalfOpen]
[action: state=Closed; probeInFlight=false;
failureCount=0; openedAt=time.Time{} (zero)]
▼ mu.Lock(); ▲ mu.Unlock()
ILLEGAL: OPEN → CLOSED
direct skip of HALF_OPEN probe
violates recovery contract
COOLDOWN ELAPSED
[trigger: Allow() called by any request goroutine]
[guard: time.Since(openedAt) >= OpenDuration(30s)]
[action: state=HalfOpen; probeInFlight=true;
failureCount=0; return (true,nil) — THIS request IS the probe]
▼ mu.Lock() in Allow(); ▲ mu.Unlock()
PROBE FAILURE
[trigger: RecordFailure() called by ErrorHandler or ModifyResponse]
[guard: state==HalfOpen]
[action: state=Open; openedAt=now() (fresh timer);
probeInFlight=false; failureCount=0]
▼ mu.Lock(); ▲ mu.Unlock()
HALF_OPEN
Invariants
state == StateHalfOpen
openedAt.IsZero() == false
failureCount == 0
probeInFlight ∈ {true, false}
Fields (live)
state State = 2 (HalfOpen)
failureCount int = 0
lastFailureAt time.Time
openedAt time.Time // retained
probeInFlight bool // true = probe active
Allow() — probe slot:
!probeInFlight
→ set true, return (true, nil)
probeInFlight
→ return (false, ErrCircuitOpen)
ILLEGAL: HALF_OPEN → HALF_OPEN
(self-loop)
probe must resolve to
CLOSED or OPEN; no re-entry

Legend 🟢 CLOSED — normal traffic flow 🔴 OPEN — fast-fail; 503 returned immediately 🟡 HALF_OPEN — single probe allowed; all others → 503 **Failure definition:** HTTP 5xx from downstream OR connection refused / timeout (30 s) **Mutex:** `mu sync.Mutex` acquired in `Allow()`, `RecordSuccess()`, `RecordFailure()` **Atomic read:** `State()` uses `atomic.LoadInt32` (no mutex needed for observability-only reads) **Config (DefaultConfig):** `FailureThreshold = 5` consecutive failures `Window = 10 s` observation period `OpenDuration = 30 s` before probe allowed

5.3 Gateway Request Routing Algorithm

```
PROCEDURE GatewayRouter.ServeHTTP(w http.ResponseWriter, r *http.Request):
  -- Middleware chain (applied in order, outermost first):
  -- 1. CorrelationMiddleware — generate/propagate X-Correlation-ID
  -- 2. RequestLogger — log all requests (applied after correlation to capture ID)
  -- 3. GatewayJWTMiddleware — validate JWT for /api/* routes except /api/auth/*
  -- 4. RateLimitMiddleware — apply per-IP limits based on route pattern
  -- 5. Route-specific handler (proxy forward or error response)
  path = r.URL.Path
  -- Route matching (evaluated in specificity order):
  MATCH path:
  CASE "GET /health":
    → handler.NewHealthHandler("gateway", log)
  CASE "POST /api/auth/register":
    → NO JWT required
    → NO rate limit
    → rp.Forward("user-service", w, r)
  CASE "POST /api/auth/login":
    → NO JWT required
    → NO rate limit
    → rp.Forward("user-service", w, r)
    -- Strip /api prefix: forward as POST /login to user-service
  CASE "GET /r/{code}":
    → NO JWT required
    → RateLimitMiddleware(LimitRedirect)
    → circuitBreaker.Allow() check
    → rp.Forward("url-service", w, r)
    -- Forward as GET /{code}
  CASE "POST /api/shorten":
    → JWT required
    → RateLimitMiddleware(LimitShorten)
    → circuitBreaker.Allow() check
    → rp.Forward("url-service", w, r)
    -- Forward as POST /shorten
  CASE "GET /api/urls":
    → JWT required
    → rp.Forward("url-service", w, r)
    -- Forward as GET /urls
  CASE "DELETE /api/urls/{code}":
    → JWT required
    → rp.Forward("url-service", w, r)
  CASE "GET /api/stats/{code}":
    → NO JWT required (public stats)
    → rp.Forward("analytics-service", w, r)
    -- Forward as GET /stats/{code}
  CASE "GET /api/stats/{code}/timeline":
    → NO JWT required
    → rp.Forward("analytics-service", w, r)
  CASE "GET /api/me":
    → JWT required
    → rp.Forward("user-service", w, r)
    -- Forward as GET /me
  CASE "GET /api/notifications":
    → JWT required
    → rp.Forward("notification-service", w, r)
    -- Forward as GET /notifications
  DEFAULT:
    → 404 {"error":"not found"}
  END MATCH
END PROCEDURE
```

Path stripping: The gateway routes `/api/shorten` to url-service as `/shorten`. Use `httputil.ReverseProxy` with a custom `Director` that strips the `/api` prefix for all `/api/*` forwarding. The `/r/{code}` path strips `/r` and forwards as `/{code}`. The `/api/auth/*` paths strip `/api/auth` and forward as `/*` (e.g., `/register`, `/login`).

```go
// Director function for url-service proxy:

director := func(req *http.Request) {

    req.URL.Scheme = target.Scheme

    req.URL.Host = target.Host

    // Strip /api prefix

    req.URL.Path = strings.TrimPrefix(req.URL.Path, "/api")

    // Preserve query string

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

    req.Header.Set("X-Correlation-ID", CorrelationIDFromContext(req.Context()))

}
```

5.4 Rate Limit Middleware Integration

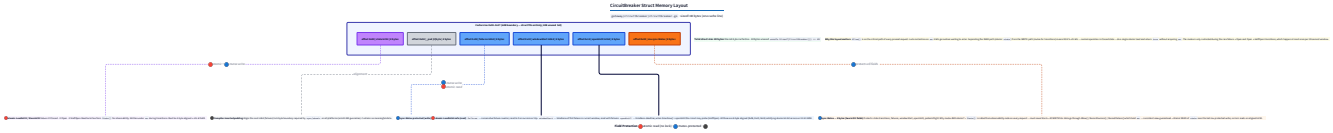
```
PROCEDURE RateLimitMiddleware(route RouteLimit, rl *RateLimiter) func(http.Handler) http.Handler:
  RETURN func(next http.Handler) http.Handler:
    RETURN http.HandlerFunc(func(w http.ResponseWriter, r *http.Request):
      ip = extractClientIP(r)
      result = rl.Allow(r.Context(), ip, route)
      IF !result.Allowed:
        retryAfterSecs = int(result.RetryAfter.Seconds())
        IF retryAfterSecs < 1: retryAfterSecs = 1
        w.Header().Set("Retry-After", strconv.Itoa(retryAfterSecs))
        w.Header().Set("X-RateLimit-Limit", strconv.FormatInt(route.Limit, 10))
        w.Header().Set("X-RateLimit-Remaining", "0")
        writeJSON(w, http.StatusTooManyRequests, ErrorResponse{Error: "rate limit exceeded"})
        return
      next.ServeHTTP(w, r)
    )
END PROCEDURE
```

5.5 Circuit Breaker Middleware Integration

```
PROCEDURE CircuitBreakerMiddleware(cb *CircuitBreaker) func(http.Handler) http.Handler:
  RETURN func(next http.Handler) http.Handler:
    RETURN http.HandlerFunc(func(w http.ResponseWriter, r *http.Request):
      allowed, err = cb.Allow()
      IF !allowed:
        log.Warn().Str("path", r.URL.Path).Msg("circuit breaker open; rejecting request")
        writeJSON(w, http.StatusServiceUnavailable,
            ErrorResponse{Error: "service unavailable"})
        return
      -- Request is allowed (Closed state or HalfOpen probe)
      -- RecordSuccess/RecordFailure is called inside ReverseProxy.ModifyResponse
      -- and ReverseProxy.ErrorHandler — not here. This avoids double-recording.
      next.ServeHTTP(w, r)
    )
END PROCEDURE
```

Sequence of middleware for url-service routes:

```
CorrelationMiddleware
  → RequestLogger
    → GatewayJWTMiddleware (if /api/*)
      → RateLimitMiddleware (if shorten or redirect)
        → CircuitBreakerMiddleware (url-service routes only)
          → ReverseProxy.Forward("url-service", ...)
```



5.6 Structured Logging Propagation Across Services

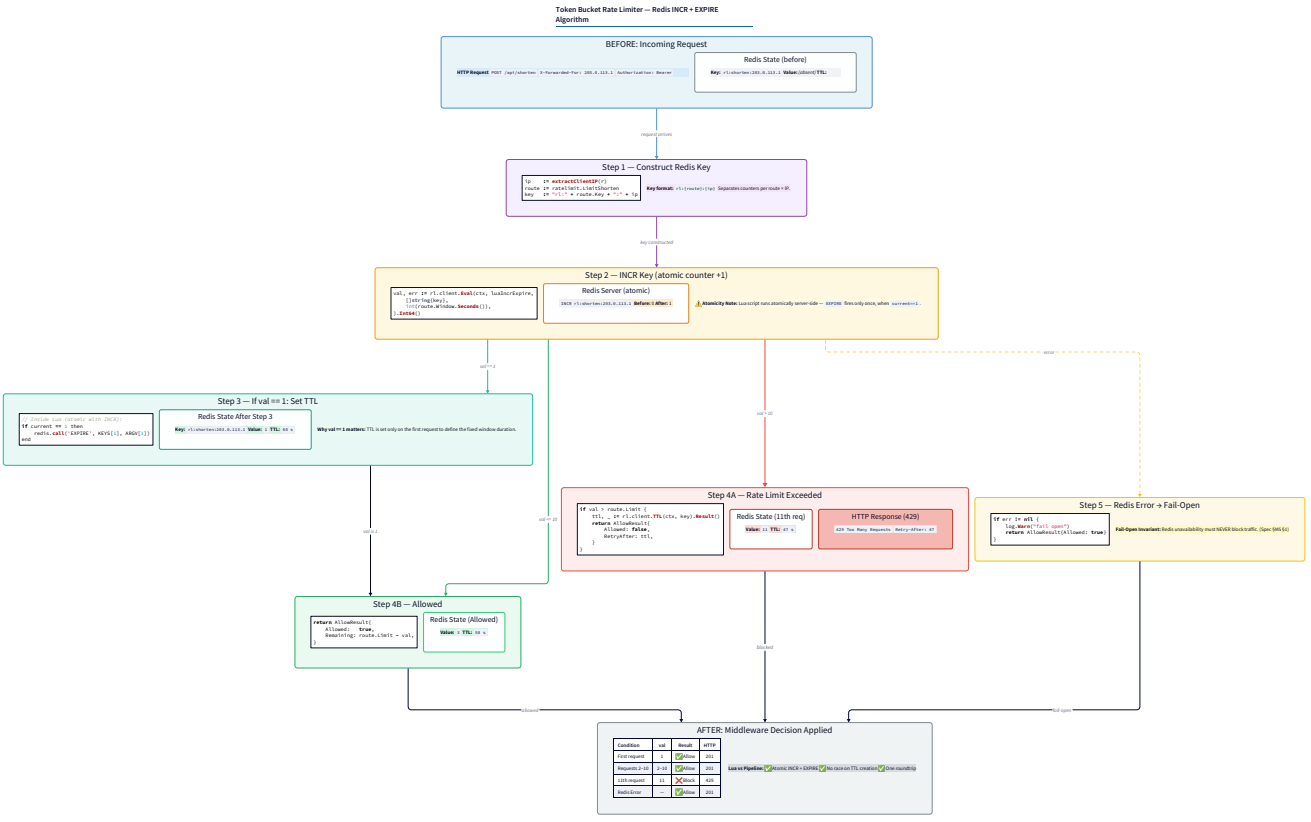
Every HTTP handler in every service must:

1. Extract `X-Correlation-ID` from the request header at the start of the handler.
2. Call `logger.WithCorrelationID(baseLog, correlationID)` to get a request-scoped logger.
3. Use the request-scoped logger for all log calls within that handler's execution path.
4. When making outbound HTTP calls (not applicable in M5 for services other than gateway), forward the `X-Correlation-ID` header. **Request logger middleware (applied in all services, not just gateway):**

```go
// shared/logger/logger.go

func RequestLogger(log zerolog.Logger) func(http.Handler) http.Handler {

    return func(next http.Handler) http.Handler {

        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

            start := time.Now()

            correlationID := r.Header.Get("X-Correlation-ID")

            reqLog := log.With().

                Str("correlation_id", correlationID).

                Str("method", r.Method).

                Str("path", r.URL.Path).

                Logger()

            // Wrap ResponseWriter to capture status code

            rw := &responseWriter{ResponseWriter: w, status: http.StatusOK}

            next.ServeHTTP(rw, r)

            reqLog.Info().

                Int("status", rw.status).

                Int64("duration_ms", time.Since(start).Milliseconds()).

                Msg("request completed")

        })

    }

}

// responseWriter wraps http.ResponseWriter to capture the written status code.

type responseWriter struct {

    http.ResponseWriter

    status int

}

func (rw *responseWriter) WriteHeader(code int) {

    rw.status = code

    rw.ResponseWriter.WriteHeader(code)

}
```

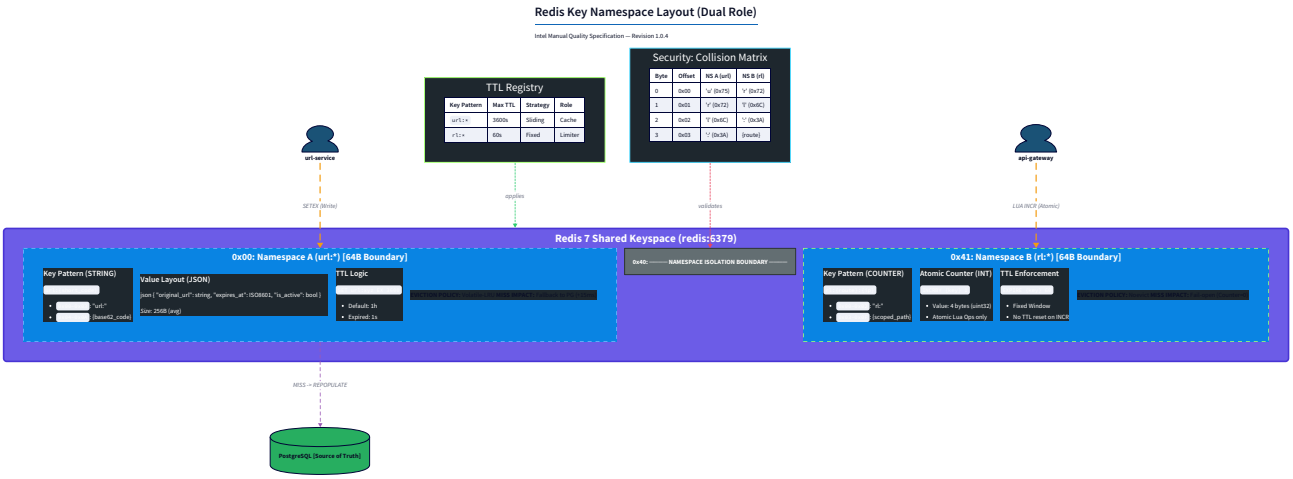
5.7 GET /notifications Handler Algorithm

```
PROCEDURE HandleListNotifications(w http.ResponseWriter, r *http.Request):
  claims, ok = middleware.ClaimsFromContext(r.Context())  -- from RequireAuth
  IF !ok → 401 {"error":"unauthorized"}
  afterCursor = r.URL.Query().Get("after")
  pageSizeStr = r.URL.Query().Get("limit")
  pageSize = 20  -- default
  IF pageSizeStr != "":
    pageSize, err = strconv.Atoi(pageSizeStr)
    IF err OR pageSize < 1 → 400 {"error":"limit must be a positive integer"}
  pageSize = clamp(pageSize, 1, 100)
  rows, err = repo.ListByUser(r.Context(), claims.Sub, afterCursor, pageSize+1)
  IF err → 500 + log.Error
  hasMore = len(rows) > pageSize
  IF hasMore:
    rows = rows[:pageSize]
  var nextCursor *string
  IF hasMore:
    last = rows[len(rows)-1].ID
    nextCursor = &last
  items = make([]NotificationItem, len(rows))
  FOR i, n := range rows:
    items[i] = NotificationItem{
        ID:        n.ID,
        EventType: n.EventType,
        Status:    n.Status,
        CreatedAt: n.CreatedAt.UTC().Format(time.RFC3339),
    }
    IF n.SentAt != nil:
      s = n.SentAt.UTC().Format(time.RFC3339)
      items[i].SentAt = &s
  writeJSON(w, 200, NotificationListResponse{
      Notifications: items,
      NextCursor:    nextCursor,
  })
END PROCEDURE
```



5.8 Gateway Startup Sequence

```
PROCEDURE GatewayMain():
  1. Parse all env vars with mustGetEnv; abort on missing.
  2. log = logger.New("gateway", os.Stdout)
  3. redisClient, err = ConnectRedis(ctx, redisAddr)
     IF err: log.Warn — rate limiter will fail-open
  4. verifier = sharedauth.NewJWTVerifier([]byte(jwtSecret))
  5. rateLimiter = ratelimit.NewRateLimiter(redisClient, log)
  6. urlServiceCB = circuitbreaker.NewCircuitBreaker(circuitbreaker.DefaultConfig())
  7. Construct ReverseProxy with targets:
       "url-service"          → URL_SERVICE_URL
       "analytics-service"    → ANALYTICS_SERVICE_URL
       "user-service"         → USER_SERVICE_URL
       "notification-service" → NOTIFICATION_SERVICE_URL
     Each target gets its own httputil.ReverseProxy instance.
     url-service proxy gets urlServiceCB injected into ModifyResponse + ErrorHandler.
     Other proxies have no circuit breaker in M5 (only url-service specified).
  8. Perform startup health checks (§ 5.2 of M1 spec) — log warnings for unhealthy.
  9. Build mux via router.New(...) — inject all dependencies.
  10. Apply middleware stack:
       handler = CorrelationMiddleware(
           RequestLogger(log)(
               mux
           )
       )
      Note: JWT and rate limit middleware are applied per-route, not globally.
  11. Start http.Server on PORT with:
       ReadTimeout:  5s
       WriteTimeout: 35s  -- 30s proxy timeout + 5s overhead
       IdleTimeout:  120s
  12. Block on server.ListenAndServe.
  13. On SIGTERM: server.Shutdown(10s context).
END PROCEDURE
```

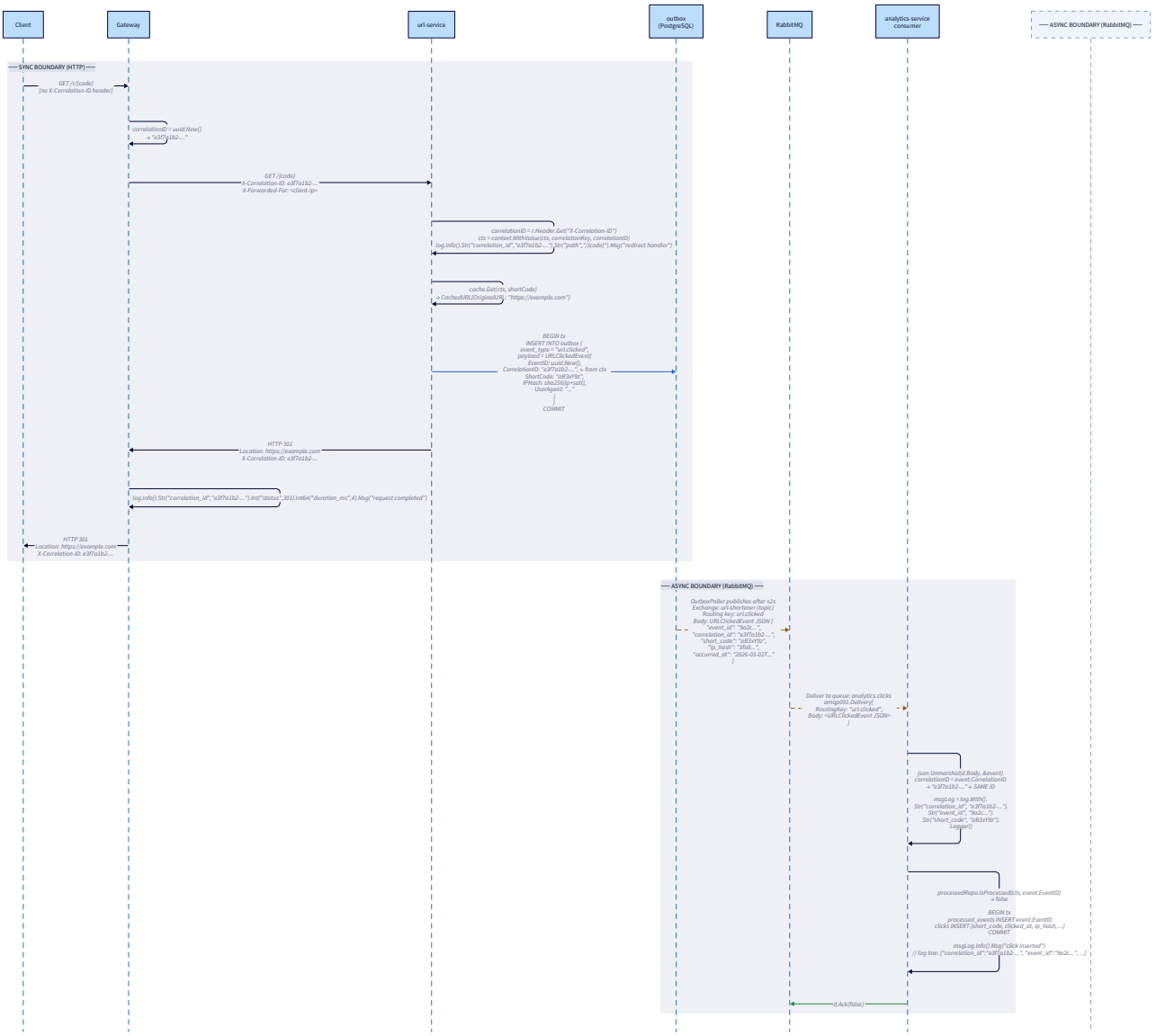


6. State Machine: Circuit Breaker

```
States: Closed → Open → HalfOpen → Closed
        HalfOpen → Open (on probe failure)
Transitions:
  Closed  →[failureCount >= threshold within window]→ Open
  Open    →[openDuration elapsed, Allow() called]→ HalfOpen
  HalfOpen→[RecordSuccess()]→ Closed
  HalfOpen→[RecordFailure()]→ Open (new openedAt)
Illegal transitions (must never occur):
  Closed  → HalfOpen  (must pass through Open)
  Open    → Closed    (must pass through HalfOpen probe)
  HalfOpen→ HalfOpen  (no self-transition; probe resolves to Closed or Open)
  Any state → any state without mutex held
HalfOpen probe semantics:
  - At most ONE request is allowed through as a probe.
  - All other concurrent requests during HalfOpen → rejected (treated as Open).
  - probeInFlight bool is the single-probe enforcement mechanism.
  - probeInFlight is set true in Allow(), cleared in RecordSuccess() or RecordFailure().
Failure counting semantics (Closed state):
  - failureCount resets to 1 (not 0) when a failure arrives after Window elapsed.
  - This models a sliding window: failures in the previous window don't count.
  - Consecutive failures within Window accumulate; non-consecutive failures may not trip.
Observability:
  - State() method uses atomic int32 read for lock-free observability.
  - State field is a State (int32); transitions write with mu held, read atomically.
```

GO

```go
// Atomic state read (no mutex):

func (cb *CircuitBreaker) State() State {

    return State(atomic.LoadInt32((*int32)(&cb.state)))

}

// State transitions write the state field while holding mu:

// cb.state = StateOpen  (inside mu-protected block)
```



Trace Reconstruction

Query any log aggregator for `correlation_id = "e3f7a1b2-..."`:

Service	Log line	Latency
gateway	`request completed` status=301	t+0ms
url-service	`redirect handler` + outbox INSERT	t+2ms
analytics-service	`click inserted` (async)	t+2000ms

Single ID links sync HTTP trace → async RabbitMQ consumer.

7. Error Handling Matrix

Notification Service

Error	Detected By	Recovery	ACK/NACK	Logged?	Notes
`MALFORMED_JSON`	`json.Unmarshal`	Log error, ACK	ACK	Error	Poison message
`UNKNOWN_ROUTING_KEY`	routing key switch	Log warn, ACK	ACK	Warn	Future-proof; unknown events dropped
`MISSING_USER_ID`	field validation	Log error, ACK	ACK	Error	Event schema violation
`DUPLICATE_EVENT`	`IsProcessed` → true	Log debug, ACK	ACK	Debug	Normal at-least-once redelivery
`INSERT_FAIL`	`repo.Insert`	Log error, NACK requeue	NACK	Error	Transient DB error
`MARK_PROCESSED_FAIL`	`repo.MarkProcessed`	Log error, ACK	ACK	Error	Insert succeeded; rare redelivery possible
`CONSUMER_PANIC`	`recover()` in supervisor	Log error, restart after 5s	N/A	Error	Delivery not ACKed; broker redelivers
`AMQP_DISCONNECTED`	Delivery channel closed	Log warn, reconnect loop	N/A	Warn	`/health` still 200
`LIST_NOTIFICATIONS_DB_FAIL`	`repo.ListByUser`	Log error, 500	N/A	Error	
`JWT_MISSING`	`RequireAuth` middleware	401	N/A	No	
`JWT_INVALID`	`RequireAuth` middleware	401	N/A	Warn	

API Gateway

Error	Detected By	HTTP Status	Response Body	Logged?	Notes
`RATE_LIMIT_EXCEEDED`	`RateLimiter.Allow` → false	429	`{"error":"rate limit exceeded"}`	No	`Retry-After` header set
`CIRCUIT_OPEN`	`CircuitBreaker.Allow` → false	503	`{"error":"service unavailable"}`	Warn	Only for url-service routes
`JWT_INVALID`	`GatewayJWTMiddleware`	401	`{"error":"unauthorized"}`	Warn	Request never forwarded
`JWT_MISSING`	`GatewayJWTMiddleware`	401	`{"error":"unauthorized"}`	No	
`JWT_EXPIRED`	`GatewayJWTMiddleware`	401	`{"error":"unauthorized"}`	No	No expiry detail in response
`DOWNSTREAM_5XX`	`ModifyResponse` status check	Proxied as-is	Downstream body	Warn + RecordFailure	Client sees the 5xx
`DOWNSTREAM_TIMEOUT`	`ErrorHandler` + context deadline	502	`{"error":"upstream error"}`	Error + RecordFailure	30s timeout
`DOWNSTREAM_CONN_REFUSED`	`ErrorHandler`	502	`{"error":"upstream error"}`	Error + RecordFailure	Service down
`REDIS_DOWN_FOR_RATE_LIMIT`	`RateLimiter` nil client or error	200 (allowed)	—	Warn	Fail-open: never block traffic
`UNKNOWN_ROUTE`	mux default	404	`{"error":"not found"}`	No	
`UPSTREAM_UNKNOWN`	`ReverseProxy.Forward` no target	502	`{"error":"unknown upstream"}`	Error	Programming error
`CORRELATION_ID_MISSING`	`CorrelationMiddleware`	—	—	No	Generated transparently

8. Route Registration in `gateway/main.go`

```go
// gateway/main.go — full route table with middleware chains

// Build each route's middleware stack explicitly.

// net/http 1.22 method+path patterns used throughout.

jwtMw := middleware.GatewayJWTMiddleware(verifier, log)

rateLimitShorten  := middleware.RateLimitMiddleware(ratelimit.LimitShorten, rateLimiter, log)

rateLimitRedirect := middleware.RateLimitMiddleware(ratelimit.LimitRedirect, rateLimiter, log)

cbMw := middleware.CircuitBreakerMiddleware(urlServiceCB, log)

mux := http.NewServeMux()

// Health — no auth, no rate limit

mux.HandleFunc("GET /health", handler.NewHealthHandler("gateway", log))

// Auth routes — no JWT, no rate limit, no circuit breaker

mux.HandleFunc("POST /api/auth/register",

    chain(rp.HandlerFor("user-service")))

mux.HandleFunc("POST /api/auth/login",

    chain(rp.HandlerFor("user-service")))

// Public redirect — rate limited, circuit broken

mux.Handle("GET /r/{code}",

    rateLimitRedirect(cbMw(rp.HandlerFor("url-service"))))

// URL service — JWT + rate limit (shorten) + circuit breaker

mux.Handle("POST /api/shorten",

    jwtMw(rateLimitShorten(cbMw(rp.HandlerFor("url-service")))))

mux.Handle("GET /api/urls",

    jwtMw(cbMw(rp.HandlerFor("url-service"))))

mux.Handle("DELETE /api/urls/{code}",

    jwtMw(cbMw(rp.HandlerFor("url-service"))))

// Analytics — public

mux.Handle("GET /api/stats/{code}",

    rp.HandlerFor("analytics-service"))

mux.Handle("GET /api/stats/{code}/timeline",

    rp.HandlerFor("analytics-service"))

// User service — JWT

mux.Handle("GET /api/me",

    jwtMw(rp.HandlerFor("user-service")))

// Notification service — JWT

mux.Handle("GET /api/notifications",

    jwtMw(rp.HandlerFor("notification-service")))

// Global middleware (outermost)

handler := middleware.CorrelationMiddleware(

    logger.RequestLogger(log)(mux),
```

```
)
```

`rp.HandlerFor(name string) http.Handler` returns a pre-built `http.Handler` that calls `rp.Forward(name, w, r)` for the named downstream target. This enables clean composition with the middleware chain. {{DIAGRAM:tdd-diag-38}}

9. Concurrency Specification

9.1 Circuit Breaker

Field	Access Pattern	Protection
`state`	Written under `mu`; read via `atomic.LoadInt32`	`sync.Mutex` for writes; atomic for reads
`failureCount`	Read and written under `mu`	`sync.Mutex`
`lastFailureAt`	Read and written under `mu`	`sync.Mutex`
`openedAt`	Read and written under `mu`	`sync.Mutex`
`probeInFlight`	Read and written under `mu`	`sync.Mutex`
Lock ordering: There is only one mutex in `CircuitBreaker`. No lock ordering issue is possible.		
Concurrent Allow + RecordFailure: `Allow()` acquires `mu`, reads state, potentially transitions, returns. `RecordFailure()` also acquires `mu`. If multiple goroutines call `Allow()` concurrently (one probe is allowed through, others rejected), `probeInFlight` prevents multiple probes. The mutex serializes all state access — no TOCTOU window.		

9.2 Rate Limiter

The Lua script executes atomically on the Redis server. No local locking is required in the Go process. Concurrent requests from the same IP for the same route each independently call `Eval` on the Lua script — Redis serializes them server-side. The `*redis.Client` connection pool is goroutine-safe per `go-redis` documentation.

9.3 Reverse Proxy

`httputil.ReverseProxy` is goroutine-safe — each `ServeHTTP` call handles one request independently. The `Director`, `ModifyResponse`, and `ErrorHandler` functions are called within the goroutine serving that request. `RecordSuccess()` and `RecordFailure()` on the circuit breaker are called from these hooks and are protected by the circuit breaker's mutex.

9.4 Notification Consumer

Identical single-goroutine consumer pattern to analytics-service (M4 § 8.1). Prefetch=1 enforces sequential processing. No shared mutable state within the processor goroutine. `*pgxpool.Pool` is goroutine-safe.

9.5 Gateway HTTP Server

`net/http` launches one goroutine per request. All state that handlers read (circuit breaker, rate limiter, proxy targets, JWT verifier) is either:

- Immutable after construction (proxy targets, JWT verifier secret)
- Goroutine-safe via internal locking (circuit breaker mutex, Redis client)
- Goroutine-safe at Redis server level (rate limit counters) No global mutable variables exist in the gateway codebase. {{DIAGRAM:tdd-diag-39}}

10. Implementation Sequence with Checkpoints

Phase 1 — Notification Service: Schema + Consumer + API (2–2.5 hr)

1. Write `migrations/001_create_notifications.sql` (§ 3.1).
2. Apply migration:

```bash
docker exec -i notification_db psql -U notification_user -d notification_db \
  < services/notification-service/migrations/001_create_notifications.sql
```

3. Verify: `docker exec notification_db psql -U notification_user -d notification_db -c "\d notifications"`.
4. Write `repository/notification_repository.go`: `Notification`, `NotificationSummary`, `NotificationRepository` interface, `pgxNotificationRepository`, all four methods.
5. Write `consumer/consumer.go`: `NewAMQPConsumer` — declare exchange (idempotent), declare `notifications.events` queue with three bindings (`url.created`, `url.deleted`, `milestone.reached`), set Qos(1).
6. Write `consumer/processor.go`: `NotificationProcessor`, `Process` following § 5.1.
7. Write `handler/notifications.go`: `HandleListNotifications` (§ 5.7) behind `RequireAuth`.
8. Wire `main.go`: DB connect, repositories, processor, consumer supervisor goroutine, routes.
9. Run `docker compose up --build notification-service`. **Checkpoint:**

```bash
# Trigger a URLCreatedEvent via url-service:
TOKEN=$(curl -s -X POST http://localhost:8083/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"password123"}' | jq -r .token)
curl -s -X POST http://localhost:8081/shorten \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"url":"https://notification-test.example.com"}' > /dev/null
sleep 5  # outbox poller + notification consumer
# Verify notification row:
docker exec notification_db psql -U notification_user -d notification_db \
  -c "SELECT event_type, status, sent_at FROM notifications ORDER BY created_at DESC LIMIT 1;"
# Expected: event_type=url.created, status=sent, sent_at IS NOT NULL
# Check logs for mock email:
docker compose logs notification-service | grep "would send email"
# Expected: at least one "would send email to user" log line
# List notifications via API:
curl -s http://localhost:8084/notifications \
  -H "Authorization: Bearer $TOKEN" | jq .
# Expected: {"notifications":[{"id":"...","event_type":"url.created","status":"sent",...}],"next_cursor":null}
# Wrong user:
TOKEN2=$(curl -s -X POST http://localhost:8083/login \
  -H "Content-Type: application/json" \
  -d '{"email":"bob@example.com","password":"password123"}' | jq -r .token)
curl -s http://localhost:8084/notifications \
  -H "Authorization: Bearer $TOKEN2" | jq '.notifications | length'
# Expected: 0 (bob has no notifications)
```

Phase 2 — shared/logger: Finalize Structured Logger (1–1.5 hr)

1. Finalize `shared/logger/logger.go`: `New`, `WithCorrelationID`, `RequestLogger`, `responseWriter` (§ 4.7, § 5.6).
2. Run `go test ./... -v` in `shared/`.
3. Update all services' `main.go` to use `logger.RequestLogger(log)` as a middleware around the mux.
4. Verify structured output: `docker compose logs url-service 2>&1 | head -5 | python3 -m json.tool`. **Checkpoint:**

```bash
curl -s http://localhost:8081/health > /dev/null

docker compose logs url-service 2>&1 | tail -3 | python3 -m json.tool

# Expected output contains fields: level, time, service, correlation_id, method, path, status, duration_ms, msg

# correlation_id should be present (empty string or UUID if called via gateway)
```

Phase 3 — Gateway: Routing + Reverse Proxy + Correlation Middleware (2–2.5 hr)

1. Add `github.com/google/uuid` to `gateway/go.mod`.
2. Write `middleware/correlation.go`: `CorrelationMiddleware`, `CorrelationIDFromContext` (§ 4.5).
3. Write `proxy/proxy.go`: `ReverseProxy`, `Forward`, `HandlerFor`, director with path stripping, `ModifyResponse`, `ErrorHandler` (§ 4.4).
4. Write `router/router.go`: full route table from § 8 (stubs for JWT + rate limit — wire them as pass-through for now).
5. Write `handler/health.go`: `NewHealthHandler("gateway", log)`.
6. Write `gateway/main.go`: env parsing, startup health checks, server configuration.
7. Run `docker compose up --build gateway`. **Checkpoint:**

```bash
# Direct proxy test:

curl -s http://localhost:8080/health | jq .

# Expected: {"status":"ok","service":"gateway"}

# Auth passthrough (no JWT enforcement yet):

curl -s -X POST http://localhost:8080/api/auth/register \

  -H "Content-Type: application/json" \

  -d '{"email":"charlie@example.com","password":"password123"}' | jq .

# Expected: 201 {"user_id":"...","email":"charlie@example.com"}

# Correlation ID propagation:

curl -v http://localhost:8080/health 2>&1 | grep "X-Correlation-ID"

# Expected: response header X-Correlation-ID: <uuid>

# Pass custom correlation ID:

curl -v -H "X-Correlation-ID: test-corr-id-001" http://localhost:8080/health 2>&1 \

  | grep "X-Correlation-ID"

# Expected: response header X-Correlation-ID: test-corr-id-001 (echoed back)

# Gateway logs show correlation_id: "test-corr-id-001"
```

Phase 4 — Gateway: JWT Middleware (1–1.5 hr)

1. Write `middleware/jwt.go`: `GatewayJWTMiddleware` (§ 4.6).
2. Wire JWT middleware into route table for all `/api/*` routes except `/api/auth/*`.
3. Add `JWT_SECRET` to `gateway/` environment block in `docker-compose.yml`.
4. Write `middleware/middleware_test.go`: JWT middleware tests (§ 11.3).
5. Run `docker compose up --build gateway`. **Checkpoint:**

```bash
# Protected route without JWT:

curl -s -o /dev/null -w "%{http_code}" http://localhost:8080/api/urls \

  -H "Authorization: Bearer invalidtoken"

# Expected: 401

# Auth route without JWT (should pass to user-service):

curl -s -o /dev/null -w "%{http_code}" -X POST http://localhost:8080/api/auth/login \

  -H "Content-Type: application/json" \

  -d '{"email":"alice@example.com","password":"wrongpass"}'

# Expected: 401 (from user-service, not gateway JWT check)

# Protected route with valid JWT:

TOKEN=$(curl -s -X POST http://localhost:8080/api/auth/login \

  -H "Content-Type: application/json" \

  -d '{"email":"alice@example.com","password":"password123"}' | jq -r .token)

curl -s -o /dev/null -w "%{http_code}" http://localhost:8080/api/urls \

  -H "Authorization: Bearer $TOKEN"

# Expected: 200

# X-User-ID header forwarded to downstream:

# Check url-service logs after above request; should contain user_id from JWT claims
```

Phase 5 — Gateway: Redis Rate Limiter (1.5–2 hr)

1. Write `ratelimit/ratelimit.go`: `RateLimiter`, `Allow`, Lua script (§ 4.3).
2. Write `ratelimit/ratelimit_test.go`: unit + integration tests (§ 11.4).
3. Write `middleware/logging.go` (rate limit logging).
4. Wire `RateLimitMiddleware` for `POST /api/shorten` (LimitShorten) and `GET /r/{code}` (LimitRedirect).
5. Run `go test ./ratelimit/... -v`.
6. Run `docker compose up --build gateway`. **Checkpoint:**

```bash
# Rate limit test: POST /api/shorten 11 times from same IP
TOKEN=$(curl -s -X POST http://localhost:8080/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"password123"}' | jq -r .token)
for i in {1..11}; do
  STATUS=$(curl -s -o /dev/null -w "%{http_code}" \
    -X POST http://localhost:8080/api/shorten \
    -H "Authorization: Bearer $TOKEN" \
    -H "Content-Type: application/json" \
    -d "{\"url\":\"https://rate-limit-test.com/$i\"}")
  echo "Request $i: $STATUS"
done
# Expected: requests 1-10 return 201, request 11 returns 429
# Verify Retry-After header on 11th request:
curl -s -o /dev/null -w "%{http_code}\n" -D - \
  -X POST http://localhost:8080/api/shorten \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"url":"https://rate-limit-test.com/12"}' | grep -E "429|Retry-After"
# Expected: 429 + Retry-After: <seconds>
# Redis down fail-open test:
docker compose stop redis
curl -s -o /dev/null -w "%{http_code}" \
  -X POST http://localhost:8080/api/shorten \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"url":"https://redis-down-test.com"}'
# Expected: 201 (not 503 — fail-open)
docker compose start redis
```

Phase 6 — Gateway: Circuit Breaker (1.5–2 hr)

1. Write `circuitbreaker/circuitbreaker.go`: `CircuitBreaker`, `Allow`, `RecordSuccess`, `RecordFailure`, `State` (§ 4.2, § 5.2).
2. Write `circuitbreaker/circuitbreaker_test.go` (§ 11.2).
3. Wire `CircuitBreakerMiddleware` for url-service routes.
4. Run `go test ./circuitbreaker/... -v`.
5. Run `docker compose up --build gateway`. **Checkpoint:**

```bash
# Trip the circuit breaker by making url-service return 5xx:

# Stop url-service to cause connection refused (counts as failure):

docker compose stop url-service

# 5 requests should trip the breaker:

TOKEN=$(curl -s -X POST http://localhost:8080/api/auth/login \

  -H "Content-Type: application/json" \

  -d '{"email":"alice@example.com","password":"password123"}' | jq -r .token)

for i in {1..5}; do

  curl -s -o /dev/null -w "%{http_code} " \

    http://localhost:8080/api/urls -H "Authorization: Bearer $TOKEN"

done

# Expected: 502 502 502 502 502

# 6th request should get circuit breaker response (503):

sleep 1

curl -s http://localhost:8080/api/urls -H "Authorization: Bearer $TOKEN"

# Expected: 503 {"error":"service unavailable"}

# Restart url-service; wait 30s for circuit to probe:

docker compose start url-service

sleep 31  # OpenDuration = 30s

curl -s -o /dev/null -w "%{http_code}" \

  http://localhost:8080/api/urls -H "Authorization: Bearer $TOKEN"

# Expected: 200 (circuit closed after successful probe)
```

Phase 7 — Structured Logging Across All Services (1–1.5 hr)

1. Confirm `shared/logger.RequestLogger` is wired in all five services' `main.go`.
2. Add `WithCorrelationID` call in every handler that accesses request context.
3. Verify `X-Correlation-ID` is forwarded by gateway's `Director` function to all downstream services.
4. Check that async events carry `CorrelationID` through to notification and analytics consumers' log output. **Checkpoint:**

```bash
CORR_ID="e2e-test-$(date +%s)"

TOKEN=$(curl -s -X POST http://localhost:8080/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","password":"password123"}' | jq -r .token)

curl -s -X POST http://localhost:8080/api/shorten \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -H "X-Correlation-ID: $CORR_ID" \
  -d '{"url":"https://e2e-logging-test.com"}' > /dev/null

sleep 5  # outbox + consumers

# Correlation ID must appear in all service logs:
for svc in gateway url-service analytics-service notification-service; do
  COUNT=$(docker compose logs $svc 2>&1 | grep "$CORR_ID" | wc -l)
  echo "$svc: $COUNT log lines with correlation ID"
done

# Expected: gateway >= 1, url-service >= 1, analytics-service >= 1, notification-service >= 1
```

Phase 8 — End-to-End Integration Test + Rate Limit Test (1.5–2 hr)

1. Write `test/e2e_m5_test.go` (§ 11.6).
2. Run full E2E test: `go test -tags integration -v -timeout 120s ./test/...`. **Checkpoint:** All E2E tests pass. `go vet ./...` clean across all services. `go test -race ./...` clean.

---

11. Test Specification

11.1 `repository/notification_repository_test.go` (integration)

```go
// Build tag: //go:build integration

// Requires: notification_db running. DSN from TEST_DATABASE_DSN env var.

// TestInsert_HappyPath: Insert notification → query returns row with status="sent", sent_at IS NOT NULL.

func TestInsert_HappyPath(t *testing.T)

// TestInsert_PayloadPreserved: payload JSONB round-trips correctly.

func TestInsert_PayloadPreserved(t *testing.T)

// TestListByUser_OnlyOwnerRows: insert notifications for two users; ListByUser(user1) returns only user1's.

func TestListByUser_OnlyOwnerRows(t *testing.T)

// TestListByUser_Pagination: insert 5 notifications for same user; page by 2; cursor chain covers all 5.

func TestListByUser_Pagination(t *testing.T)

// TestListByUser_EmptyCursor: first page returns newest first.

func TestListByUser_EmptyCursor(t *testing.T)

// TestIsProcessed_NotFound: unknown eventKey returns false, nil.

func TestIsProcessed_NotFound(t *testing.T)

// TestMarkProcessed_Idempotent: two calls with same eventKey → no error.

func TestMarkProcessed_Idempotent(t *testing.T)

// TestIsProcessed_AfterMark: IsProcessed returns true after MarkProcessed.

func TestIsProcessed_AfterMark(t *testing.T)
```

11.2 `circuitbreaker/circuitbreaker_test.go`

```go
// All tests are pure unit tests; no external dependencies.

// TestNewCircuitBreaker_InitialState: state is Closed; failureCount is 0.
func TestNewCircuitBreaker_InitialState(t *testing.T)

// TestAllow_Closed_AlwaysReturnsTrue: 100 concurrent Allow() calls on Closed CB all return true.
func TestAllow_Closed_AlwaysReturnsTrue(t *testing.T)

// TestTrip_AfterThresholdFailures: 5 RecordFailure() within window → State() returns Open.
func TestTrip_AfterThresholdFailures(t *testing.T)

// TestAllow_Open_ReturnsError: after trip, Allow() returns (false, ErrCircuitOpen).
func TestAllow_Open_ReturnsError(t *testing.T)

// TestHalfOpen_AfterOpenDuration: after OpenDuration elapses, Allow() transitions to HalfOpen.
// Use a Config with OpenDuration=100ms for fast tests.
func TestHalfOpen_AfterOpenDuration(t *testing.T)

// TestHalfOpen_OnlyOneProbeAllowed: two concurrent Allow() in HalfOpen → one returns true, one false.
func TestHalfOpen_OnlyOneProbeAllowed(t *testing.T)

// TestHalfOpen_ProbeSuccess_TransitionsClosed: RecordSuccess() in HalfOpen → State() returns Closed.
func TestHalfOpen_ProbeSuccess_TransitionsClosed(t *testing.T)

// TestHalfOpen_ProbeFailure_TransitionsOpen: RecordFailure() in HalfOpen → State() returns Open.
func TestHalfOpen_ProbeFailure_TransitionsOpen(t *testing.T)

// TestWindowReset: 3 failures, then gap > Window, then 5 more failures → trips on 5th (not 3rd).
func TestWindowReset(t *testing.T)

// TestRecordSuccess_Closed_ResetsCount: 4 failures + 1 success + 4 failures → no trip (count reset).
func TestRecordSuccess_Closed_ResetsCount(t *testing.T)

// TestConcurrentAccessNoPanic: 100 goroutines concurrently call Allow/RecordSuccess/RecordFailure.
// Run with go test -race. Expected: no panic, no data race.
func TestConcurrentAccessNoPanic(t *testing.T)

// TestState_AtomicRead: goroutine-safe read via State() while another goroutine transitions.
func TestState_AtomicRead(t *testing.T)

// BenchmarkAllow_Closed: < 1µs per call.
func BenchmarkAllow_Closed(b *testing.B)

// BenchmarkAllow_Open: < 1µs per call (mu + time check only).
func BenchmarkAllow_Open(b *testing.B)
```

11.3 `middleware/middleware_test.go`

```go
// All tests use httptest.NewRecorder + httptest.NewRequest.

// TestCorrelationMiddleware_GeneratesIDWhenAbsent: no X-Correlation-ID in request →

// response header has UUID v4; context contains same ID.

func TestCorrelationMiddleware_GeneratesIDWhenAbsent(t *testing.T)

// TestCorrelationMiddleware_PropagatesExisting: X-Correlation-ID: "abc" →

// response header echoes "abc"; context contains "abc".

func TestCorrelationMiddleware_PropagatesExisting(t *testing.T)

// TestCorrelationIDFromContext_Present: returns stored ID and true.

func TestCorrelationIDFromContext_Present(t *testing.T)

// TestCorrelationIDFromContext_Absent: returns "", false.

func TestCorrelationIDFromContext_Absent(t *testing.T)

// TestGatewayJWT_ValidToken_CallsNext: valid token → next handler called, 200.

func TestGatewayJWT_ValidToken_CallsNext(t *testing.T)

// TestGatewayJWT_InvalidToken_Returns401: invalid token → 401, next NOT called.

func TestGatewayJWT_InvalidToken_Returns401(t *testing.T)

// TestGatewayJWT_MissingHeader_Returns401: no Authorization header → 401.

func TestGatewayJWT_MissingHeader_Returns401(t *testing.T)

// TestGatewayJWT_ExpiredToken_Returns401: expired token → 401.

func TestGatewayJWT_ExpiredToken_Returns401(t *testing.T)

// TestGatewayJWT_AuthRouteExempt: POST /api/auth/register → next called without JWT.

func TestGatewayJWT_AuthRouteExempt(t *testing.T)

// TestGatewayJWT_PublicRedirect_Exempt: GET /r/abc → next called without JWT.

func TestGatewayJWT_PublicRedirect_Exempt(t *testing.T)

// TestGatewayJWT_ForwardsUserHeaders: valid token → X-User-ID and X-User-Email set on forwarded request.

func TestGatewayJWT_ForwardsUserHeaders(t *testing.T)
```

11.4 `ratelimit/ratelimit_test.go`

```go
// Unit tests use a nil Redis client to test fail-open behavior.

// Integration tests use TEST_REDIS_ADDR env var; build tag: //go:build integration.

// TestAllow_NilClient_FailOpen: nil *redis.Client → AllowResult.Allowed = true.

func TestAllow_NilClient_FailOpen(t *testing.T)

// TestAllow_RedisError_FailOpen: Redis returns error (use mock) → AllowResult.Allowed = true.

func TestAllow_RedisError_FailOpen(t *testing.T)

// TestAllow_UnderLimit: calls 1..limit all return Allowed=true.

func TestAllow_UnderLimit(t *testing.T) // integration

// TestAllow_ExactlyAtLimit: call number limit returns Allowed=true; call limit+1 returns false.

func TestAllow_ExactlyAtLimit(t *testing.T) // integration

// TestAllow_RetryAfterPositive: when Allowed=false, RetryAfter > 0.

func TestAllow_RetryAfterPositive(t *testing.T) // integration

// TestAllow_DifferentIPs_Independent: IP A at limit, IP B still allowed.

func TestAllow_DifferentIPs_Independent(t *testing.T) // integration

// TestAllow_DifferentRoutes_Independent: shorten at limit, redirect still allowed for same IP.

func TestAllow_DifferentRoutes_Independent(t *testing.T) // integration

// TestAllow_WindowExpiry: fill window, wait for TTL to expire, fill again → all allowed.

// Use RouteLimit with Window=2s for fast test.

func TestAllow_WindowExpiry(t *testing.T) // integration

// TestAllow_ConcurrentRequests_Atomic: 50 concurrent Allow() calls for same IP+route;

// exactly limit allowed, rest rejected. Verify atomicity via Redis counter.

func TestAllow_ConcurrentRequests_Atomic(t *testing.T) // integration + race detector

// BenchmarkAllow: measure Redis round-trip < 3ms p99.

func BenchmarkAllow(b *testing.B)
```

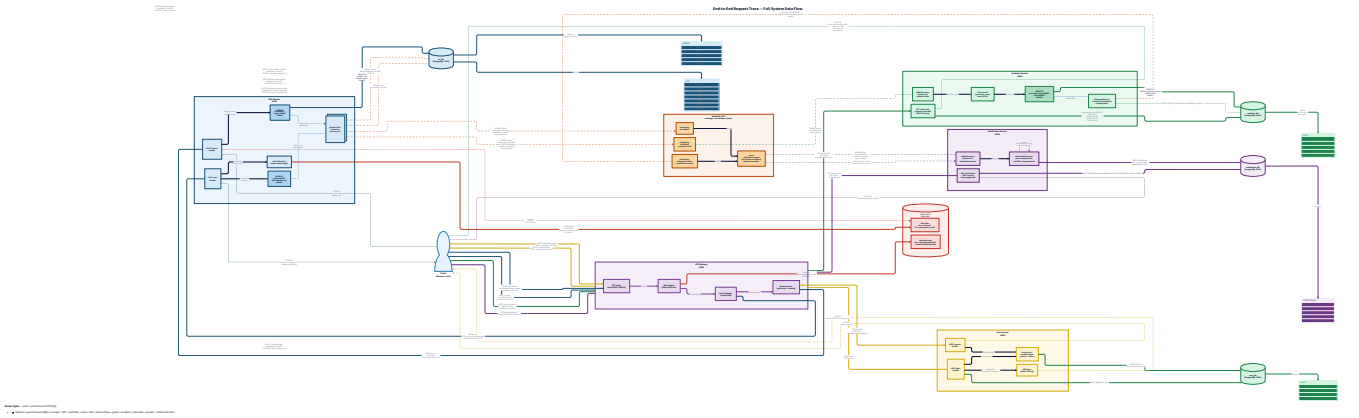
11.5 `consumer/processor_test.go` (Notification Service)

```go
// Unit tests use mock NotificationRepository.

// TestProcess_URLCreated_HappyPath: URLCreatedEvent → ProcessResultInserted, Insert called.

func TestProcess_URLCreated_HappyPath(t *testing.T)

// TestProcess_URLDeleted_HappyPath: URLDeletedEvent → ProcessResultInserted.

func TestProcess_URLDeleted_HappyPath(t *testing.T)

// TestProcess_MilestoneReached_HappyPath: MilestoneReachedEvent → ProcessResultInserted.

func TestProcess_MilestoneReached_HappyPath(t *testing.T)

// TestProcess_MalformedJSON: invalid JSON → ProcessResultPoisoned; Insert NOT called.

func TestProcess_MalformedJSON(t *testing.T)

// TestProcess_UnknownRoutingKey: routing_key="unknown.event" → ProcessResultPoisoned.

func TestProcess_UnknownRoutingKey(t *testing.T)

// TestProcess_DuplicateEvent: IsProcessed returns true → ProcessResultDuplicate; Insert NOT called.

func TestProcess_DuplicateEvent(t *testing.T)

// TestProcess_InsertFail: Insert returns error → ProcessResultError.

func TestProcess_InsertFail(t *testing.T)

// TestProcess_MockEmailLogLine: after Insert, log output contains "would send email to user".

// Capture zerolog output via bytes.Buffer writer.

func TestProcess_MockEmailLogLine(t *testing.T)

// TestProcess_PanicRecovery: mock Insert panics → returns ProcessResultError; no propagated panic.

func TestProcess_PanicRecovery(t *testing.T)
```

11.6 End-to-End Integration Test (`test/e2e_m5_test.go`)

```go
// Build tag: //go:build integration

// Requires: full docker compose stack running.

// BASE_URL env var: "http://localhost:8080"

// TestE2E_FullFlow:

//   1. POST /api/auth/register → 201 {user_id, email}

//   2. POST /api/auth/login → 200 {token, expires_at}

//   3. POST /api/shorten → 201 {short_code, short_url, ...}

//   4. GET /r/{code} (no auth) → 301; follow redirect → external URL

//   5. Wait 5s for outbox + consumers

//   6. GET /api/stats/{code} (no auth) → 200 {total_clicks: 1, ...}

//   7. GET /api/notifications (with token) → 200, contains url.created event

//   8. Verify X-Correlation-ID in response headers matches request header

func TestE2E_FullFlow(t *testing.T)

// TestE2E_RateLimitShorten:

//   Register/login user. POST /api/shorten 11 times from same IP.

//   Requests 1-10: 201. Request 11: 429 with Retry-After header.

//   Verify Retry-After is integer string, > 0.

func TestE2E_RateLimitShorten(t *testing.T)

// TestE2E_CircuitBreaker:

//   (Requires ability to stop url-service; skip in CI if not possible.)

//   Stop url-service. 5 requests → 502. 6th → 503 {"error":"service unavailable"}.

//   Start url-service, wait 31s. Request → 200 (circuit healed).

func TestE2E_CircuitBreaker(t *testing.T)

// TestE2E_JWTGatewayEnforcement:

//   GET /api/urls without token → 401 from gateway (not 401 from url-service).

//   Verify response body is gateway's {"error":"unauthorized"} not url-service's.

//   Distinguish: gateway responds in < 2ms (no proxy overhead); url-service in > 5ms.

func TestE2E_JWTGatewayEnforcement(t *testing.T)

// TestE2E_CorrelationIDPropagation:

//   Send request with X-Correlation-ID: "e2e-test-abc".

//   Verify response X-Correlation-ID: "e2e-test-abc".

//   Check logs: gateway, url-service logs both contain "e2e-test-abc".

func TestE2E_CorrelationIDPropagation(t *testing.T)

// TestE2E_NotificationMilestone:

//   Generate 10 clicks on same short_code (via /r/{code}).

//   Wait 10s.

//   GET /api/notifications → contains milestone.reached event.

//   GET /api/stats/{code} → total_clicks = 10.

func TestE2E_NotificationMilestone(t *testing.T)
```

```
// TestE2E_PublicEndpoints_NoAuth:
//   GET /r/{code} → 301 (no auth required, not 401).
//   GET /api/stats/{code} → 200 (no auth required).
//   GET /health → 200.
func TestE2E_PublicEndpoints_NoAuth(t *testing.T)
```



12. Environment Variable Reference

Notification Service

Variable	Required	Example	Description
`PORT`	Yes	`8084`	HTTP listen port
`SERVICE_NAME`	Yes	`notification-service`	Embedded in log lines
`DATABASE_DSN`	Yes	`postgres://notification_user:notification_secret@notification_db:5432/notification_db?sslmode=disable`	notification_db connection
`RABBITMQ_URL`	Yes	`amqp://admin:admin@rabbitmq:5672/`	AMQP broker
`JWT_SECRET`	Yes	`change-me-to-at-least-32-random-bytes`	JWT verification for `GET /notifications`

API Gateway

Variable	Required	Example	Description
`PORT`	Yes	`8080`	HTTP listen port
`SERVICE_NAME`	Yes	`gateway`	Embedded in log lines
`JWT_SECRET`	Yes	`change-me-to-at-least-32-random-bytes`	JWT verification (same as all services)
`REDIS_ADDR`	Yes	`redis:6379`	Redis for rate limit counters; unavailability = fail-open
`URL_SERVICE_URL`	Yes	`http://url-service:8081`	Downstream target
`ANALYTICS_SERVICE_URL`	Yes	`http://analytics-service:8082`	Downstream target
`USER_SERVICE_URL`	Yes	`http://user-service:8083`	Downstream target
`NOTIFICATION_SERVICE_URL`	Yes	`http://notification-service:8084`	Downstream target
`LOG_LEVEL`	No	`info`	`debug|info|warn|error`; default `info`
Add to `docker-compose.yml` for notification-service and gateway:			

```yaml
# notification-service environment additions:
JWT_SECRET: "change-me-to-at-least-32-random-bytes"
# gateway environment additions:
JWT_SECRET: "change-me-to-at-least-32-random-bytes"
# (REDIS_ADDR already present from M1)
```

`JWT_SECRET` must be identical across all five services. Use a shared Docker Compose variable or `.env` file to prevent drift:

```yaml
# docker-compose.yml (add at top level)
x-common-jwt: &common-jwt
  JWT_SECRET: "${JWT_SECRET:-change-me-to-at-least-32-random-bytes}"
# Apply in each service's environment:
environment:
  <<: *common-jwt
```

13. Performance Targets

Operation	Target	How to Measure
Gateway proxy overhead (added latency)	< 2ms p99	Compare `GET /r/{code}` via gateway (8080) vs direct url-service (8081); delta is proxy overhead
Rate limit check (Redis INCR Lua)	< 3ms p99	`go test -bench=BenchmarkAllow ./ratelimit/...` with real Redis
Circuit breaker `Allow()` state check	< 1µs	`go test -bench=BenchmarkAllow_Closed ./circuitbreaker/...`
`GET /api/notifications` p99	< 20ms	`wrk -t2 -c10 -d30s` with valid JWT + 100 notification rows
Correlation ID generation (`uuid.New()`)	< 1µs	`go test -bench=BenchmarkUUID ./middleware/...`
Gateway startup to all-healthy	< 90s total stack	`time docker compose up --wait`
Notification consumer lag (event → DB insert)	< 2s p99	Measure `notifications.created_at` vs `URLCreatedEvent.OccurredAt` delta
`GET /health` on all services	< 10ms p99	`wrk -t2 -c10 -d10s http://localhost:8080/health`
Rate limit test (11 shorten requests)	1-10: 201; 11: 429	Scripted `curl` loop per Phase 8 checkpoint
Circuit breaker trip (5 failures)	< 500ms elapsed	Measure time from first failure to first 503 via gateway
Circuit breaker half-open probe	30s ± 1s	Measure time from Open → first 200 after url-service restarts
Gateway JWT verification	< 1ms (local HMAC)	No DB call; bounded by CPU; benchmark HMAC-SHA256

14. Anti-Pattern Guard Rail Checklist

Before completing this milestone, verify:

- ☐ **No business logic in the gateway.** `grep -r "short_code\|analytics\|milestone\|password" gateway/` returns only forwarding/proxy code, never domain computation.
- ☐ **Circuit breaker has three states, not a retry loop.** `grep -r "time.Sleep\|for.*retry" gateway/circuitbreaker/` returns nothing. The state machine is `State` enum with mutex-protected transitions.
- ☐ **Notification service does not call user-service.** `grep -r "user-service\|8083" services/notification-service/` returns nothing except comments. `UserEmail` comes from the event payload.
- ☐ **Rate limit counters in Redis, not process memory.** `grep -r "sync.Map\|var.*counter\|atomic.Int" gateway/ratelimit/` returns nothing. All counters are Redis keys.
- ☐ **Redis failure allows requests (fail-open).** `grep -A5 "redis.*err\|nil.*client" gateway/ratelimit/ratelimit.go` shows `return AllowResult{Allowed: true}` on error.
- ☐ **`X-Correlation-ID` propagated to async events.** `grep "CorrelationID" shared/events/events.go` shows the field on all event structs. `grep "CorrelationID" services/notification-service/consumer/processor.go` shows it is logged.
- ☐ **Gateway JWT middleware does not call user-service.** `grep -r "user-service\|8083" gateway/middleware/jwt.go` returns nothing. Only `sharedauth.NewJWTVerifier` and its `Verify` method are used.
- ☐ **Circuit breaker `probeInFlight` prevents concurrent probes.** `grep "probeInFlight" gateway/circuitbreaker/circuitbreaker.go` shows it is set before returning `true` from HalfOpen and cleared in both `RecordSuccess` and `RecordFailure`.
- ☐ **Downstream 5xx counts as circuit breaker failure.** `grep "RecordFailure\|StatusCode >= 500" gateway/proxy/proxy.go` shows `ModifyResponse` calls `RecordFailure` for 5xx responses.
- ☐ **`notifications` response never exposes raw payload.** `NotificationItem` struct has no `payload` or `data` field — only `id`, `event_type`, `status`, `created_at`, `sent_at`.
- ☐ **All log lines have `correlation_id` field.** `docker compose logs url-service 2>&1 | python3 -c "import sys,json; [json.loads(l) for l in sys.stdin if l.strip()]"` — all lines parse as JSON with `correlation_id` key present.

Project Structure: URL Shortener (Microservices)

Directory Tree

```
url-shortener/                                  # repo root
│
├── docker-compose.yml                          # M1: full 11-container topology
├── .env.example                                # M1: all env vars documented
├── README.md                                   # M1: setup commands
│
├── shared/                                     # imported by all services
│   ├── go.mod                                  # M1: module github.com/.../shared
│   ├── go.sum                                  # M1: auto-generated
│   ├── events/
│   │   └── events.go                           # M1: DomainEvent interface + 4 event structs
│   ├── auth/
│   │   └── types.go                            # M2: JWTVerifier interface, Claims struct, sentinel errors
│   ├── middleware/
│   │   ├── jwt_middleware.go                   # M2: RequireAuth http.Handler wrapper
│   │   └── jwt_middleware_test.go              # M2: middleware unit tests
│   └── logger/
│       └── logger.go                           # M1 stub → M5 finalized: New, WithCorrelationID, RequestLogger
│
├── services/
│   │
│   ├── url-service/                            # M1 scaffold → M3 full implementation
│   │   ├── go.mod                              # M1: module + replace shared
│   │   ├── go.sum                              # M1: auto-generated
│   │   ├── Dockerfile                          # M1: multi-stage Alpine build
│   │   ├── main.go                             # M1 stub → M3: wiring env/DB/Redis/RabbitMQ/routes/outbox
│   │   ├── migrations/
│   │   │   ├── 001_create_urls.sql             # M3: urls table + indexes
│   │   │   └── 002_create_outbox.sql           # M3: outbox table + partial index
│   │   ├── codegen/
│   │   │   ├── codegen.go                      # M3: base62 alphabet, Generate(), CodeGenerator interface
│   │   │   └── codegen_test.go                 # M3: length/alphabet/randomness/no-math-rand tests
│   │   ├── repository/
│   │   │   ├── url_repository.go               # M3: URL/URLSummary structs, URLRepository interface + pgx impl
│   │   │   ├── url_repository_test.go          # M3: integration tests (conflict, cursor, ownership)
│   │   │   ├── outbox_repository.go            # M3: OutboxEntry, OutboxRepository interface + pgx impl
│   │   │   └── outbox_repository_test.go       # M3: insert/rollback/fetch/mark tests
│   │   ├── cache/
│   │   │   ├── cache.go                        # M3: CachedURL, CacheClient interface, nil-safe Redis impl
│   │   │   └── cache_test.go                   # M3: nil-client/TTL/set-get/del tests
│   │   ├── amqp/
│   │   │   ├── publisher.go                    # M3: AMQPPublisher interface + amqp091 impl with sync.Mutex
│   │   │   └── publisher_test.go               # M3: publish tests
│   │   ├── outbox/
│   │   │   ├── poller.go                       # M3: OutboxPoller coordinator + 3 workers
│   │   │   └── poller_test.go                  # M3: worker pool / drain tests
│   │   ├── service/
│   │   │   └── url_service.go                  # M3: URLService orchestrating repo/codegen/cache/outbox
│   │   ├── auth/
│   │   │   ├── password.go                     # M2 (used by url-service for JWT verify only): bcrypt impl
│   │   │   ├── password_test.go                # M2: hash/compare/cost tests
│   │   │   ├── jwt.go                         # M2: JWTSigner + JWTVerifier implementations
│   │   │   └── jwt_test.go                     # M2: sign/verify/expiry/tamper tests
│   │   └── handler/
│   │       ├── health.go                       # M1: GET /health
│   │       ├── shorten.go                      # M3: POST /shorten
│   │       ├── redirect.go                     # M3: GET /{code} 301 redirect
│   │       ├── list.go                         # M3: GET /urls cursor pagination
│   │       ├── delete.go                       # M3: DELETE /urls/{code}
│   │       ├── helpers.go                      # M3: writeJSON, writeError, extractClientIP, hashIP, ptr
│   │       └── handler_test.go                 # M3: unit + integration tests + benchmarks
│   │
│   ├── analytics-service/                      # M1 scaffold → M4 full implementation
│   │   ├── go.mod                              # M1: module + replace shared
│   │   ├── go.sum                              # M1: auto-generated
│   │   ├── Dockerfile                          # M1: multi-stage Alpine build
│   │   ├── main.go                             # M1 stub → M4: wiring DB/RabbitMQ/consumer/routes
│   │   ├── migrations/
│   │   │   ├── 001_create_clicks.sql           # M4: clicks table + two indexes (incl. partial)
│   │   │   ├── 002_create_milestones.sql       # M4: milestones table + unique (short_code, milestone)
│   │   │   └── 003_create_processed_events.sql # M4: processed_events dedup table (UUID PK)
│   │   ├── repository/
│   │   │   ├── click_repository.go             # M4: Click/RefererCount/PeriodCount, ClickRepository + pgx impl
│   │   │   ├── click_repository_test.go        # M4: insert/count/referer/timeline integration tests
│   │   │   ├── processed_event_repository.go   # M4: ProcessedEvent, ProcessedEventRepository + pgx impl
│   │   │   ├── processed_event_repository_test.go # M4: is-processed/mark/idempotent/rollback tests
│   │   │   ├── milestone_repository.go         # M4: Milestone, MilestoneRepository + pgx impl
```

```
│   │   │   └── milestone_repository_test.go      # M4: get-latest/insert/conflict tests
│   │   ├── consumer/
│   │   │   ├── consumer.go                       # M4: AMQPConsumer setup (exchange+queue+bind+Qos+Consume)
│   │   │   ├── processor.go                      # M4: ClickProcessor with full idempotency algorithm
│   │   │   └── processor_test.go                 # M4: unit (mock repos) + integration duplicate-event test
│   │   ├── milestone/
│   │   │   ├── detector.go                       # M4: Thresholds [10,100,1000], Detector interface + impl
│   │   │   └── detector_test.go                  # M4: all threshold transition table-driven tests
│   │   ├── publisher/
│   │   │   ├── publisher.go                      # M4: AMQPPublisher for MilestoneReachedEvent
│   │   │   └── publisher_test.go                 # M4: publish tests
│   │   └── handler/
│   │       ├── health.go                         # M1: GET /health
│   │       ├── stats.go                          # M4: GET /stats/{code} with errgroup concurrent queries
│   │       ├── timeline.go                       # M4: GET /stats/{code}/timeline
│   │       ├── helpers.go                        # M4: writeJSON, writeError
│   │       └── handler_test.go                   # M4: unit (mock repos) + benchmarks
│   │
│   ├── user-service/                             # M1 scaffold → M2 full implementation
│   │   ├── go.mod                                # M1: module + replace shared
│   │   ├── go.sum                                # M1: auto-generated
│   │   ├── Dockerfile                            # M1: multi-stage Alpine build
│   │   ├── main.go                               # M1 stub → M2: wiring DB/auth/routes/middleware
│   │   ├── migrations/
│   │   │   └── 001_create_users.sql              # M2: users table (UUID PK, UNIQUE email, bcrypt hash)
│   │   ├── repository/
│   │   │   ├── user_repository.go                # M2: User struct, UserRepository interface + pgx impl
│   │   │   └── user_repository_test.go           # M2: create/duplicate/find/case-sensitivity tests
│   │   ├── auth/
│   │   │   ├── password.go                       # M2: PasswordHasher interface + bcryptHasher (cost=12)
│   │   │   ├── password_test.go                  # M2: hash/compare/cost/benchmark tests
│   │   │   ├── jwt.go                            # M2: JWTSigner + JWTVerifier (implements shared/auth.JWTVerifier)
│   │   │   └── jwt_test.go                       # M2: sign/verify/expiry/issuer/tamper tests
│   │   ├── service/
│   │   │   └── user_service.go                   # M2: UserService (Register/Login + dummyHash timing equalization)
│   │   └── handler/
│   │       ├── health.go                         # M1: GET /health
│   │       ├── register.go                       # M2: POST /register
│   │       ├── login.go                          # M2: POST /login
│   │       ├── me.go                             # M2: GET /me (behind RequireAuth)
│   │       ├── helpers.go                        # M2: writeJSON, writeInternalError, ErrorResponse
│   │       └── handler_test.go                   # M2: integration round-trip + edge case tests
│   │
│   └── notification-service/                     # M1 scaffold → M5 full implementation
│       ├── go.mod                                # M1: module + replace shared
│       ├── go.sum                                # M1: auto-generated
│       ├── Dockerfile                            # M1: multi-stage Alpine build
│       ├── main.go                               # M1 stub → M5: wiring DB/RabbitMQ/consumer/routes/JWT
│       ├── migrations/
│       │   └── 001_create_notifications.sql      # M5: notifications + processed_notifications tables
│       ├── repository/
│       │   ├── notification_repository.go        # M5: Notification/NotificationSummary, NotificationRepository + pgx impl
│       │   └── notification_repository_test.go   # M5: insert/list/cursor/dedup integration tests
│       ├── consumer/
│       │   ├── consumer.go                       # M5: AMQPConsumer (3 bindings: url.created/deleted/milestone.reached)
│       │   ├── processor.go                      # M5: NotificationProcessor with routing-key switch + mock email log
│       │   └── processor_test.go                 # M5: unit tests for all event types + panic recovery
│       └── handler/
│           ├── health.go                         # M1: GET /health
│           ├── notifications.go                  # M5: GET /notifications (JWT-protected, cursor-paginated)
│           ├── helpers.go                        # M5: writeJSON, writeError
│           └── handler_test.go                   # M5: list/pagination/ownership tests
│
├── gateway/                                      # M1 scaffold → M5 full implementation
│   ├── go.mod                                    # M1: module + replace shared
│   ├── go.sum                                    # M1: auto-generated
│   ├── Dockerfile                                # M1: multi-stage Alpine build
│   ├── main.go                                   # M1 stub → M5: full wiring env/Redis/CB/proxy/routes/server
│   ├── circuitbreaker/
│   │   ├── circuitbreaker.go                     # M5: CircuitBreaker 3-state machine (Closed/Open/HalfOpen)
│   │   └── circuitbreaker_test.go                # M5: state transitions/concurrency/window-reset tests
│   ├── ratelimit/
│   │   ├── ratelimit.go                          # M5: RateLimiter — Redis Lua INCR+EXPIRE, fail-open
│   │   └── ratelimit_test.go                     # M5: unit (nil client) + integration (window/atomic) tests
│   ├── proxy/
│   │   ├── proxy.go                              # M5: ReverseProxy wrapper with Director path-strip + CB hooks
│   │   └── proxy_test.go                         # M5: forward/error-handling tests
│   ├── middleware/
│   │   ├── correlation.go                        # M5: CorrelationMiddleware — X-Correlation-ID inject/propagate
│   │   ├── jwt.go                                # M5: GatewayJWTMiddleware — local HMAC verify, exempt routes
│   │   ├── logging.go                            # M5: RequestLogger middleware (wraps shared/logger)
```

```
│   │   └── middleware_test.go                  # M5: correlation/JWT/exempt-route tests
│   ├── router/
│   │   ├── router.go                           # M5: full route table (all 12 routes with middleware chains)
│   │   └── router_test.go                      # M5: route matching + middleware wiring tests
│   └── handler/
│       ├── health.go                           # M1: GET /health
│       └── handler_test.go                     # M5: health handler test
│
└── test/
    ├── smoke_m1.sh                             # M1: shell smoke test for all 5 /health endpoints
    └── e2e_m5_test.go                          # M5: full end-to-end integration tests (Go, build tag: integration)
```

Creation Order

1. Monorepo Scaffold (30–45 min)

Create all directories first, then stub files:

```bash
mkdir -p shared/{events,auth,middleware,logger}

mkdir -p services/{url-service,analytics-service,user-service,notification-service}

mkdir -p services/url-service/{migrations,codegen,repository,cache,amqp,outbox,service,auth,handler}

mkdir -p services/analytics-service/{migrations,repository,consumer,milestone,publisher,handler}

mkdir -p services/user-service/{migrations,repository,auth,service,handler}

mkdir -p services/notification-service/{migrations,repository,consumer,handler}

mkdir -p gateway/{circuitbreaker,ratelimit,proxy,middleware,router,handler}

mkdir -p test
```

- `shared/go.mod`
- `services/url-service/go.mod`
- `services/analytics-service/go.mod`
- `services/user-service/go.mod`
- `services/notification-service/go.mod`
- `gateway/go.mod`
- Empty `main.go` stubs in each service and gateway
- Run `go mod tidy` in each directory

2. Shared Foundation — M1 (45–60 min)

- `shared/events/events.go` — all 4 event structs + constants + DomainEvent interface
- `shared/logger/logger.go` — initial stub: `New()` only
- `docker-compose.yml` — full 11-container topology
- `.env.example` — all environment variables
- `README.md`

3. Dockerfiles — M1 (30 min)

- `services/url-service/Dockerfile`
- `services/analytics-service/Dockerfile`
- `services/user-service/Dockerfile`
- `services/notification-service/Dockerfile`
- `gateway/Dockerfile`

4. Health Handlers + Infra Bootstrap — M1 (60–90 min)

- `services/url-service/handler/health.go`
- `services/analytics-service/handler/health.go`
- `services/user-service/handler/health.go`
- `services/notification-service/handler/health.go`

- `gateway/handler/health.go`
- Full `main.go` for each service: `ConnectDB`, `ConnectRedis` (url-service + gateway), `ConnectRabbitMQ`, `mustGetEnv`, startup sequence, `/health` route, RabbitMQ queue declarations
- `test/smoke_m1.sh` **Checkpoint:** `docker compose up --build` → all 11 containers healthy. All `/health` endpoints return 200.

5. User Service Schema + Repository — M2 (60–90 min)

- `services/user-service/migrations/001_create_users.sql`
- `services/user-service/repository/user_repository.go`
- `services/user-service/repository/user_repository_test.go`

6. User Service Auth Layer — M2 (60–90 min)

- `services/user-service/auth/password.go`
- `services/user-service/auth/password_test.go`
- `services/user-service/auth/jwt.go`
- `services/user-service/auth/jwt_test.go`

7. Shared Auth Types — M2 (20 min)

- `shared/auth/types.go` — JWTVerifier interface, Claims struct, sentinel errors

8. User Service Business Logic + Handlers — M2 (90–120 min)

- `services/user-service/service/user_service.go`
- `services/user-service/handler/helpers.go`
- `services/user-service/handler/register.go`
- `services/user-service/handler/login.go`
- `services/user-service/handler/me.go`
- `services/user-service/handler/handler_test.go`
- Update `services/user-service/main.go` with full M2 wiring

9. Shared JWT Middleware — M2 (45–60 min)

- `shared/middleware/jwt_middleware.go`
- `shared/middleware/jwt_middleware_test.go` **Checkpoint:** Register → Login → GET /me round-trip works. Duplicate email → 409. Wrong password → 401 (same body as unknown email).

10. URL Service Schema + Code Generator — M3 (60–90 min)

- `services/url-service/migrations/001_create_urls.sql`
- `services/url-service/migrations/002_create_outbox.sql`
- `services/url-service/codegen/codegen.go`
- `services/url-service/codegen/codegen_test.go`

11. URL Service Repository Layer — M3 (90–120 min)

- `services/url-service/repository/url_repository.go`
- `services/url-service/repository/url_repository_test.go`
- `services/url-service/repository/outbox_repository.go`
- `services/url-service/repository/outbox_repository_test.go`

12. URL Service Cache + AMQP + Service Layer — M3 (90–120 min)

- `services/url-service/cache/cache.go`
- `services/url-service/cache/cache_test.go`
- `services/url-service/amqp/publisher.go`
- `services/url-service/amqp/publisher_test.go`
- `services/url-service/service/url_service.go`

13. URL Service Handlers + Outbox Poller — M3 (90–120 min)

- `services/url-service/handler/helpers.go`

- `services/url-service/handler/shorten.go`
- `services/url-service/handler/redirect.go`
- `services/url-service/handler/list.go`
- `services/url-service/handler/delete.go`
- `services/url-service/handler/handler_test.go`
- `services/url-service/outbox/poller.go`
- `services/url-service/outbox/poller_test.go`
- Update `services/url-service/main.go` with full M3 wiring **Checkpoint:** POST /shorten → 201. GET /:code → 301. DELETE /urls/:code → 204. Outbox poller marks rows published within 4s. RabbitMQ management shows `url-shortener` exchange.

14. Analytics Service Schema + Repository — M4 (90–120 min)

- `services/analytics-service/migrations/001_create_clicks.sql`
- `services/analytics-service/migrations/002_create_milestones.sql`
- `services/analytics-service/migrations/003_create_processed_events.sql`
- `services/analytics-service/repository/click_repository.go`
- `services/analytics-service/repository/click_repository_test.go`
- `services/analytics-service/repository/processed_event_repository.go`
- `services/analytics-service/repository/processed_event_repository_test.go`
- `services/analytics-service/repository/milestone_repository.go`
- `services/analytics-service/repository/milestone_repository_test.go`

15. Analytics Service Consumer + Milestone Detection — M4 (90–120 min)

- `services/analytics-service/consumer/consumer.go`
- `services/analytics-service/consumer/processor.go`
- `services/analytics-service/consumer/processor_test.go`
- `services/analytics-service/milestone/detector.go`
- `services/analytics-service/milestone/detector_test.go`
- `services/analytics-service/publisher/publisher.go`
- `services/analytics-service/publisher/publisher_test.go`

16. Analytics Service Handlers — M4 (60–90 min)

- `services/analytics-service/handler/helpers.go`
- `services/analytics-service/handler/stats.go`
- `services/analytics-service/handler/timeline.go`
- `services/analytics-service/handler/handler_test.go`
- Update `services/analytics-service/main.go` with full M4 wiring **Checkpoint:** Make 10 redirects → GET /stats/:code shows total_clicks=10. Milestone row with milestone=10 in analytics_db. MilestoneReachedEvent visible in notifications.events queue.

17. Notification Service Schema + Repository — M5 (60–90 min)

- `services/notification-service/migrations/001_create_notifications.sql`
- `services/notification-service/repository/notification_repository.go`
- `services/notification-service/repository/notification_repository_test.go`

18. Notification Service Consumer + Handlers — M5 (90–120 min)

- `services/notification-service/consumer/consumer.go`
- `services/notification-service/consumer/processor.go`
- `services/notification-service/consumer/processor_test.go`
- `services/notification-service/handler/helpers.go`
- `services/notification-service/handler/notifications.go`
- `services/notification-service/handler/handler_test.go`
- Update `services/notification-service/main.go` with full M5 wiring

19. Shared Logger Finalization — M5 (45–60 min)

- Update `shared/logger/logger.go` — add `WithCorrelationID`, `RequestLogger`, `responseWriter`

- Update all 5 services' `main.go` to wrap mux with `logger.RequestLogger`

20. Gateway Middleware — M5 (60–90 min)

- `gateway/middleware/correlation.go`
- `gateway/middleware/jwt.go`
- `gateway/middleware/logging.go`
- `gateway/middleware/middleware_test.go`

21. Gateway Circuit Breaker + Rate Limiter — M5 (90–120 min)

- `gateway/circuitbreaker/circuitbreaker.go`
- `gateway/circuitbreaker/circuitbreaker_test.go`
- `gateway/ratelimit/ratelimit.go`
- `gateway/ratelimit/ratelimit_test.go`

22. Gateway Proxy + Router + Main — M5 (90–120 min)

- `gateway/proxy/proxy.go`
- `gateway/proxy/proxy_test.go`
- `gateway/router/router.go`
- `gateway/router/router_test.go`
- Update `gateway/main.go` with full M5 wiring

23. End-to-End Tests — M5 (60–90 min)

- `test/e2e_m5_test.go`
- Run: `go test -tags integration -v -timeout 120s ./test/...` **Checkpoint:** All 11 containers healthy. Full E2E flow: register → login → shorten → redirect → stats → notification. Rate limit 429 on 11th shorten. Circuit breaker 503 after 5 url-service failures. Correlation ID in every log line.

File Count Summary

Category	Count
`shared/` Go source files	6
`url-service/` Go source files	16
`analytics-service/` Go source files	16
`user-service/` Go source files	12
`notification-service/` Go source files	10
`gateway/` Go source files	15
SQL migration files	9
Dockerfiles	5
`go.mod` files	6
Shell/test scripts	2
Config/docs	3
Total tracked files	**~100**
Directories	~45
Estimated lines of code	~8,000–10,000

> **Verification:** Every file mentioned in M1–M5 TDD `File Structure` sections is represented. Creation order respects dependency direction: `shared/` before services, M1 infra before M2 auth before M3 URL before M4 analytics before M5 gateway+notifications. A learner can execute `mkdir -p` and `touch` from this tree without ambiguity.