URL Shortener — Engineering Design Document

System Purpose

Five independently deployable Go services (four domain services plus a gateway) implement a production-grade URL shortener: accepting long URLs, generating collision-resistant 7-character base62 short codes, serving sub-millisecond redirects via a Redis read-through cache, and delivering asynchronous side-effects (click analytics, milestone notifications) through a RabbitMQ topic exchange. The system uses URL shortening as a deliberately narrow domain so that every architectural boundary—service ownership, event flow, cache semantics, auth propagation—is unambiguous and verifiable in isolation.

Service Map

Service	Port	Responsibility	DB	Owns Events
api-gateway	8080	JWT validation, per-IP token-bucket rate limiting, circuit breaker, correlation ID injection, reverse proxy routing	Redis (rate-limit counters)	None
url-service	8081	Shorten, redirect (read-through cache), list/delete URLs, outbox worker pool	url_db (PostgreSQL)	URLCreatedEvent, URLClickedEvent, URLDeletedEvent
user-service	8082	Registration, bcrypt password hashing, HS256 JWT issuance; exports shared/auth package consumed by all services	user_db (PostgreSQL)	None
analytics-service	8083	Sequential URLClickedEvent consumer with event-ID deduplication, click aggregation, milestone detection, public stats API	analytics_db (PostgreSQL)	MilestoneReachedEvent
notification-service	8084	Consumes url.created / url.deleted / milestone.reached events, persists notification rows, logs mock email delivery	notification_db (PostgreSQL)	None

Tech Stack

- **Language:** Go 1.23 — stdlib `net/http`, `log/slog`, `crypto/rand`, `math/big`; no HTTP framework
- **Message broker:** RabbitMQ 3.13 — single topic exchange `url-shortener`; durable queues `analytics.clicks` and `notifications.events`
- **Databases:** PostgreSQL 16 — one container per service (host ports 5432–5435); `pgx/v5` with `pgxpool`
- **Cache / rate-limit store:** Redis 7 — ephemeral, no persistence; `go-redis/v9`
- **Auth:** HS256 JWT via `golang-jwt/jwt/v5`; verified locally by each service from a shared `JWT_SECRET` — no inter-service auth calls
- **Containerization:** Docker Compose with per-service `depends_on: condition: service_healthy` gates; Go workspace (`go.work`) monorepo

Key Design Decisions

- **Outbox pattern over direct AMQP publish:** URL mutations (create, click, delete) write the domain event into the `outbox` table within the same PostgreSQL transaction as the URL row change. A background coordinator (1 goroutine) polls every 2 seconds and fans out to 3 worker goroutines for publish. This decouples redirect latency from broker availability and prevents silent event loss on AMQP channel failure; `FOR UPDATE SKIP LOCKED` makes the poller safe against accidental dual-coordinator scenarios.
- **Redis as non-authoritative accelerator:** The redirect hot path attempts a Redis `GET` with a 50 ms timeout; any error or miss falls through to PostgreSQL. Redis holds no data that is not already durable in Postgres. A 410-triggering deactivation immediately invalidates the Redis key after committing the SQL update, keeping the cache consistent without distributed coordination.
- **Database isolation enforced at the container level:** Each service is given its own PostgreSQL container. Cross-service joins are physically impossible, not merely policy-prohibited. Service boundaries are therefore enforceable in tests without process-level mocking.
- **Gateway as stateless infrastructure:** The gateway imports no domain packages (`shared/events` is explicitly excluded). It performs JWT HMAC verification locally, runs a `sync.Mutex`-guarded three-state circuit breaker (CLOSED → OPEN → HALF_OPEN) exclusively on the url-

service proxy path, and applies Redis INCR+EXPIRE token-bucket rate limiting per client IP. Domain ownership rules and business validation live solely in downstream services.

- **Event payloads carry all consumer-needed context:** `URLCreatedEvent`, `URLDeletedEvent`, and `MilestoneReachedEvent` include `user_email` directly. The notification service never calls user-service to resolve an email address, eliminating a synchronous dependency and a failure mode.

Anti-Patterns Explicitly Avoided

Anti-Pattern	Mitigation
Shared database	Four separate PostgreSQL containers; no cross-service schema access at any layer
Synchronous call chain for analytics	Analytics consumes `URLClickedEvent` from RabbitMQ; it never calls url-service to validate a short_code
God service	Auth, URL lifecycle, analytics, and notifications are separate binaries with separate bounded contexts and separate databases
Chatty notification service	All consumer-needed fields (user email, short code) are denormalized into event payloads at publish time
Redis as source of truth	Every write goes to PostgreSQL first; Redis holds only TTL-bounded copies with a hard 1-hour cap
Algorithm confusion in JWT	Token verifier checks `token.Method.(*jwt.SigningMethodHMAC)` before accepting the key, rejecting RS256 and `none`

TDD

Five independently deployable Go binaries (4 services + 1 gateway) communicate via HTTP/JSON for synchronous reads and RabbitMQ topic exchange for asynchronous side effects. Each service owns a dedicated PostgreSQL instance. Redis serves dual roles: read-through URL cache in url-service and shared rate-limit counter store in the gateway. The outbox pattern guarantees at-least-once event delivery without distributed transactions. A circuit breaker state machine protects the redirect hot path. Correlation IDs propagate through both sync HTTP headers and async event payloads, enabling full cross-service trace reconstruction from structured JSON logs.

Foundation: Repo Layout, Shared Contracts, Local Dev Stack

Module ID: `url-shortener-m1`

1. Module Charter

This module establishes the entire structural foundation that every subsequent milestone builds upon. It creates the monorepo directory layout, writes one `go.mod` per service, authors the `docker-compose.yml` that defines the full container mesh (4 PostgreSQL instances, 1 RabbitMQ, 1 Redis, 4 app services, 1 gateway stub), defines the canonical domain event Go structs in `shared/events`, provides a structured-JSON logger stub in `shared/logger`, and wires each service to its own database via `pgxpool` with correct pool sizing. It does not implement any business logic, URL shortening, authentication, message publishing, or consumer loops. The only RabbitMQ work performed here is topology declaration (exchange + queue + binding) on startup — no messages are produced or consumed. The only HTTP surface exposed is `/health` on each service. Every invariant established here — one database per service, Redis as optional cache (never

authoritative), RabbitMQ connection with retry-backoff, fatal exit on missing required config — must never be violated by later milestones. This module is complete when `docker compose up` brings all containers to a healthy state within 60 seconds and every `/health` endpoint returns 200

2. File Structure

Create files in the numbered order below. This ordering ensures dependencies (shared packages) exist before services import them.

```
url-shortener/                          ← monorepo root
│
├── 1.  go.work                         ← Go workspace file tying all modules
│
├── shared/
│   ├── events/
│   │   ├── 2.  events.go               ← all domain event structs
│   │   └── 3.  go.mod                  ← module: github.com/yourhandle/url-shortener/shared/events
│   └── logger/
│       ├── 4.  logger.go               ← structured JSON logger
│       └── 5.  go.mod                  ← module: github.com/yourhandle/url-shortener/shared/logger
│
├── services/
│   ├── url-service/
│   │   ├── 6.  go.mod                  ← module: github.com/yourhandle/url-shortener/url-service
│   │   ├── 7.  main.go
│   │   ├── 8.  config.go
│   │   ├── 9.  db.go
│   │   ├── 10. redis.go
│   │   ├── 11. rabbitmq.go
│   │   ├── 12. health.go
│   │   └── 13. Dockerfile
│   │
│   ├── analytics-service/
│   │   ├── 14. go.mod
│   │   ├── 15. main.go
│   │   ├── 16. config.go
│   │   ├── 17. db.go
│   │   ├── 18. rabbitmq.go
│   │   ├── 19. health.go
│   │   └── 20. Dockerfile
│   │
│   ├── user-service/
│   │   ├── 21. go.mod
│   │   ├── 22. main.go
│   │   ├── 23. config.go
│   │   ├── 24. db.go
│   │   ├── 25. health.go
│   │   └── 26. Dockerfile
│   │
│   └── notification-service/
│       ├── 27. go.mod
│       ├── 28. main.go
│       ├── 29. config.go
│       ├── 30. db.go
│       ├── 31. rabbitmq.go
│       ├── 32. health.go
│       └── 33. Dockerfile
│
├── gateway/
│   ├── 34. go.mod                      ← module: github.com/yourhandle/url-shortener/gateway
│   ├── 35. main.go                     ← stub: /health only
│   ├── 36. config.go
│   ├── 37. health.go
│   └── 38. Dockerfile
│
├── 39. docker-compose.yml
└── 40. README.md
```

3. Complete Data Model

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

All events share a common header. Each event is serialized to JSON when placed in RabbitMQ message bodies. The `EventType` string is the RabbitMQ routing key.

```go
package events

import (
    "time"
)

// EventType constants — these are also the RabbitMQ routing keys.
const (
    EventTypeURLCreated      = "url.created"
    EventTypeURLClicked      = "url.clicked"
    EventTypeURLDeleted      = "url.deleted"
    EventTypeMilestoneReached = "milestone.reached"
)

// BaseEvent carries fields present on every domain event.
// Embedding this in concrete events guarantees envelope consistency.
type BaseEvent struct {
    EventType     string    `json:"event_type"`      // routing key literal
    OccurredAt    time.Time `json:"occurred_at"`     // UTC wall clock at event creation
    CorrelationID string    `json:"correlation_id"`  // propagated from HTTP X-Correlation-ID header
    EventID       string    `json:"event_id"`        // UUID v4, used for idempotency dedup
}

// URLCreatedEvent is published by url-service after a new short URL is persisted.
// Includes user email so notification-service never needs to call user-service.
type URLCreatedEvent struct {
    BaseEvent
    ShortCode   string    `json:"short_code"`
    OriginalURL string    `json:"original_url"`
    UserID      string    `json:"user_id"`   // UUID string
    UserEmail   string    `json:"user_email"`
    ExpiresAt   *time.Time `json:"expires_at,omitempty"`
}

// URLClickedEvent is published by url-service on every successful redirect.
// Analytics-service is the primary consumer; milestone checks happen there.
type URLClickedEvent struct {
    BaseEvent
    ShortCode   string `json:"short_code"`
    IPHash      string `json:"ip_hash"`    // SHA-256(IP+salt), never raw IP
    UserAgent   string `json:"user_agent"`
    Referer     string `json:"referer,omitempty"`
```

```
    ClickedAt   time.Time `json:"clicked_at"` // duplicates OccurredAt for query clarity
}

// URLDeletedEvent is published by url-service when is_active set to false.
type URLDeletedEvent struct {
    BaseEvent
    ShortCode string `json:"short_code"`
    UserID    string `json:"user_id"`
    UserEmail string `json:"user_email"` // for notification-service, avoids callback
}

// MilestoneReachedEvent is published by analytics-service when a click
// threshold (10, 100, 1000) is crossed for a short code.
type MilestoneReachedEvent struct {
    BaseEvent
    ShortCode   string `json:"short_code"`
    UserID      string `json:"user_id"`
    UserEmail   string `json:"user_email"` // denormalized; analytics stores from URLClickedEvent context
    MilestoneN  int    `json:"milestone"`  // 10 | 100 | 1000
    TotalClicks int64  `json:"total_clicks"`
}
```

Why each field exists:

- `EventID` (UUID): deduplication key for idempotent consumers; without it, at-least-once delivery causes double-counting.
- `CorrelationID`: cross-service trace continuity through async boundary; must survive the message broker hop.
- `UserEmail` on mutation events: prevents notification-service from calling user-service (avoids chatty service anti-pattern and sync coupling).
- `ClickedAt` on `URLClickedEvent`: mirrors `OccurredAt` with explicit semantic name; PostgreSQL `date_trunc` queries use this column directly.

3.2 Per-Service Config Structs

Each service has its own `config.go`. All fields are populated once at startup from environment variables. `os.Getenv` is **never** called in handler or hot-path code.

```go
// services/url-service/config.go

package urlservice

type Config struct {

    DatabaseURL string // required; fatal if empty

    RedisURL    string // required; non-fatal if unreachable

    RabbitMQURL string // required; retry with backoff

    Port        string // default "8080"

    ServiceName string // constant "url-service"

}

func loadConfig() (*Config, error) {

    cfg := &Config{

        DatabaseURL: os.Getenv("DATABASE_URL"),

        RedisURL:    os.Getenv("REDIS_URL"),

        RabbitMQURL: os.Getenv("RABBITMQ_URL"),

        Port:        envOrDefault("PORT", "8080"),

        ServiceName: "url-service",

    }

    if cfg.DatabaseURL == "" {

        return nil, fmt.Errorf("DATABASE_URL is required")

    }

    if cfg.RabbitMQURL == "" {

        return nil, fmt.Errorf("RABBITMQ_URL is required")

    }

    if cfg.RedisURL == "" {

        return nil, fmt.Errorf("REDIS_URL is required") // required env var; non-fatal only if *connection* fails

    }

    return cfg, nil

}

func envOrDefault(key, def string) string {

    if v := os.Getenv(key); v != "" {

        return v

    }

    return def

}
```

```go
// services/analytics-service/config.go

package analytics

type Config struct {

    DatabaseURL string

    RabbitMQURL string

    Port        string

    ServiceName string

}

// services/user-service/config.go

package userservice

type Config struct {

    DatabaseURL string

    Port        string

    ServiceName string

}

// services/notification-service/config.go

package notification

type Config struct {

    DatabaseURL string

    RabbitMQURL string

    Port        string

    ServiceName string

}

// gateway/config.go

package gateway

type Config struct {

    URLServiceURL          string // e.g. http://url-service:8080

    AnalyticsServiceURL    string

    UserServiceURL         string

    NotificationServiceURL string

    RedisURL               string

    Port                   string

    ServiceName            string

}
```

For services without Redis or RabbitMQ, those fields simply do not exist in the config struct.

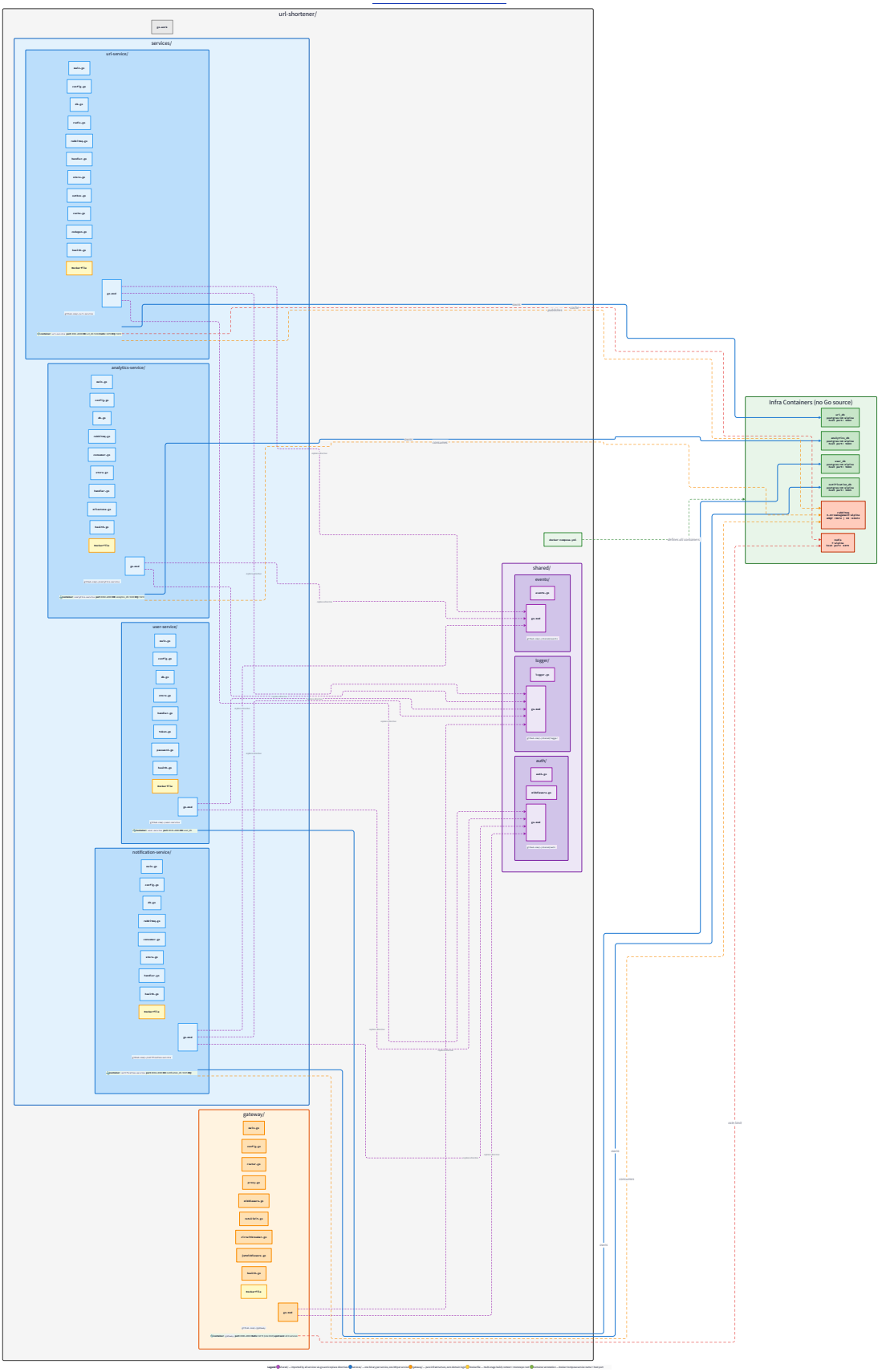
3.3 Health Response Schema

```go
type HealthResponse struct {

    Status  string `json:"status"`  // always "ok"

    Service string `json:"service"` // service name constant

}
```

3.4 RabbitMQ Topology

```
Exchange: "url-shortener"   type: topic   durable: true   autoDelete: false
Queue: "analytics.clicks"
  binding: exchange="url-shortener"  routing_key="url.clicked"  durable: true
Queue: "notifications.events"
  binding: exchange="url-shortener"  routing_key="url.created"  durable: true
  binding: exchange="url-shortener"  routing_key="url.deleted"  durable: true
  binding: exchange="url-shortener"  routing_key="milestone.reached"  durable: true
```

URL Shortener — Monorepo Directory Tree



```
┌──────────────────────────────────────────────────────────┐
│                   RabbitMQ Broker                        │
│                                                          │
│   Exchange: "url-shortener" (topic, durable)             │
│       │                                                  │
│       ├── routing_key: "url.clicked" ──────► Queue:      │
│       │                                      "analytics.clicks"│
│       │                                                  │
│       ├── routing_key: "url.created" ──────► Queue:      │
│       │                                      "notifications.   │
│       ├── routing_key: "url.deleted" ──────►  events"    │
│       │                                                  │
│       └── routing_key: "milestone.reached" ─►  (same queue)   │
└──────────────────────────────────────────────────────────┘
Declaration responsibility:
  url-service:            declares exchange "url-shortener"
  analytics-service:      declares queue "analytics.clicks" + binding
  notification-service:   declares queue "notifications.events" + all 3 bindings
```

---

4. Interface Contracts

4.1 `shared/logger` Package

```go
package logger

import (

    "log/slog"

    "os"

)

// New returns a *slog.Logger writing JSON to stdout.

// serviceName is embedded in every log record as "service" field.

// This is the ONLY logger constructor that should be used across all services.

func New(serviceName string) *slog.Logger

// Usage pattern (all services):

//   log := logger.New("url-service")

//   log.Info("connected to DB", "host", "url_db:5432")

//   log.Error("startup failure", "error", err)
```

Implementation detail: Use `slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{Level: slog.LevelInfo})`. Wrap with `slog.New`. Add `serviceName` as a default attribute via `logger.With("service", serviceName)`. The returned logger is safe for concurrent use (slog guarantees this). **Fields emitted in every log line:**

Field	Type	Source
`time`	RFC3339Nano	slog default
`level`	string	slog default
`service`	string	constructor arg
`msg`	string	log call
any extras	varies	per log call

4.2 DB Connection Factory

```go
// db.go (per service — same pattern, different pool owner)

package urlservice

import (

    "context"

    "fmt"

    "github.com/jackc/pgx/v5/pgxpool"

)

// NewDBPool creates a pgxpool with project-standard settings.

// Returns fatal error if the pool cannot ping the DB within 10s.

//

// Parameters:

//   ctx        - a context with a 10-second timeout applied internally

//   databaseURL - postgres:// DSN from config

//   log        - structured logger for startup messages

//

// Returns:

//   *pgxpool.Pool - ready pool, caller must call .Close() on shutdown

//   error         - non-nil if DB unreachable; caller must os.Exit(1)

func NewDBPool(ctx context.Context, databaseURL string, log *slog.Logger) (*pgxpool.Pool, error) {

    cfg, err := pgxpool.ParseConfig(databaseURL)

    if err != nil {

        return nil, fmt.Errorf("parse db url: %w", err)

    }

    cfg.MaxConns = 10

    cfg.MinConns = 2

    pool, err := pgxpool.NewWithConfig(ctx, cfg)

    if err != nil {

        return nil, fmt.Errorf("create pool: %w", err)

    }

    pingCtx, cancel := context.WithTimeout(ctx, 10*time.Second)

    defer cancel()

    if err := pool.Ping(pingCtx); err != nil {

        pool.Close()

        return nil, fmt.Errorf("ping db: %w", err)

    }
```

```
    log.Info("connected to DB", "max_conns", cfg.MaxConns, "min_conns", cfg.MinConns)

    return pool, nil

}
```

Error contract:

- If `pgxpool.ParseConfig` fails → return error immediately, do not attempt connection.
- If `pool.Ping` times out or is refused → close pool, return wrapped error.
- Caller in `main.go` must call `log.Error(...); os.Exit(1)` on non-nil return.

4.3 Redis Connection Factory (url-service only)

```go
// redis.go — url-service only

package urlservice

import (

    "context"

    "fmt"

    "time"

    "github.com/redis/go-redis/v9"

)

// NewRedisClient creates a Redis client and pings the server.

// UNLIKE the DB pool, a failed ping here is NON-FATAL.

// The function always returns a *redis.Client (usable even if server is down).

// The bool return indicates whether the initial ping succeeded.

//

// Parameters:

//   ctx      - used for initial ping only

//   redisURL - redis://host:port/0 DSN

//   log      - structured logger

//

// Returns:

//   *redis.Client - always non-nil; configured client

//   bool          - true if initial ping succeeded, false otherwise

func NewRedisClient(ctx context.Context, redisURL string, log *slog.Logger) (*redis.Client, bool) {

    opts, err := redis.ParseURL(redisURL)

    if err != nil {

        log.Warn("redis URL parse failed, cache disabled", "error", err)

        return redis.NewClient(&redis.Options{Addr: "localhost:6379"}), false

    }

    client := redis.NewClient(opts)

    pingCtx, cancel := context.WithTimeout(ctx, 3*time.Second)

    defer cancel()

    if err := client.Ping(pingCtx).Err(); err != nil {

        log.Warn("redis unreachable on startup, cache will be disabled until available", "error", err)

        return client, false

    }

    log.Info("connected to Redis cache", "addr", opts.Addr)
```

```
    return client, true

}
```

Critical: The Redis client is returned regardless of ping success. Later cache operations must each independently handle errors (wrap in a helper that logs and continues).

4.4 RabbitMQ Connection Factory

```go
// rabbitmq.go (url-service, analytics-service, notification-service)
package urlservice
import (
    "context"
    "fmt"
    "time"
    amqp "github.com/rabbitmq/amqp091-go"
)
const (
    exchangeName = "url-shortener"
    exchangeType = "topic"
)
// RabbitMQConn wraps an AMQP connection and channel.
type RabbitMQConn struct {
    Conn    *amqp.Connection
    Channel *amqp.Channel
}
// NewRabbitMQConn connects to RabbitMQ with exponential backoff.
// Retries up to maxAttempts times before returning error.
// Each attempt is logged. On success, declares the exchange.
//
// Parameters:
//   ctx         - for cancellation during retry loop
//   amqpURL     - amqp://user:pass@host:5672/
//   log         - structured logger
//   maxAttempts - typically 10 for startup; backoff doubles each retry (1s→2s→4s...max 30s)
//
// Returns:
//   *RabbitMQConn - connected and channel-open; caller owns Close()
//   error         - non-nil after maxAttempts exhausted
func NewRabbitMQConn(ctx context.Context, amqpURL string, log *slog.Logger, maxAttempts int) (*RabbitMQConn, error) {
    var conn *amqp.Connection
    var err error
    backoff := time.Second
    for attempt := 1; attempt <= maxAttempts; attempt++ {
```

```
        conn, err = amqp.Dial(amqpURL)
        if err == nil {
            break
        }
        log.Warn("rabbitmq connection attempt failed",
            "attempt", attempt,
            "max_attempts", maxAttempts,
            "backoff_seconds", backoff.Seconds(),
            "error", err,
        )
        select {
        case <-ctx.Done():
            return nil, fmt.Errorf("context cancelled during rabbitmq connect: %w", ctx.Err())
        case <-time.After(backoff):
        }
        backoff = min(backoff*2, 30*time.Second)
    }
    if err != nil {
        return nil, fmt.Errorf("rabbitmq connect after %d attempts: %w", maxAttempts, err)
    }
    ch, err := conn.Channel()
    if err != nil {
        conn.Close()
        return nil, fmt.Errorf("open rabbitmq channel: %w", err)
    }
    if err := declareExchange(ch); err != nil {
        ch.Close()
        conn.Close()
        return nil, fmt.Errorf("declare exchange: %w", err)
    }
    log.Info("connected to RabbitMQ", "exchange", exchangeName)
    return &RabbitMQConn{Conn: conn, Channel: ch}, nil
}
func declareExchange(ch *amqp.Channel) error {
    return ch.ExchangeDeclare(
        exchangeName, // name
```

```
        exchangeType, // kind: "topic"
        true,         // durable
        false,        // autoDelete
        false,        // internal
        false,        // noWait
        nil,          // args
    )
}

// Close shuts down channel then connection in correct order.
func (r *RabbitMQConn) Close() {
    if r.Channel != nil {
        r.Channel.Close()
    }
    if r.Conn != nil {
        r.Conn.Close()
    }
}
```

4.5 Queue/Binding Declaration (Consumer Services)

```go
// analytics-service rabbitmq.go — additional function after NewRabbitMQConn

func DeclareAnalyticsQueue(ch *amqp.Channel) error {

    q, err := ch.QueueDeclare(

        "analytics.clicks", // name

        true,               // durable

        false,              // autoDelete

        false,              // exclusive

        false,              // noWait

        nil,                // args

    )

    if err != nil {

        return fmt.Errorf("declare analytics.clicks queue: %w", err)

    }

    if err := ch.QueueBind(

        q.Name,        // queue name

        "url.clicked", // routing key

        exchangeName,  // exchange

        false,         // noWait

        nil,           // args

    ); err != nil {

        return fmt.Errorf("bind analytics.clicks to url.clicked: %w", err)

    }

    return nil

}

// notification-service rabbitmq.go

func DeclareNotificationQueue(ch *amqp.Channel) error {

    q, err := ch.QueueDeclare(

        "notifications.events",

        true, false, false, false, nil,

    )

    if err != nil {

        return fmt.Errorf("declare notifications.events queue: %w", err)

    }

    for _, routingKey := range []string{"url.created", "url.deleted", "milestone.reached"} {

        if err := ch.QueueBind(q.Name, routingKey, exchangeName, false, nil); err != nil {
```

```
            return fmt.Errorf("bind notifications.events to %s: %w", routingKey, err)
        }
    }
    return nil
}
```

Error contract for consumer services: If queue/binding declaration fails, it is **fatal**. The consumer service cannot function without its queue. Call `log.Error(...); os.Exit(1)`.

4.6 Health Handler

```go
// health.go (same pattern all services; serviceName differs)
package urlservice
import (
    "encoding/json"
    "net/http"
)
type HealthResponse struct {
    Status  string `json:"status"`
    Service string `json:"service"`
}
// NewHealthHandler returns an http.HandlerFunc for GET /health.
// No authentication required. Responds in < 10ms.
// Does NOT check DB connectivity on every request (too expensive).
func NewHealthHandler(serviceName string) http.HandlerFunc {
    resp := HealthResponse{Status: "ok", Service: serviceName}
    body, _ := json.Marshal(resp) // pre-encoded; never changes
    return func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusOK)
        w.Write(body)
    }
}
```

Why pre-encode: The response never changes during the process lifetime. Encoding once at startup avoids per-request allocation. `json.Marshal` on this tiny struct cannot fail (no custom marshalers, no channels/funcs)

---

5. Algorithm Specification

5.1 Startup Sequence — url-service (most complex; others are subsets)

```
1. loadConfig()
   → if any required env var is empty: log.Error + os.Exit(1)
2. log := logger.New(cfg.ServiceName)
   → all subsequent log calls use this logger
3. pool, err := NewDBPool(ctx, cfg.DatabaseURL, log)
   → if err != nil: log.Error("db unreachable", "error", err); os.Exit(1)
   → if ok: log("connected to DB") already emitted inside factory
4. redisClient, _ := NewRedisClient(ctx, cfg.RedisURL, log)
   → warning already logged inside factory; continue regardless
5. mq, err := NewRabbitMQConn(ctx, cfg.RabbitMQURL, log, maxAttempts=10)
   → if err != nil: log.Error + os.Exit(1)
   → exchange declared inside factory
6. Register HTTP routes:
   mux := http.NewServeMux()
   mux.HandleFunc("GET /health", NewHealthHandler(cfg.ServiceName))
7. Start HTTP server on cfg.Port
   → log.Info("server listening", "port", cfg.Port)
8. Block on server.ListenAndServe() or OS signal
   → on SIGTERM/SIGINT: pool.Close(), mq.Close(), redisClient.Close()
```

5.2 Startup Sequence — analytics-service

Steps 1-5 as url-service except:

- No Redis (step 4 omitted)
- After step 5 (RabbitMQ connected), call `DeclareAnalyticsQueue(mq.Channel)`
  - If error: log.Error + os.Exit(1)

5.3 Startup Sequence — notification-service

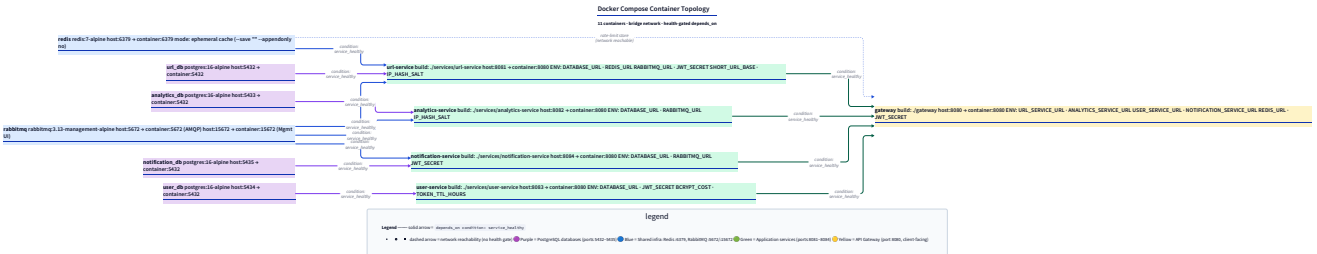
Same as analytics-service but calls `DeclareNotificationQueue(mq.Channel)` instead.

5.4 Startup Sequence — user-service

Steps 1 (DATABASE_URL required, no RABBITMQ_URL), 2, 3, 6, 7, 8 only. No Redis, no RabbitMQ.

5.5 Startup Sequence — gateway (stub)

Steps 1 (requires URLServiceURL, AnalyticsServiceURL, UserServiceURL, NotificationServiceURL), 2, 6, 7, 8. Registers `/health` only at this milestone.



```
url-service startup:
  ┌──────────┐  ┌────────┐  ┌───────┐  ┌─────────┐  ┌────────┐
  |loadConfig|→ |DB ping |→ |Redis  |→ |RabbitMQ |→ |HTTP    |
  |          |  |(fatal) |  |ping   |  |connect  |→ |server  |
  └──────────┘  └────────┘  |(warn) |  |+ exch   |  |listen  |
                            └───────┘  |(fatal)  |  └────────┘
                                       └─────────┘
analytics/notification startup:
  loadConfig → DB ping (fatal) → RabbitMQ connect (fatal) → queue declare (fatal) → HTTP
user-service startup:
  loadConfig → DB ping (fatal) → HTTP
gateway startup:
  loadConfig → HTTP (health only for now)
```

6. Docker Compose Specification

6.1 Complete `docker-compose.yml`

YAML

```yaml
version: "3.9"

networks:
  url-shortener:
    driver: bridge

volumes:
  url_db_data:
  analytics_db_data:
  user_db_data:
  notification_db_data:
  rabbitmq_data:
  redis_data:

services:
  # ── Databases ──────────────────────────────────────────────────────
  url_db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: urldb
      POSTGRES_USER: urluser
      POSTGRES_PASSWORD: urlpass
    ports:
      - "5432:5432"
    volumes:
      - url_db_data:/var/lib/postgresql/data
    networks:
      - url-shortener
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U urluser -d urldb"]
      interval: 5s
      timeout: 5s
      retries: 10
      start_period: 10s
  analytics_db:
    image: postgres:16-alpine
    environment:
```

```
      POSTGRES_DB: analyticsdb
      POSTGRES_USER: analyticsuser
      POSTGRES_PASSWORD: analyticspass
    ports:
      - "5433:5432"
    volumes:
      - analytics_db_data:/var/lib/postgresql/data
    networks:
      - url-shortener
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U analyticsuser -d analyticsdb"]
      interval: 5s
      timeout: 5s
      retries: 10
      start_period: 10s
  user_db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: userdb
      POSTGRES_USER: useruser
      POSTGRES_PASSWORD: userpass
    ports:
      - "5434:5432"
    volumes:
      - user_db_data:/var/lib/postgresql/data
    networks:
      - url-shortener
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U useruser -d userdb"]
      interval: 5s
      timeout: 5s
      retries: 10
      start_period: 10s
  notification_db:
    image: postgres:16-alpine
    environment:
```

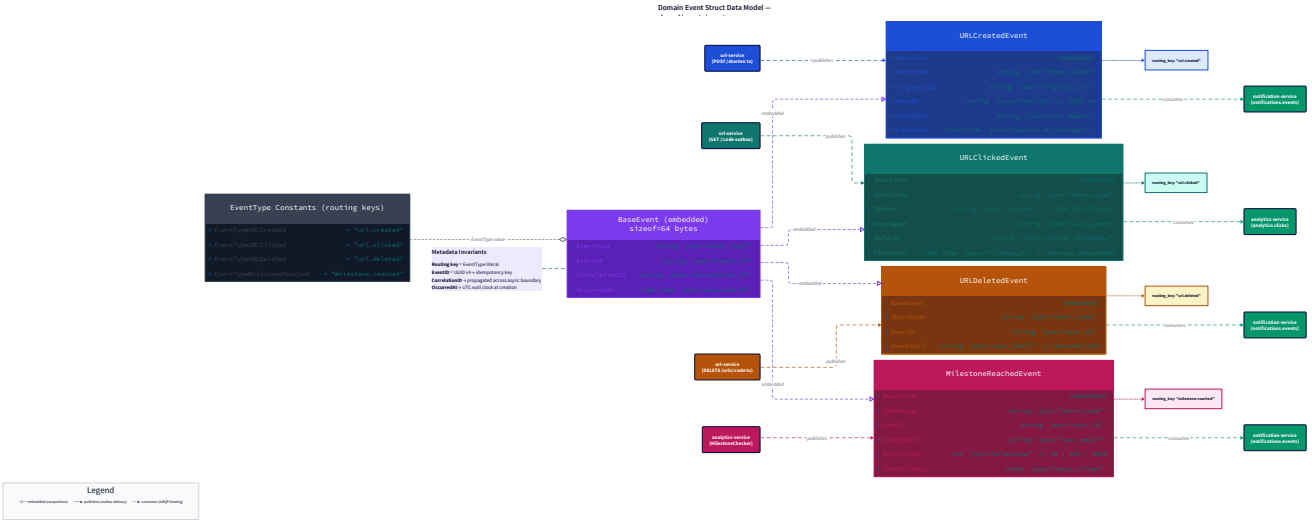
```
    POSTGRES_DB: notificationdb
    POSTGRES_USER: notificationuser
    POSTGRES_PASSWORD: notificationpass
  ports:
    - "5435:5432"
  volumes:
    - notification_db_data:/var/lib/postgresql/data
  networks:
    - url-shortener
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U notificationuser -d notificationdb"]
    interval: 5s
    timeout: 5s
    retries: 10
    start_period: 10s

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

rabbitmq:
  image: rabbitmq:3.13-management-alpine
  environment:
    RABBITMQ_DEFAULT_USER: guest
    RABBITMQ_DEFAULT_PASS: guest
  ports:
    - "5672:5672" # AMQP
    - "15672:15672" # Management UI
  volumes:
    - rabbitmq_data:/var/lib/rabbitmq
  networks:
    - url-shortener
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "ping"]
    interval: 10s
    timeout: 10s
    retries: 10
    start_period: 20s

# ── Cache ──────────────────────────────────────────────

redis:
```

```yaml
  image: redis:7-alpine
  command: redis-server --save "" --appendonly no # ephemeral; cache only
  ports:
    - "6379:6379"
  volumes:
    - redis_data:/data
  networks:
    - url-shortener
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 5s
    timeout: 3s
    retries: 10
# ── Application Services ──────────────────────────────────────
url-service:
  build:
    context: ./services/url-service
    dockerfile: Dockerfile
  environment:
    DATABASE_URL: postgres://urluser:urlpass@url_db:5432/urldb
    REDIS_URL: redis://redis:6379/0
    RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
    PORT: "8080"
  ports:
    - "8081:8080"
  networks:
    - url-shortener
  depends_on:
    url_db:
      condition: service_healthy
    redis:
      condition: service_healthy
    rabbitmq:
      condition: service_healthy
  healthcheck:
    test: ["CMD-SHELL", "wget -qO- http://localhost:8080/health || exit 1"]
```

```
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 15s
analytics-service:
  build:
    context: ./services/analytics-service
    dockerfile: Dockerfile
  environment:
    DATABASE_URL: postgres://analyticsuser:analyticspass@analytics_db:5432/analyticsdb
    RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
    PORT: "8080"
  ports:
    - "8082:8080"
  networks:
    - url-shortener
  depends_on:
    analytics_db:
      condition: service_healthy
    rabbitmq:
      condition: service_healthy
  healthcheck:
    test: ["CMD-SHELL", "wget -qO- http://localhost:8080/health || exit 1"]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 15s
user-service:
  build:
    context: ./services/user-service
    dockerfile: Dockerfile
  environment:
    DATABASE_URL: postgres://useruser:userpass@user_db:5432/userdb
    PORT: "8080"
  ports:
    - "8083:8080"
```

```
    networks:
      - url-shortener
    depends_on:
      user_db:
        condition: service_healthy
    healthcheck:
      test: ["CMD-SHELL", "wget -qO- http://localhost:8080/health || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 15s
  notification-service:
    build:
      context: ./services/notification-service
      dockerfile: Dockerfile
    environment:
      DATABASE_URL: postgres://notificationuser:notificationpass@notification_db:5432/notificationdb
      RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
      PORT: "8080"
    ports:
      - "8084:8080"
    networks:
      - url-shortener
    depends_on:
      notification_db:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
    healthcheck:
      test: ["CMD-SHELL", "wget -qO- http://localhost:8080/health || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 15s
  gateway:
    build:
```

```
    context: ./gateway
    dockerfile: Dockerfile
  environment:
    URL_SERVICE_URL: http://url-service:8080
    ANALYTICS_SERVICE_URL: http://analytics-service:8080
    USER_SERVICE_URL: http://user-service:8080
    NOTIFICATION_SERVICE_URL: http://notification-service:8080
    PORT: "8080"
  ports:
    - "8080:8080"
  networks:
    - url-shortener
  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
```




```
Port mapping (host → container):
  8080 → gateway
  8081 → url-service
  8082 → analytics-service
  8083 → user-service
  8084 → notification-service
  5432 → url_db         (postgres)
  5433 → analytics_db   (postgres)
  5434 → user_db        (postgres)
  5435 → notification_db (postgres)
  5672 → rabbitmq       (AMQP)
  15672→ rabbitmq       (management UI)
  6379 → redis
depends_on chain (healthy gate):
  gateway ← {url-service, analytics-service, user-service, notification-service}
  url-service ← {url_db, redis, rabbitmq}
  analytics-service ← {analytics_db, rabbitmq}
  notification-service ← {notification_db, rabbitmq}
  user-service ← {user_db}
```

6.2 Dockerfile (uniform across all services)

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

FROM golang:1.23-alpine AS builder

WORKDIR /app

# Copy go.work and all module go.mod/go.sum files first (cache layer)

COPY go.mod go.sum ./

# Download dependencies

RUN go mod download

# Copy source

COPY . .

# Build static binary

RUN CGO_ENABLED=0 GOOS=linux go build -ldflags="-s -w" -o /service ./...

FROM scratch

COPY --from=builder /service /service

EXPOSE 8080

ENTRYPOINT ["/service"]
```

Note on go.work: Because services share `shared/events` and `shared/logger`, the `go.work` file at root must list all modules so Docker build context resolves replace directives correctly. Copy the entire monorepo into each service's build context, or use a root-level multi-stage build. The simpler approach for this milestone: set Docker build context to monorepo root and specify Dockerfile path per service. Revised `docker-compose.yml` build section for monorepo:

```yaml
url-service:

  build:

    context: . # monorepo root

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

And corresponding Dockerfile modification to `WORKDIR /app` then `COPY . .` from root.

6.3 Go Workspace (`go.work`)

```
go 1.23
use (
    ./shared/events
    ./shared/logger
    ./services/url-service
    ./services/analytics-service
    ./services/user-service
    ./services/notification-service
    ./gateway
)
```

Each service `go.mod` references shared packages:

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

7. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?	Exit?
Missing required env var (DATABASE_URL, RABBITMQ_URL)	`loadConfig()` at startup	Log descriptive message naming the variable	No (pre-HTTP)	Yes (`os.Exit(1)`)
DB unreachable at startup (`pool.Ping` timeout)	`NewDBPool()`	Log error with DSN host (mask password)	No	Yes (`os.Exit(1)`)
DB DSN malformed	`pgxpool.ParseConfig`	Log with raw error	No	Yes (`os.Exit(1)`)
Redis URL malformed	`redis.ParseURL`	Log warning, use default client, continue	No	No
Redis unreachable at startup (`Ping` fail)	`NewRedisClient()`	Log warning, return client anyway	No	No
RabbitMQ unreachable, within maxAttempts	`amqp.Dial` in retry loop	Log each attempt with backoff duration, retry	No	No (retrying)
RabbitMQ unreachable, maxAttempts exhausted	`NewRabbitMQConn()`	Log fatal	No	Yes (`os.Exit(1)`)
Exchange declare failure	`declareExchange()`	Log error	No	Yes (`os.Exit(1)`)
Queue declare failure (consumer services)	`DeclareAnalyticsQueue` / `DeclareNotificationQueue`	Log error	No	Yes (`os.Exit(1)`)
Queue binding failure	same functions	Log error	No	Yes (`os.Exit(1)`)
Context cancelled during RabbitMQ retry	select in retry loop	Return wrapped error	No	Yes (caller exits)
`/health` handler write error	`w.Write` return	Ignored (client disconnected)	No	No

Password masking for DB DSN in logs:

```go
// When logging the DSN in an error, strip credentials:
func maskDSN(dsn string) string {
    u, err := url.Parse(dsn)
    if err != nil {
        return "[unparseable DSN]"
    }
    u.User = url.User("***")
    return u.String()
}
```

8. Implementation Sequence with Checkpoints

Phase 1: Monorepo Scaffold and go.mod per Service (0.5–1h)

1. Create directory tree exactly as shown in §2.
2. Write `go.work` at root.
3. Write `go.mod` for `shared/events` and `shared/logger` (no dependencies beyond stdlib for logger).
4. Write `go.mod` for each service with the `require` + `replace` blocks.
5. Run `go work sync` at root. **Checkpoint:** `go build ./...` from monorepo root exits 0 (even with empty `main.go` stubs that just `package main; func main() {}`).

Phase 2: docker-compose.yml with All Containers (1–1.5h)

1. Write `docker-compose.yml` as specified in §6.1.
2. Verify `docker compose config` exits 0 (validates YAML syntax).
3. Run `docker compose up rabbitmq redis url_db analytics_db user_db notification_db -d`.
4. Check `docker compose ps` — all six infra containers show `healthy`. **Checkpoint:** `docker compose ps` shows all infra containers in `healthy` state. `psql -h localhost -p 5432 -U urluser -d urldb -c '\l'` connects. `redis-cli -p 6379 ping` returns `PONG`. RabbitMQ management UI reachable at `http://localhost:15672` (guest/guest).

Phase 3: shared/events and shared/logger (0.5h)

1. Write `shared/events/events.go` exactly as §3.1.
2. Write `shared/logger/logger.go`:

```go
package logger

import (

    "log/slog"

    "os"

)

func New(serviceName string) *slog.Logger {

    h := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{

        Level: slog.LevelInfo,

    })

    return slog.New(h).With("service", serviceName)

}
```

1. Run `go build github.com/yourhandle/url-shortener/shared/...` → exits 0. **Checkpoint:** A throwaway `main.go` that imports `shared/logger`, calls `logger.New("test").Info("hello")`, and runs prints valid JSON to stdout: `{"time":"...","level":"INFO","service":"test","msg":"hello"}`.

Phase 4: Per-Service Config, DB Pool, Redis, Health Handler (1–1.5h)

For each service in order (url-service → analytics-service → user-service → notification-service → gateway):

1. Write `config.go` (Config struct + `loadConfig()`).
2. Write `db.go` (`NewDBPool`).
3. (url-service only) Write `redis.go` (`NewRedisClient`).
4. Write `health.go` (`NewHealthHandler`).
5. Wire in `main.go`:

```go
// services/url-service/main.go

package main

import (

    "context"

    "fmt"

    "net/http"

    "os"

    "os/signal"

    "syscall"

    "github.com/yourhandle/url-shortener/shared/logger"

)

func main() {

    cfg, err := loadConfig()

    if err != nil {

        // logger not yet initialized; use fmt

        fmt.Fprintf(os.Stderr, "config error: %v\n", err)

        os.Exit(1)

    }

    log := logger.New(cfg.ServiceName)

    ctx := context.Background()

    pool, err := NewDBPool(ctx, cfg.DatabaseURL, log)

    if err != nil {

        log.Error("db unreachable", "error", err)

        os.Exit(1)

    }

    defer pool.Close()

    redisClient, _ := NewRedisClient(ctx, cfg.RedisURL, log)

    defer redisClient.Close()

    mq, err := NewRabbitMQConn(ctx, cfg.RabbitMQURL, log, 10)

    if err != nil {

        log.Error("rabbitmq unreachable", "error", err)

        os.Exit(1)

    }

    defer mq.Close()

    mux := http.NewServeMux()

    mux.HandleFunc("GET /health", NewHealthHandler(cfg.ServiceName))

    srv := &http.Server{Addr: ":" + cfg.Port, Handler: mux}
```

```
    go func() {
        log.Info("server listening", "port", cfg.Port)
        if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Error("server error", "error", err)
            os.Exit(1)
        }
    }()
    quit := make(chan os.Signal, 1)
    signal.Notify(quit, syscall.SIGTERM, syscall.SIGINT)
    <-quit
    log.Info("shutting down")
    srv.Shutdown(ctx)
}
```

1. Build each service: `go build ./services/url-service/...` → exits 0. **Checkpoint:** `go build ./...` from root exits 0 for all services. Each binary starts without panicking when environment variables are set.

Phase 5: RabbitMQ Exchange and Queue/Binding Declaration (0.5–1h)

1. Complete `rabbitmq.go` in url-service (exchange declare already in `NewRabbitMQConn`).
2. Write `DeclareAnalyticsQueue` in analytics-service `rabbitmq.go`.
3. Write `DeclareNotificationQueue` in notification-service `rabbitmq.go`.
4. Call declare functions in `main.go` of analytics-service and notification-service after `NewRabbitMQConn` returns. **Checkpoint:** Run all four app services locally (with env vars pointing to Docker infra containers):

```bash
DATABASE_URL=... REDIS_URL=... RABBITMQ_URL=... go run ./services/url-service/
```

RabbitMQ management UI at `http://localhost:15672/#/exchanges` shows `url-shortener` exchange (type: topic, durable). `http://localhost:15672/#/queues` shows `analytics.clicks` and `notifications.events` queues with correct bindings.

Phase 6: Full docker compose up + README (0.5h)

1. Write Dockerfiles as §6.2, ensuring build context is monorepo root.
2. Update `docker-compose.yml` build contexts to `.` with explicit Dockerfile paths.
3. Run `docker compose up --build -d`.
4. Wait up to 60s then check `docker compose ps` — all containers `healthy`.
5. Smoke test each `/health` endpoint:

```bash
for port in 8080 8081 8082 8083 8084; do
  echo -n "Port $port: "
  curl -s http://localhost:$port/health
  echo
done
```

Expected output for each:

```json
{ "status": "ok", "service": "<service-name>" }
```

1. Write `README.md` (§9). **Checkpoint:** All five `/health` endpoints return 200 with correct `service` field. `docker compose down -v` tears down everything cleanly with exit 0. `docker compose up` again restores all to healthy.

---

9. Test Specification

9.1 `shared/events` Package Tests

```go
// shared/events/events_test.go

package events_test

import (

    "encoding/json"

    "testing"

    "time"

    "github.com/yourhandle/url-shortener/shared/events"

)

func TestURLCreatedEvent_RoundTrip(t *testing.T) {

    evt := events.URLCreatedEvent{

        BaseEvent: events.BaseEvent{

            EventType:     events.EventTypeURLCreated,

            OccurredAt:    time.Now().UTC().Truncate(time.Millisecond),

            CorrelationID: "corr-123",

            EventID:       "evt-456",

        },

        ShortCode:   "abc1234",

        OriginalURL: "https://example.com/long/path",

        UserID:      "user-uuid",

        UserEmail:   "user@example.com",

    }

    b, err := json.Marshal(evt)

    if err != nil {

        t.Fatalf("marshal: %v", err)

    }

    var decoded events.URLCreatedEvent

    if err := json.Unmarshal(b, &decoded); err != nil {

        t.Fatalf("unmarshal: %v", err)

    }

    if decoded.EventType != events.EventTypeURLCreated {

        t.Errorf("event_type: got %q want %q", decoded.EventType, events.EventTypeURLCreated)

    }

    if decoded.ShortCode != evt.ShortCode {

        t.Errorf("short_code mismatch")
```

```
    }
    if decoded.ExpiresAt != nil {
        t.Errorf("nil expires_at should stay nil")
    }
}
func TestURLClickedEvent_OmitEmptyReferer(t *testing.T) {
    evt := events.URLClickedEvent{
        BaseEvent: events.BaseEvent{EventType: events.EventTypeURLClicked},
        // Referer intentionally empty
    }
    b, _ := json.Marshal(evt)
    var m map[string]any
    json.Unmarshal(b, &m)
    if _, ok := m["referer"]; ok {
        t.Error("referer should be omitted when empty")
    }
}
func TestEventTypeConstants(t *testing.T) {
    // routing keys must match RabbitMQ binding keys exactly
    cases := map[string]string{
        events.EventTypeURLCreated:       "url.created",
        events.EventTypeURLClicked:       "url.clicked",
        events.EventTypeURLDeleted:       "url.deleted",
        events.EventTypeMilestoneReached: "milestone.reached",
    }
    for got, want := range cases {
        if got != want {
            t.Errorf("constant mismatch: got %q want %q", got, want)
        }
    }
}
```

9.2 Config Loading Tests

```go
// services/url-service/config_test.go

package urlservice

import (

    "os"

    "testing"

)

func TestLoadConfig_MissingDatabaseURL(t *testing.T) {

    os.Unsetenv("DATABASE_URL")

    os.Setenv("RABBITMQ_URL", "amqp://localhost:5672")

    os.Setenv("REDIS_URL", "redis://localhost:6379")

    _, err := loadConfig()

    if err == nil {

        t.Fatal("expected error for missing DATABASE_URL")

    }

}

func TestLoadConfig_MissingRabbitMQURL(t *testing.T) {

    os.Setenv("DATABASE_URL", "postgres://user:pass@localhost/db")

    os.Unsetenv("RABBITMQ_URL")

    os.Setenv("REDIS_URL", "redis://localhost:6379")

    _, err := loadConfig()

    if err == nil {

        t.Fatal("expected error for missing RABBITMQ_URL")

    }

}

func TestLoadConfig_DefaultPort(t *testing.T) {

    os.Setenv("DATABASE_URL", "postgres://user:pass@localhost/db")

    os.Setenv("RABBITMQ_URL", "amqp://localhost:5672")

    os.Setenv("REDIS_URL", "redis://localhost:6379")

    os.Unsetenv("PORT")

    cfg, err := loadConfig()

    if err != nil {

        t.Fatalf("unexpected error: %v", err)

    }

    if cfg.Port != "8080" {

        t.Errorf("default port: got %q want %q", cfg.Port, "8080")
```

```
    }
}

func TestLoadConfig_ServiceName(t *testing.T) {
    os.Setenv("DATABASE_URL", "postgres://user:pass@localhost/db")
    os.Setenv("RABBITMQ_URL", "amqp://localhost:5672")
    os.Setenv("REDIS_URL", "redis://localhost:6379")
    cfg, _ := loadConfig()
    if cfg.ServiceName != "url-service" {
        t.Errorf("service name: got %q want url-service", cfg.ServiceName)
    }
}
```

9.3 Health Handler Tests

```go
// services/url-service/health_test.go

package urlservice

import (

    "encoding/json"

    "net/http"

    "net/http/httptest"

    "testing"

)

func TestHealthHandler_ResponseShape(t *testing.T) {

    handler := NewHealthHandler("url-service")

    req := httptest.NewRequest("GET", "/health", nil)

    rec := httptest.NewRecorder()

    handler(rec, req)

    if rec.Code != http.StatusOK {

        t.Errorf("status: got %d want 200", rec.Code)

    }

    ct := rec.Header().Get("Content-Type")

    if ct != "application/json" {

        t.Errorf("content-type: got %q want application/json", ct)

    }

    var resp HealthResponse

    if err := json.Unmarshal(rec.Body.Bytes(), &resp); err != nil {

        t.Fatalf("parse body: %v", err)

    }

    if resp.Status != "ok" {

        t.Errorf("status field: got %q want ok", resp.Status)

    }

    if resp.Service != "url-service" {

        t.Errorf("service field: got %q want url-service", resp.Service)

    }

}

func TestHealthHandler_DifferentServiceNames(t *testing.T) {

    for _, name := range []string{"analytics-service", "user-service", "notification-service", "gateway"} {

        t.Run(name, func(t *testing.T) {

            h := NewHealthHandler(name)
```

```
            req := httptest.NewRequest("GET", "/health", nil)
            rec := httptest.NewRecorder()
            h(rec, req)
            var resp HealthResponse
            json.Unmarshal(rec.Body.Bytes(), &resp)
            if resp.Service != name {
                t.Errorf("service: got %q want %q", resp.Service, name)
            }
        })
    }
}
```

9.4 Integration Smoke Test (shell script)

```bash
#!/usr/bin/env bash

# scripts/smoke_test.sh — run after docker compose up

set -e

SERVICES=(

  "http://localhost:8080/health:gateway"

  "http://localhost:8081/health:url-service"

  "http://localhost:8082/health:analytics-service"

  "http://localhost:8083/health:user-service"

  "http://localhost:8084/health:notification-service"

)

for entry in "${SERVICES[@]}"; do

  URL="${entry%%:*}"

  EXPECTED_SERVICE="${entry##*:}"

  RESPONSE=$(curl -sf "$URL")

  STATUS=$(echo "$RESPONSE" | python3 -c "import sys,json; d=json.load(sys.stdin); print(d['status'])")

  SERVICE=$(echo "$RESPONSE" | python3 -c "import sys,json; d=json.load(sys.stdin); print(d['service'])")

  if [ "$STATUS" != "ok" ]; then

    echo "FAIL $URL: status=$STATUS"

    exit 1

  fi

  if [ "$SERVICE" != "$EXPECTED_SERVICE" ]; then

    echo "FAIL $URL: service=$SERVICE (expected $EXPECTED_SERVICE)"

    exit 1

  fi

  echo "PASS $URL → {status:ok, service:$SERVICE}"

done

echo ""

echo "All health checks passed."
```

10. Performance Targets

Operation	Target	How to Measure
`docker compose up` → all containers healthy	≤ 60s	`time docker compose up -d` then poll `docker compose ps` until no container shows `starting`
`/health` response time (any service)	< 10ms p99	`wrk -t1 -c10 -d10s http://localhost:8081/health` — check Latency 99th percentile
`pgxpool.Acquire` on warm pool	< 100ms	Instrument `NewDBPool` with `time.Since` around `pool.Ping`; log duration
`docker compose down -v` teardown	≤ 30s	`time docker compose down -v`
Service startup (process → HTTP ready)	< 5s after infra healthy	Measure from container start to first successful healthcheck

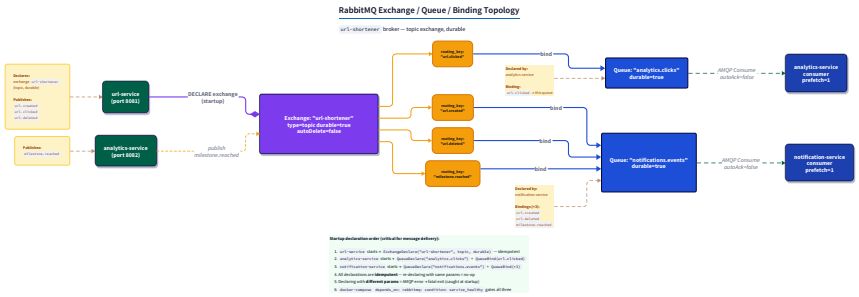
11. README Content

```
# URL Shortener — Microservices

## Prerequisites

- Docker >= 24 with Compose v2
- Go 1.23+ (for local development outside Docker)

## Start the full stack

\```bash
docker compose up --build -d
\```
Wait ~60s for all containers to reach healthy state:
\```bash
docker compose ps
\```

## Verify health

\```bash
bash scripts/smoke_test.sh
\```

## Stop and clean up (including volumes)

\```bash
docker compose down -v
\```

## Service ports (host)

| Service              | Port  |
| -------------------- | ----- |
| Gateway              | 8080  |
| URL Service          | 8081  |
| Analytics Service    | 8082  |
```

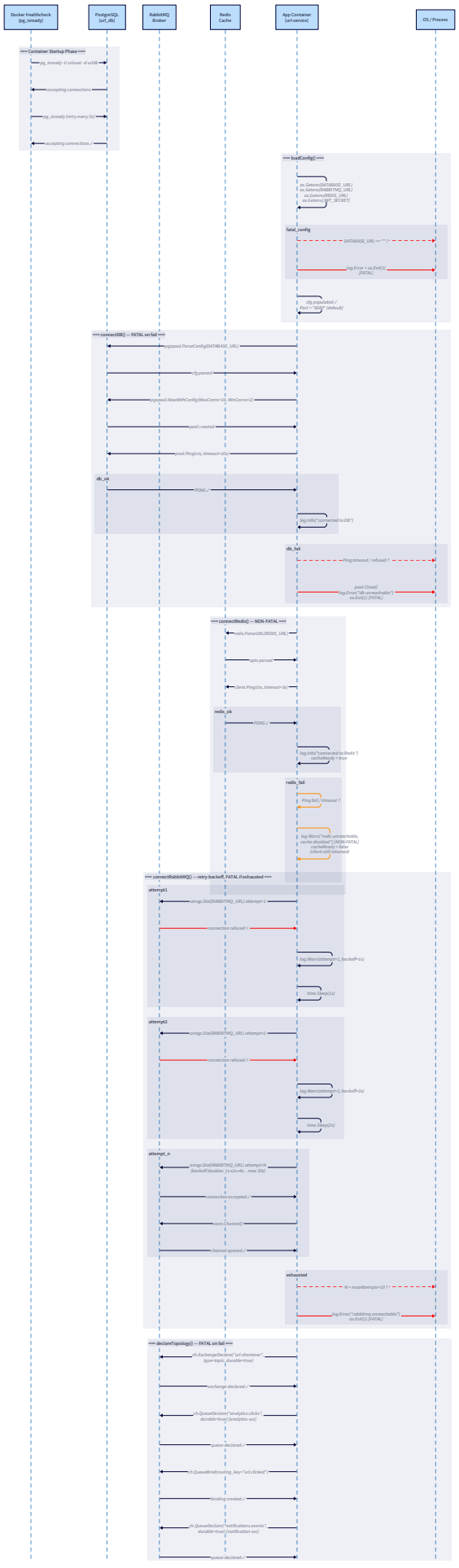
```
| User Service          | 8083  |
| Notification Service | 8084  |
| RabbitMQ AMQP         | 5672  |
| RabbitMQ Management   | 15672 |
| Redis                 | 6379  |
| URL DB (PostgreSQL)   | 5432  |
| Analytics DB          | 5433  |
| User DB               | 5434  |
| Notification DB       | 5435  |

## RabbitMQ credentials

User: guest / Password: guest
Management UI: http://localhost:15672
```

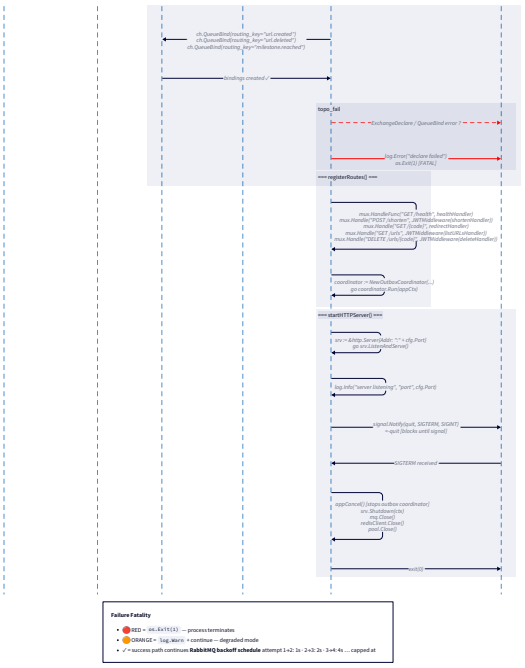
---

12. Pitfall Prevention

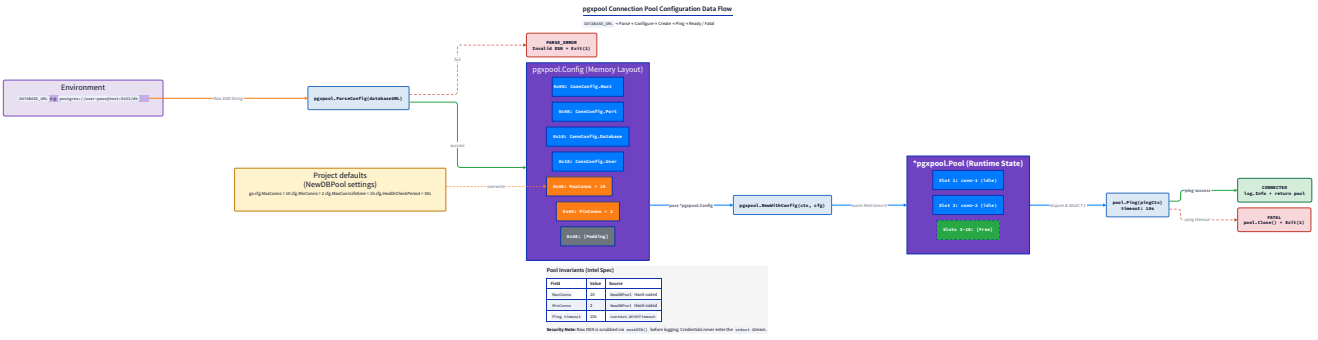



```
PITFALL MAP for M1:
1. Hardcoded "localhost" in DSN
   Wrong:  DATABASE_URL=postgres://user:pass@localhost:5432/urldb
   Right:  DATABASE_URL=postgres://user:pass@url_db:5432/urldb
   Why: Docker bridge network resolves service names, not "localhost"
2. Single PostgreSQL with multiple schemas
   Wrong:  url_db: ... POSTGRES_DB: urls,analytics,users
   Right:  Four separate postgres containers on ports 5432-5435
   Why: database-per-service is the invariant; schemas in one instance
        still couple services via a shared crash domain
3. Producer-only exchange/queue declaration
   Wrong:  url-service declares "analytics.clicks" queue
   Right:  analytics-service declares its OWN queue and binding
   Why:    If analytics-service starts first, messages publish to
           exchange but no queue exists → messages dropped silently
4. No depends_on + healthcheck → crash loop
   Wrong:  app starts, DB isn't ready, pgxpool.Ping fails → os.Exit(1)
           Docker restarts → repeat
   Right:  depends_on with condition: service_healthy gates app start
           until pg_isready returns 0
5. Redis as authoritative store
   Wrong:  Storing the canonical URL record only in Redis
   Right:  All writes go to PostgreSQL first; Redis holds TTL copies
```


Docker Healthcheck (pg_isready)
PostgreSQL (url_db)
RabbitMQ Broker
Redis Cache
App Container (url-service)
OS / Process
=== Container Startup Phase ===
pg_isready -U urluser -d urldb
accepting connections
pg_isready (retry every 3s)
accepting connections ✓
=== loadConfig() ===
os.Getenv(DATABASE_URL)
os.Getenv(RABBITMQ_URL)
os.Getenv(REDIS_URL)
os.Getenv(JWT_SECRET)
fatal_config
DATABASE_URL == "" ?
log.Error + os.Exit(1) [FATAL]
cfg populated ✓
Port = "8080" (default)
=== connectDB() — FATAL on fail ===
pgxpool.ParseConfig(DATABASE_URL)
cfg.parsed
pgxpool.NewWithConfig(MaxConns=10, MinConns=2)
pool created
pool.Ping(ctx, timeout=20s)
db_ok
PONG ✓
log.Info("connected to DB")
db_fail
Ping timeout / refused ?
pool.Close()
log.Error("db unreachable")
os.Exit(1) [FATAL]
=== connectRedis() — NON-FATAL ===
redis.ParseURL(REDIS_URL)
opts.parsed
client.Ping(ctx, timeout=3s)
redis_ok
PONG ✓
log.Info("connected to Redis")
cacheReady = true
redis_fail
Ping fail / timeout ?
log.Warn("redis unreachable, cache disabled") [NON-FATAL]
cacheReady = false
(client still returned)
=== connectRabbitMQ() — retry-backoff, FATAL if exhausted ===
attempt1
amqp.Dial(RABBITMQ_URL) attempt=1
connection refused ✗
log.Warn(attempt=1, backoff=2s)
time.Sleep(2s)
attempt2
amqp.Dial(RABBITMQ_URL) attempt=2
connection refused ✗
log.Warn(attempt=2, backoff=2s)
time.Sleep(2s)
attempt_n
amqp.Dial(RABBITMQ_URL) attempt=N
(backoff doubles: 1→2→4s ... max 30s)
connection accepted ✓
conn.Channel()
channel opened ✓
exhausted
N = maxAttempts=10 ?
log.Error("rabbitmq unreachable")
os.Exit(1) [FATAL]
=== declareTopology() — FATAL on fail ===
ch.ExchangeDeclare("url-shortener", type=topic, durable=true)
exchange declared ✓
ch.QueueDeclare("analytics.clicks", durable=true) [analytics-svc]
queue declared ✓
ch.QueueBind(routing_key="url.clicked")
binding created ✓
ch.QueueDeclare("notifications.events", durable=true) [notification-svc]
queue declared ✓




```
Exchange/Queue declaration sequence (correct):
  url-service     analytics-service    notification-service
       |                 |                    |
       ▼                 ▼                    ▼
  ExchangeDeclare    QueueDeclare         QueueDeclare
  "url-shortener"  "analytics.clicks"  "notifications.events"
                         |                    |
                    QueueBind            QueueBind × 3
                    url.clicked          url.created
                                         url.deleted
                                         milestone.reached
All three declarations are idempotent:
  → declaring an existing exchange/queue with same params = no-op
  → declaring with DIFFERENT params = error (caught at startup)
```




```
startup fatality matrix:
Service              | DB fail | Redis fail | RabbitMQ fail | Queue declare fail
---------------------|---------|------------|---------------|-------------------
url-service          |  FATAL  | WARN+cont  |    FATAL      |  N/A (producer)
analytics-service    |  FATAL  |    N/A     |    FATAL      |      FATAL
user-service         |  FATAL  |    N/A     |     N/A       |       N/A
notification-service |  FATAL  |    N/A     |    FATAL      |      FATAL
gateway              |   N/A   |    N/A     |     N/A       |       N/A
```

User Service: Registration, Login, JWT Issuance

Module ID: `url-shortener-m2`

1. Module Charter

This module implements the User Service as a fully self-contained authentication provider. It owns the `users` table in its dedicated PostgreSQL instance (`user_db`), hashes passwords with bcrypt at cost 12, issues HS256-signed JWT tokens on successful login, and exposes three HTTP endpoints: `POST /register` , `POST /login` , and `GET /me` . It produces a reusable `shared/auth` package containing the JWT verification middleware consumed by **all other services** to validate tokens locally without inter-service calls. This module does **not** publish domain events to RabbitMQ. It does **not** call any other service. It does **not** implement URL shortening, analytics, or notification logic. It does **not** maintain session state — every token verification is purely computational (HMAC-SHA256 + claims parsing), with no database round-trip after issuance. The User Service is the **only** service that writes to `user_db` ; all other services treat user identity as a verified JWT claim, never as a DB lookup. **Upstream dependencies:** `shared/logger` (structured JSON logging), `shared/auth` (produced by this module, consumed by others), PostgreSQL `user_db` (this service's exclusive database), `shared/events` package (imported by go.mod but unused in this module — no events emitted). **Downstream consumers:** Every other service imports `shared/auth` for the `JWTMiddleware` and `VerifyToken` function. The `JWT_SECRET` environment variable is the shared secret — it must be identical across all services. **Invariants that must always hold:**

- Plaintext passwords never appear in logs, HTTP responses, or the database.
- The bcrypt hash stored in `password_hash` column is never returned in any API response.
- Failed login always returns 401 with a generic message — the response must be identical whether the email does not exist or the password is wrong (timing-safe behavior at the application layer; bcrypt compare is constant-time internally).
- `user_id` in JWT is always a UUID v4 (not an integer); integer IDs would be enumerable.
- `JWT_SECRET` is read from environment at startup and stored in the Config struct; it is never read from `os.Getenv` in hot paths.

2. File Structure

Create files in the numbered order below. Shared packages first, then service files.

```
url-shortener/
│
├── shared/
│   └── auth/
│       ├── 1. go.mod          ← module: github.com/yourhandle/url-shortener/shared/auth
│       ├── 2. auth.go         ← Claims struct, TokenIssuer interface, VerifyToken func
│       └── 3. middleware.go   ← JWTMiddleware: func(http.Handler) http.Handler
│
├── services/
│   └── user-service/
│       ├── (existing from M1)
│       │   ├── go.mod         ← add shared/auth + golang-jwt/jwt dependency
│       │   ├── main.go        ← extend: add routes, wire handler deps
│       │   ├── config.go      ← extend: add JWTSecret, BCryptCost, TokenTTL fields
│       │   ├── db.go          ← (unchanged from M1)
│       │   └── health.go      ← (unchanged from M1)
│       │
│       ├── 4.  migration.sql  ← CREATE TABLE users + indexes
│       ├── 5.  store.go       ← UserRepository interface + pgxUserStore implementation
│       ├── 6.  password.go    ← PasswordHasher interface + bcryptHasher implementation
│       ├── 7.  token.go       ← jwtTokenIssuer implementing TokenIssuer
│       ├── 8.  handler.go     ← Handler struct with Register, Login, Me methods
│       ├── 9.  validate.go    ← email format + password length validators
│       ├── 10. errors.go      ← sentinel errors and writeError helper
│       └── 11. user_test.go   ← unit tests (store mock, handler tests, validator tests)
│
└── (M1 files unchanged: docker-compose.yml, gateway/, analytics-service/, etc.)
```

3. Complete Data Model

3.1 PostgreSQL Schema (`migration.sql`)

```sql
-- migration.sql (run once against user_db on startup or via init script)

CREATE TABLE IF NOT EXISTS users (

    id            UUID        PRIMARY KEY DEFAULT gen_random_uuid(),

    email         TEXT        UNIQUE NOT NULL,

    password_hash TEXT        NOT NULL,

    created_at    TIMESTAMPTZ NOT NULL DEFAULT now()

);

-- Supports: lookup by email on login (exact match, unique index)

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

-- Note: no index needed on id — it is the PRIMARY KEY (clustered B-tree by default in pgx).

-- Note: created_at has no index; no time-range queries are required for this service.
```

Column rationale:

Column	Type	Constraint	Why
`id`	UUID	PK, DEFAULT gen_random_uuid()	Non-enumerable identifier; safe to embed in JWT `sub` claim; no sequential leak
`email`	TEXT	UNIQUE NOT NULL	Login credential, must be unique; TEXT over VARCHAR because max email length (254 chars RFC 5321) is not worth constraining separately
`password_hash`	TEXT	NOT NULL	bcrypt output is always 60 chars but TEXT avoids coupling schema to algorithm output length
`created_at`	TIMESTAMPTZ	NOT NULL DEFAULT now()	Audit trail; timezone-aware

3.2 Go Structs — `shared/auth` Package

```go
// shared/auth/auth.go

package auth

import (

    "time"

)

// Claims is the JWT payload structure.

// These fields appear in every token issued by the User Service.

// Other services parse this after signature verification.

type Claims struct {

    // Sub is the user_id UUID string. Named "sub" per JWT RFC 7519 §4.1.2.

    Sub   string `json:"sub"`

    // Email is denormalized from the users table for convenience.

    // Avoids downstream services needing a DB lookup to display user context.

    Email string `json:"email"`

    // Iss is the issuer claim. Always "url-shortener".

    Iss   string `json:"iss"`

    // Iat is issued-at Unix timestamp (seconds).

    Iat   int64  `json:"iat"`

    // Exp is expiry Unix timestamp (seconds). Default: iat + 86400 (24h).

    Exp   int64  `json:"exp"`

}

// IsExpired returns true if the current wall time is past Exp.

// Does NOT verify the signature — that is done by VerifyToken.

func (c *Claims) IsExpired() bool {

    return time.Now().Unix() > c.Exp

}

// ExpiresAt converts the Exp Unix timestamp to a time.Time for response serialization.

func (c *Claims) ExpiresAt() time.Time {

    return time.Unix(c.Exp, 0).UTC()

}
```

```go
// TokenIssuer and VerifyToken live in auth.go to co-locate the contract with Claims.

// TokenIssuer abstracts JWT issuance. Allows test doubles.

// Implemented by jwtTokenIssuer in user-service/token.go.

type TokenIssuer interface {

    // Issue creates and signs a JWT for the given user.

    // Returns the signed token string and the expiry time, or an error.

    Issue(userID, email string) (tokenString string, expiresAt time.Time, err error)

    // Verify parses and validates a token string.

    // Returns the Claims on success, or an error if the token is expired,

    // has an invalid signature, or is malformed.

    Verify(tokenString string) (*Claims, error)

}
```

3.3 Go Structs — `user-service` Package

```go
// services/user-service/store.go

// User is the domain object. Never returned directly in HTTP responses.

type User struct {

    ID           string    // UUID string, e.g. "550e8400-e29b-41d4-a716-446655440000"

    Email        string

    PasswordHash string    // bcrypt hash; NEVER log this field

    CreatedAt    time.Time

}

// UserRepository abstracts all DB operations for the users table.

// Defined at the consumer (handler package), not the producer (pgxUserStore).

type UserRepository interface {

    // Insert creates a new user row. Returns ErrDuplicateEmail if email exists.

    Insert(ctx context.Context, email, passwordHash string) (*User, error)

    // FindByEmail retrieves a user by email.

    // Returns ErrUserNotFound if no row matches.

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

}
```

```go
// services/user-service/password.go

// PasswordHasher abstracts bcrypt to allow test doubles.
type PasswordHasher interface {
    // Hash returns a bcrypt hash of the plaintext password.
    // Cost is configured at construction time (default 12).
    Hash(plaintext string) (string, error)
    // Verify compares a plaintext password against a stored hash.
    // Returns nil on match, ErrPasswordMismatch on mismatch.
    // This call takes O(2^cost) CPU time — do not call in tight loops.
    Verify(plaintext, hash string) error
}
```

```go
// services/user-service/config.go (extended from M1)

package userservice

import (

    "fmt"

    "os"

    "strconv"

    "time"

)

type Config struct {

    DatabaseURL string        // required; fatal if empty

    Port        string        // default "8080"

    ServiceName string        // constant "user-service"

    JWTSecret   string        // required; HS256 signing key; min 32 bytes recommended

    BCryptCost  int           // default 12; range [10, 14] for this project

    TokenTTL    time.Duration // default 24h; configurable via TOKEN_TTL_HOURS env var

}

func loadConfig() (*Config, error) {

    cfg := &Config{

        DatabaseURL: os.Getenv("DATABASE_URL"),

        JWTSecret:   os.Getenv("JWT_SECRET"),

        Port:        envOrDefault("PORT", "8080"),

        ServiceName: "user-service",

        BCryptCost:  12,

        TokenTTL:    24 * time.Hour,

    }

    if cfg.DatabaseURL == "" {

        return nil, fmt.Errorf("DATABASE_URL is required")

    }

    if cfg.JWTSecret == "" {

        return nil, fmt.Errorf("JWT_SECRET is required")

    }

    if cost := os.Getenv("BCRYPT_COST"); cost != "" {

        n, err := strconv.Atoi(cost)

        if err != nil || n < 10 || n > 14 {

            return nil, fmt.Errorf("BCRYPT_COST must be integer in [10,14], got %q", cost)

        }

        cfg.BCryptCost = n
```

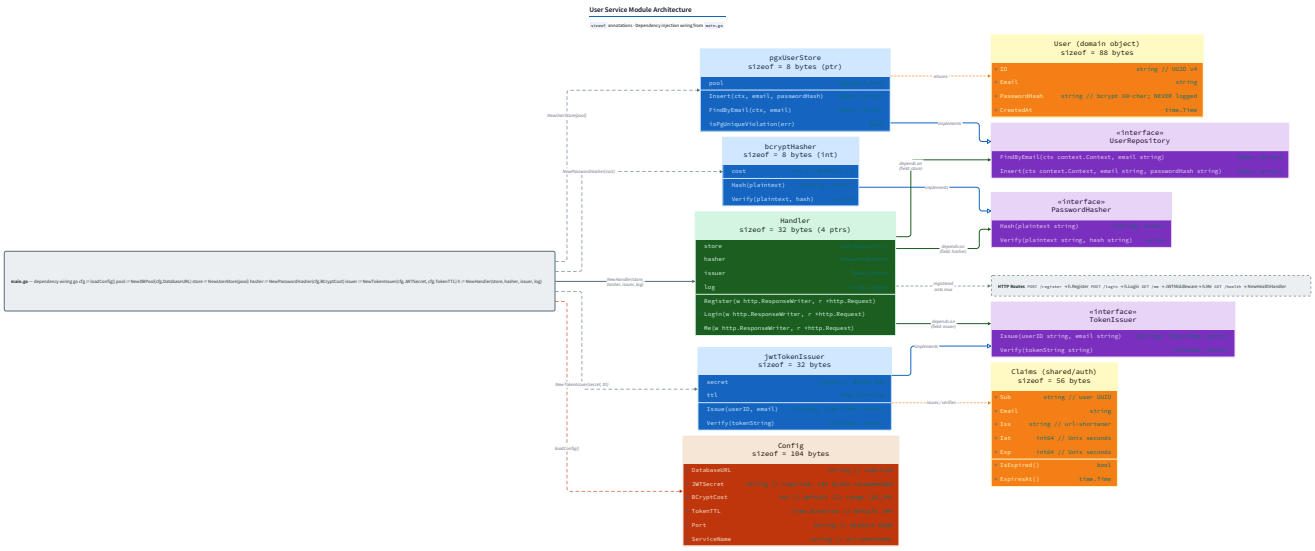
```
    }

    if ttlHours := os.Getenv("TOKEN_TTL_HOURS"); ttlHours != "" {

        h, err := strconv.Atoi(ttlHours)

        if err != nil || h < 1 {

            return nil, fmt.Errorf("TOKEN_TTL_HOURS must be positive integer, got %q", ttlHours)

        }

        cfg.TokenTTL = time.Duration(h) * time.Hour

    }

    return cfg, nil

}

func envOrDefault(key, def string) string {

    if v := os.Getenv(key); v != "" {

        return v

    }

    return def

}
```

3.4 HTTP Request/Response Schemas

```go
// services/user-service/handler.go — request/response types

type registerRequest struct {
    Email    string `json:"email"`
    Password string `json:"password"`
}

type registerResponse struct {
    UserID string `json:"user_id"`
    Email  string `json:"email"`
}

type loginRequest struct {
    Email    string `json:"email"`
    Password string `json:"password"`
}

type loginResponse struct {
    Token     string `json:"token"`
    ExpiresAt string `json:"expires_at"` // RFC3339 format: "2026-03-03T12:00:00Z"
}

type meResponse struct {
    UserID string `json:"user_id"`
    Email  string `json:"email"`
}

type errorResponse struct {
    Error string `json:"error"`
    // Field is present only for 400 validation errors, identifying which input field failed.
    Field string `json:"field,omitempty"`
}
```

3.5 Sentinel Errors (`errors.go`)

```go
// services/user-service/errors.go

package userservice

import "errors"

var (
    // ErrDuplicateEmail is returned by UserRepository.Insert when the email
    // already exists. Mapped to 409 Conflict in the handler.
    ErrDuplicateEmail = errors.New("email already registered")

    // ErrUserNotFound is returned by UserRepository.FindByEmail when no row matches.
    // MUST NOT be surfaced directly to HTTP clients (email enumeration).
    ErrUserNotFound = errors.New("user not found")

    // ErrPasswordMismatch is returned by PasswordHasher.Verify on wrong password.
    // MUST NOT be surfaced directly to HTTP clients (same 401 as ErrUserNotFound).
    ErrPasswordMismatch = errors.New("password mismatch")

    // ErrTokenInvalid is returned by TokenIssuer.Verify for any invalid token:
    // expired, wrong signature, malformed. Maps to 401 Unauthorized.
    ErrTokenInvalid = errors.New("token invalid or expired")
)
```



4. Interface Contracts

4.1 `UserRepository` — `pgxUserStore` Implementation

```go
// store.go

type pgxUserStore struct {

    pool *pgxpool.Pool

}

func NewUserStore(pool *pgxpool.Pool) UserRepository {

    return &pgxUserStore{pool: pool}

}
```

`Insert(ctx context.Context, email, passwordHash string) (*User, error)`

- Executes: `INSERT INTO users (email, password_hash) VALUES ($1, $2) RETURNING id, email, created_at`
- `email` must not be empty (caller validates before Insert call; store does not re-validate)
- `passwordHash` must not be empty (caller always hashes before calling Insert)
- On success: returns `*User` with `ID`, `Email`, `CreatedAt` populated; `PasswordHash` is **not** populated in the returned struct (read-once; no need to round-trip)
- On PostgreSQL error code `23505` (unique_violation): return `nil, ErrDuplicateEmail`
- On any other DB error: return `nil, fmt.Errorf("insert user: %w", err)` — do NOT return `ErrDuplicateEmail` for other errors
- Detecting pg error code:

```go
import "github.com/jackc/pgx/v5/pgconn"

func isPgUniqueViolation(err error) bool {

    var pgErr *pgconn.PgError

    return errors.As(err, &pgErr) && pgErr.Code == "23505"

}
```

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

- Executes: `SELECT id, email, password_hash, created_at FROM users WHERE email = $1`
- On no rows: return `nil, ErrUserNotFound`
- On success: return `*User` with all four fields populated (including `PasswordHash` — needed for bcrypt comparison)
- On any other DB error: return `nil, fmt.Errorf("find user by email: %w", err)`
- Uses `pool.QueryRow` + `pgx.ErrNoRows` detection:

```go
import "github.com/jackc/pgx/v5"

row := s.pool.QueryRow(ctx, `SELECT id, email, password_hash, created_at FROM users WHERE email = $1`, email)

var u User

err := row.Scan(&u.ID, &u.Email, &u.PasswordHash, &u.CreatedAt)

if errors.Is(err, pgx.ErrNoRows) {

    return nil, ErrUserNotFound

}

if err != nil {

    return nil, fmt.Errorf("find user by email: %w", err)

}

return &u, nil
```

4.2 `PasswordHasher` — `bcryptHasher` Implementation

```go
// password.go

type bcryptHasher struct {

    cost int

}

func NewPasswordHasher(cost int) PasswordHasher {

    return &bcryptHasher{cost: cost}

}
```

`Hash(plaintext string) (string, error)`

- Calls `bcrypt.GenerateFromPassword([]byte(plaintext), h.cost)`
- Returns the hash as a string (bcrypt output is always valid UTF-8, 60 chars, starts with `$2a$`)
- Returns `("", fmt.Errorf("hash password: %w", err))` on failure (extremely rare — only on system entropy failure)
- Does **not** validate password length or format — that is the validator's responsibility
- The returned hash string is what gets stored in `password_hash` column **`Verify(plaintext, hash string) error`**
- Calls `bcrypt.CompareHashAndPassword([]byte(hash), []byte(plaintext))`
- Returns `nil` on match
- Returns `ErrPasswordMismatch` on `bcrypt.ErrMismatchedHashAndPassword`
- Returns `fmt.Errorf("verify password: %w", err)` on any other error (malformed hash — should not occur in production)
- **Timing note:** `bcrypt.CompareHashAndPassword` is inherently slow (that is the security property). Do not add artificial delays. Do not short-circuit before calling bcrypt even when the user is not found — see Login handler algorithm in §5.2 for the dummy-hash strategy.

4.3 `TokenIssuer` — `jwtTokenIssuer` Implementation

```go
// token.go

type jwtTokenIssuer struct {

    secret []byte

    ttl    time.Duration

}

func NewTokenIssuer(secret string, ttl time.Duration) TokenIssuer {

    return &jwtTokenIssuer{secret: []byte(secret), ttl: ttl}

}
```

`Issue(userID, email string) (string, time.Time, error)`

- Algorithm: see §5.3
- Returns `("", time.Time{}, fmt.Errorf("sign token: %w", err))` on HMAC failure **`Verify(tokenString string) (*Claims, error)`**
- Algorithm: see §5.4
- Returns `(nil, ErrTokenInvalid)` for **all** failure modes (expired, wrong signature, malformed, wrong issuer)
- Must not return different errors for different failure reasons — all map to 401 with no hint

4.4 `shared/auth` — `JWTMiddleware`

```go
// shared/auth/middleware.go

// JWTMiddleware returns an http.Handler middleware that:
//   1. Reads the Authorization header
//   2. Strips the "Bearer " prefix
//   3. Verifies the token using the provided secret
//   4. On success: injects Claims into request context using claimsKey
//   5. On failure: writes 401 JSON and does NOT call next
//
// Parameters:
//   secret  - the HS256 signing secret (must match JWT_SECRET env var)
//   next    - the downstream handler to call on success
//
// Returns: http.Handler
func JWTMiddleware(secret string) func(http.Handler) http.Handler

// ClaimsFromContext extracts Claims injected by JWTMiddleware.
// Returns (nil, false) if the context has no claims (middleware not applied or failed).
func ClaimsFromContext(ctx context.Context) (*Claims, bool)

// claimsKey is the unexported context key type — prevents collisions with other packages.
type claimsKey struct{}
```

Authorization header parsing:

- Must be exactly `Authorization: Bearer <token>` (case-sensitive header name, case-sensitive scheme)
- Header absent → 401 `{"error": "authorization header required"}`
- Header present but not starting with `"Bearer "` (with space) → 401 `{"error": "invalid authorization header format"}`
- Token string empty after stripping prefix → 401 `{"error": "token is required"}`

4.5 `shared/auth` — `VerifyToken` Standalone Function

```go
// shared/auth/auth.go

// VerifyToken parses and verifies a JWT token string using the given secret.

// Used by services that verify tokens outside of HTTP middleware (e.g., in RabbitMQ consumers).

//

// Parameters:

//   tokenString - raw JWT string (no "Bearer " prefix)

//   secret      - HS256 signing key bytes

//

// Returns:

//   *Claims - populated on success

//   error   - ErrTokenInvalid on any failure (expired, wrong sig, malformed)

func VerifyToken(tokenString, secret string) (*Claims, error)
```

4.6 Validation Functions (`validate.go`)

```go
// services/user-service/validate.go

// validateEmail returns an error if email does not match a minimal valid format.

// Uses regexp: ^[^@\s]+@[^@\s]+\.[^@\s]+$

// Does NOT do DNS MX lookup — format check only.

// Returns nil on valid, non-nil with field="email" on invalid.

func validateEmail(email string) error

// validatePassword returns an error if password is shorter than 8 characters.

// No other complexity rules (no uppercase/symbol requirements per spec).

// Returns nil on valid, non-nil with field="password" on invalid.

func validatePassword(password string) error
```

5. Algorithm Specification

5.1 `POST /register` Handler

```
Input:  JSON body { email: string, password: string }
Output: 201 { user_id, email } | 400 | 409 | 500
Step 1: Decode JSON body
  - json.NewDecoder(r.Body).Decode(&req)
  - On error (malformed JSON, wrong types): writeError(w, 400, "invalid request body", "")
  - Limit body to 1MB: r.Body = http.MaxBytesReader(w, r.Body, 1<<20)
Step 2: Validate inputs
  - if err := validateEmail(req.Email); err != nil:
      writeError(w, 400, "invalid email format", "email")
      return
  - if err := validatePassword(req.Password); err != nil:
      writeError(w, 400, "password must be at least 8 characters", "password")
      return
Step 3: Hash password
  - hash, err := h.hasher.Hash(req.Password)
  - On error: log.Error("bcrypt hash failed", "error", err)
              writeError(w, 500, "internal server error", "")
              return
Step 4: Insert user
  - user, err := h.store.Insert(r.Context(), req.Email, hash)
  - if errors.Is(err, ErrDuplicateEmail):
      writeError(w, 409, "email already registered", "")
      return
  - On other error: log.Error("insert user failed", "error", err)
                    writeError(w, 500, "internal server error", "")
                    return
Step 5: Write response
  - w.Header().Set("Content-Type", "application/json")
  - w.WriteHeader(201)
  - json.NewEncoder(w).Encode(registerResponse{UserID: user.ID, Email: user.Email})
Invariants after execution:
  - user.PasswordHash never appears in the HTTP response
  - On 409: no new row was inserted (constraint violated → rolled back by Postgres)
  - On 500: caller may retry; handler is idempotent per email (duplicate → 409)
```

5.2 `POST /login` Handler — Timing-Safe Design

```
Input:  JSON body { email: string, password: string }
Output: 200 { token, expires_at } | 401
Step 1: Decode JSON body (same as /register step 1)
Step 2: Attempt to find user
  - user, findErr := h.store.FindByEmail(r.Context(), req.Email)
  - Do NOT return 401 here yet — must always call bcrypt compare (see Step 3)
Step 3: Constant-time password comparison
  - if errors.Is(findErr, ErrUserNotFound):
      // Dummy compare: prevents timing difference that reveals email existence.
      // bcrypt.CompareHashAndPassword with dummyHash takes same wall time as real compare.
      dummyHash := "$2a$12$invalidhashfortimingsafetyonlyxx"
      h.hasher.Verify(req.Password, dummyHash)  // result intentionally discarded
      writeError(w, 401, "invalid credentials", "")
      return
  - if findErr != nil:
      log.Error("find user failed", "error", findErr)
      writeError(w, 500, "internal server error", "")
      return
  // User found:
  - if err := h.hasher.Verify(req.Password, user.PasswordHash); err != nil:
      // errors.Is(err, ErrPasswordMismatch) OR unexpected error: both → 401
      writeError(w, 401, "invalid credentials", "")
      return
Step 4: Issue token
  - tokenString, expiresAt, err := h.issuer.Issue(user.ID, user.Email)
  - On error: log.Error("token issue failed", "error", err)
              writeError(w, 500, "internal server error", "")
              return
Step 5: Write response
  - w.Header().Set("Content-Type", "application/json")
  - w.WriteHeader(200)
  - json.NewEncoder(w).Encode(loginResponse{
        Token:     tokenString,
        ExpiresAt: expiresAt.Format(time.RFC3339),
    })
SECURITY NOTE on dummy hash:
  The dummyHash literal must NOT be a valid bcrypt hash that could match any password.
  Use a string that starts with "$2a$12$" (correct prefix) but has an invalid/impossible
  body so bcrypt parse succeeds (cost extraction works) but comparison always fails.
  Example: "$2a$12$000000000000000000000uZLbwxnpY0o6Fh.za8VOf/OjIPFGXGhG"
  (31 chars of deterministic non-random salt+hash data — bcrypt will always return mismatch)
  This ensures the timing profile is identical to a real failed compare.
```

5.3 `TokenIssuer.Issue` Algorithm

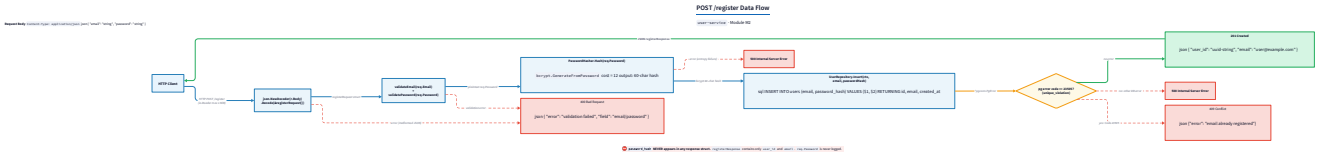
```
Input:  userID string (UUID), email string
Output: tokenString string, expiresAt time.Time, error
Step 1: Build Claims
  now := time.Now().UTC()
  claims := Claims{
      Sub:   userID,
      Email: email,
      Iss:   "url-shortener",
      Iat:   now.Unix(),
      Exp:   now.Add(i.ttl).Unix(),
  }
Step 2: Create JWT token using golang-jwt/jwt
  // Use map claims to avoid importing jwt in shared/auth — embed in jwtTokenIssuer
  token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
      "sub":   claims.Sub,
      "email": claims.Email,
      "iss":   claims.Iss,
      "iat":   claims.Iat,
      "exp":   claims.Exp,
  })
Step 3: Sign
  signed, err := token.SignedString(i.secret)
  if err != nil:
      return "", time.Time{}, fmt.Errorf("sign token: %w", err)
Step 4: Return
  expiresAt := time.Unix(claims.Exp, 0).UTC()
  return signed, expiresAt, nil
Token format (HS256, compact serialization):
  <base64url(header)>.<base64url(payload)>.<base64url(signature)>
  Header: {"alg":"HS256","typ":"JWT"}
  Payload: {"sub":"<uuid>","email":"...","iss":"url-shortener","iat":<unix>,"exp":<unix>}
  Signature: HMAC-SHA256(base64url(header) + "." + base64url(payload), secret)
```

5.4 `TokenIssuer.Verify` / `shared/auth.VerifyToken` Algorithm

```
Input:  tokenString string, secret []byte
Output: *Claims, error
Step 1: Parse token with golang-jwt/jwt
  token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
      // Verify algorithm is HS256 — MANDATORY check
      if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
          return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"])
      }
      return secret, nil
  })
Step 2: Handle parse error
  if err != nil:
      return nil, ErrTokenInvalid
      // NOTE: do not wrap err — all failures are opaque ErrTokenInvalid to callers
Step 3: Check token.Valid
  if !token.Valid:
      return nil, ErrTokenInvalid
Step 4: Extract MapClaims
  mapClaims, ok := token.Claims.(jwt.MapClaims)
  if !ok:
      return nil, ErrTokenInvalid
Step 5: Build Claims struct
  claims := &Claims{}
  claims.Sub   = mapClaims["sub"].(string)   // type assert; on panic → see Step 5a
  claims.Email = mapClaims["email"].(string)
  claims.Iss   = mapClaims["iss"].(string)
  // iat and exp are float64 in JSON-decoded maps
  if iat, ok := mapClaims["iat"].(float64); ok:
      claims.Iat = int64(iat)
  if exp, ok := mapClaims["exp"].(float64); ok:
      claims.Exp = int64(exp)
Step 5a: Safe extraction to avoid panic
  Use a helper that returns (string, bool) and check ok before asserting:
  func extractString(m jwt.MapClaims, key string) (string, bool) {
      v, exists := m[key]
      if !exists { return "", false }
      s, ok := v.(string)
      return s, ok
  }
  If any required claim is absent or wrong type: return nil, ErrTokenInvalid
Step 6: Verify issuer
  if claims.Iss != "url-shortener":
      return nil, ErrTokenInvalid
Step 7: Return
  return claims, nil
NOTE: golang-jwt/jwt automatically validates exp (token expired) during Parse
      when the jwt.Parser's ValidateExpiry option is enabled (default true).
      No manual time comparison is needed in this function.
```

5.5 `GET /me` Handler

```
Input:  Authorization: Bearer <jwt> in request header
        (Claims already in context, injected by JWTMiddleware)
Output: 200 { user_id, email } | 401
Step 1: Extract claims from context
  claims, ok := auth.ClaimsFromContext(r.Context())
  if !ok:
      // Should not happen if middleware is correctly applied, but guard anyway
      writeError(w, 401, "unauthorized", "")
      return
Step 2: Write response (no DB lookup)
  w.Header().Set("Content-Type", "application/json")
  w.WriteHeader(200)
  json.NewEncoder(w).Encode(meResponse{
      UserID: claims.Sub,
      Email:  claims.Email,
  })
```




```
POST /login timing-safe flow:
FindByEmail(email)
       |
  ┌────┴──────────────────────────────────────────────────┐
  | User NOT found                  User found            |
  |       |                              |                |
  |       ▼                              ▼                |
  |  Verify(password,             Verify(password,        |
  |  dummyHash)                   user.PasswordHash)      |
  |  [always fails,               [bcrypt.CompareHash...] |
  |   takes ~same time]                |          |      |
  |       |                          match      mismatch  |
  |       ▼                            |            |    |
  |  401 generic                Issue(id, email)   401 generic
  |                                    |
  |                             200 {token, expires_at}
  └──────────────────────────────────────────────────────┘
CRITICAL: Both 401 paths go through bcrypt.CompareHashAndPassword.
          Network latency variation masks the residual timing difference
          between a valid bcrypt hash and the dummy hash structure,
          but using the same cost prefix minimizes it further.
```

5.6 Schema Migration on Service Startup

The user-service does not use a migration tool in M2. Migration is applied via a startup SQL exec:

```go
// main.go — after NewDBPool succeeds

const schema = `

CREATE TABLE IF NOT EXISTS users (

    id            UUID        PRIMARY KEY DEFAULT gen_random_uuid(),

    email         TEXT        UNIQUE NOT NULL,

    password_hash TEXT        NOT NULL,

    created_at    TIMESTAMPTZ NOT NULL DEFAULT now()

);

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

`

func runMigrations(ctx context.Context, pool *pgxpool.Pool, log *slog.Logger) error {

    _, err := pool.Exec(ctx, schema)

    if err != nil {

        return fmt.Errorf("run migrations: %w", err)

    }

    log.Info("migrations applied")

    return nil

}
```

Called in `main()` after `NewDBPool`, before starting the HTTP server. Fatal on error

6. Error Handling Matrix

Error Scenario	Detected By	HTTP Status	Response Body	Log Level	Log Fields
Missing `DATABASE_URL`	`loadConfig()`	— (pre-HTTP)	stderr message	`Error` (fmt.Fprintf)	env var name
Missing `JWT_SECRET`	`loadConfig()`	— (pre-HTTP)	stderr message	`Error` (fmt.Fprintf)	env var name
DB unreachable at startup	`NewDBPool` → `pool.Ping`	— (pre-HTTP)	—	`Error` + `os.Exit(1)`	masked DSN
Migration SQL fails	`runMigrations`	— (pre-HTTP)	—	`Error` + `os.Exit(1)`	error string
Malformed JSON request body	`json.Decode` in handler	400	`{"error":"invalid request body"}`	`Warn`	path, body size
Invalid email format	`validateEmail`	400	`{"error":"invalid email format","field":"email"}`	none	—
Password < 8 chars	`validatePassword`	400	`{"error":"password must be at least 8 characters","field":"password"}`	none	—
Duplicate email on register	`isPgUniqueViolation` in store	409	`{"error":"email already registered"}`	`Info`	email (safe to log)
DB error on Insert	`store.Insert` non-unique error	500	`{"error":"internal server error"}`	`Error`	error string
User not found on login	`store.FindByEmail` → `ErrUserNotFound`	401	`{"error":"invalid credentials"}`	none (timing-safe: log would reveal)	—
Wrong password on login	`hasher.Verify` → `ErrPasswordMismatch`	401	`{"error":"invalid credentials"}`	none	—
DB error on FindByEmail	`store.FindByEmail` non-ErrUserNotFound	500	`{"error":"internal server error"}`	`Error`	error string
bcrypt hash failure (Hash)	`bcrypt.GenerateFromPassword`	500	`{"error":"internal server error"}`	`Error`	error string
bcrypt unexpected verify error	`bcrypt.CompareHashAndPassword` non-mismatch	401	`{"error":"invalid credentials"}`	`Error`	error (malformed stored hash — serious)
Token sign failure	`token.SignedString`	500	`{"error":"internal server error"}`	`Error`	error string
Missing Authorization header	`JWTMiddleware`	401	`{"error":"authorization header required"}`	none	—

Error Scenario	Detected By	HTTP Status	Response Body	Log Level	Log Fields
Malformed Authorization header	`JWTMiddleware`	401	`{"error":"invalid authorization header format"}`	none	—
JWT signature invalid	`jwt.Parse` error	401	`{"error":"unauthorized"}`	none	—
JWT expired	`jwt.Parse` exp validation	401	`{"error":"unauthorized"}`	none	—
JWT wrong issuer	claims.Iss check	401	`{"error":"unauthorized"}`	none	—
`GET /me` missing claims in context	`ClaimsFromContext`	401	`{"error":"unauthorized"}`	`Warn`	path

`writeError` helper:

```go
// services/user-service/errors.go

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

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

    w.WriteHeader(status)

    resp := errorResponse{Error: message, Field: field}

    json.NewEncoder(w).Encode(resp) // encode error intentionally ignored

}
```

Security note: 401 responses for login always use the message `"invalid credentials"` regardless of whether the email was not found or the password was wrong. The log lines for these two paths are also identical (neither logs). This prevents log-based email enumeration

7. Threat Model

Attack	Surface	Mitigation
Password disclosure	HTTP response	`password_hash` field never serialized in any response type
Password disclosure	Logs	`req.Password` and `user.PasswordHash` never passed to any logger
Email enumeration via login timing	POST /login	Dummy bcrypt compare on unknown email; same response body and status for both paths
Email enumeration via login response	POST /login	Identical 401 body `{"error":"invalid credentials"}` for both cases
Email enumeration via register	POST /register	409 on duplicate does reveal email existence — this is documented and acceptable per spec (user is self-registering)
JWT algorithm confusion	`jwt.Parse` key func	`if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok { return nil, err }` — rejects RS256/none
JWT `none` algorithm attack	`jwt.Parse` key func	Same check above rejects `jwt.SigningMethodNone`
JWT secret brute force	JWT_SECRET env	Secret must be ≥ 32 bytes (loaded from env; weak secrets are an ops problem, not caught in code)
Integer user_id enumeration	JWT `sub` claim	UUID v4, not sequential integer
Hardcoded JWT_SECRET	Config	`loadConfig` enforces `os.Getenv("JWT_SECRET")` — fatal if empty; no default value

8. Concurrency Specification

The User Service is stateless at the handler level. `net/http`'s default server dispatches each request in its own goroutine. The shared state components are:

Component	Thread Safety	Rationale
`pgxpool.Pool`	✅ Safe	pgxpool is goroutine-safe; connection acquisition uses internal mutex
`bcryptHasher`	✅ Safe	No mutable state; `bcrypt.GenerateFromPassword` and `CompareHashAndPassword` are pure functions
`jwtTokenIssuer`	✅ Safe	`secret []byte` is read-only after construction; `golang-jwt/jwt` operations are pure
`pgxUserStore`	✅ Safe	Only holds `*pgxpool.Pool` which is safe
`Handler` struct	✅ Safe	All fields set at construction; no mutation after `main()` wires dependencies
No mutexes are required in this module. The `JWTMiddleware` uses `context.WithValue` which is goroutine-safe. The pre-encoded health response byte slice in `NewHealthHandler` is read-only after construction.		
bcrypt CPU saturation: At cost 12, each `bcrypt.GenerateFromPassword` call takes ~200–400ms of CPU. Under concurrent load (N simultaneous logins), N goroutines will each consume one CPU for ~300ms. `net/http`'s goroutine-per-request model means the OS scheduler handles this automatically; no explicit worker pool is needed at this traffic level. If load becomes a concern in future milestones, a semaphore can limit concurrent bcrypt operations — not required for M2.		

9. Implementation Sequence with Checkpoints

Phase 1: `shared/auth` Package + `users` Table Migration (1–1.5h)

1. Create `shared/auth/go.mod`:

```
module github.com/yourhandle/url-shortener/shared/auth
go 1.23
require github.com/golang-jwt/jwt/v5 v5.2.1
```

1. Write `shared/auth/auth.go`: `Claims` struct, `IsExpired()`, `ExpiresAt()`, `TokenIssuer` interface, `VerifyToken` function, `ErrTokenInvalid` sentinel.
2. Write `shared/auth/middleware.go`: `JWTMiddleware`, `ClaimsFromContext`, `claimsKey` type.
3. Add `shared/auth` to `go.work`:

```
use (
    ...
    ./shared/auth
)
```

1. Write `services/user-service/migration.sql` as specified in §3.1.
2. Update `services/user-service/go.mod`:

```
require (
    github.com/yourhandle/url-shortener/shared/auth v0.0.0
    github.com/yourhandle/url-shortener/shared/logger v0.0.0
    github.com/jackc/pgx/v5 v5.6.0
    golang.org/x/crypto v0.23.0   // for bcrypt
    github.com/golang-jwt/jwt/v5 v5.2.1
)
replace (
    github.com/yourhandle/url-shortener/shared/auth => ../../shared/auth
    github.com/yourhandle/url-shortener/shared/logger => ../../shared/logger
)
```

1. Run `go work sync` at monorepo root. **Checkpoint:** `go build github.com/yourhandle/url-shortener/shared/auth` exits 0. Write a throwaway test that calls `VerifyToken("bad-token", "secret")` and asserts it returns `ErrTokenInvalid`. Run: `go test ./shared/auth/...` → PASS.

Phase 2: `store.go`, `password.go`, `token.go`, `errors.go` (1–1.5h)

1. Write `services/user-service/errors.go` with four sentinel errors and `writeError` helper.
2. Write `services/user-service/store.go`: `User` struct, `UserRepository` interface, `pgxUserStore`, `isPgUniqueViolation` helper.
3. Write `services/user-service/password.go`: `PasswordHasher` interface, `bcryptHasher`, both methods.
4. Write `services/user-service/token.go`: `jwtTokenIssuer`, `Issue` and `Verify` methods. **Checkpoint:** `go build ./services/user-service/...` exits 0. Run unit tests for store (with mock), password hasher (Hash → Verify round trip), and token issuer (Issue → Verify round trip). See §10 for test specs. Run: `go test ./services/user-service/...` → PASS.

Phase 3: `validate.go` + `handler.go` + Updated `config.go` and `main.go` (1.5–2h)

1. Write `services/user-service/validate.go` with `validateEmail` and `validatePassword`.
2. Update `services/user-service/config.go` to add `JWTSecret`, `BCryptCost`, `TokenTTL`.
3. Write `services/user-service/handler.go`:
   - `Handler` struct holding `store UserRepository`, `hasher PasswordHasher`, `issuer TokenIssuer`, `log *slog.Logger`
   - `NewHandler` constructor
   - `Register`, `Login`, `Me` methods
4. Update `services/user-service/main.go`:
   - Load config (now includes JWT fields)
   - Call `runMigrations`
   - Construct `NewUserStore`, `NewPasswordHasher`, `NewTokenIssuer`, `NewHandler`
   - Register routes: `POST /register`, `POST /login`, `GET /me` (wrapped in `JWTMiddleware`)
   - Remove old stub routes if any

```go
// main.go route registration

mux.HandleFunc("POST /register", h.Register)

mux.HandleFunc("POST /login", h.Login)

mux.Handle("GET /me", auth.JWTMiddleware(cfg.JWTSecret)(http.HandlerFunc(h.Me)))

mux.HandleFunc("GET /health", NewHealthHandler(cfg.ServiceName))
```

Checkpoint: Start service locally:

```bash
DATABASE_URL="postgres://useruser:userpass@localhost:5434/userdb" \

JWT_SECRET="supersecretkey-at-least-32-chars-long" \

go run ./services/user-service/
```

`curl http://localhost:8080/health` → `{"status":"ok","service":"user-service"}`. `curl -X POST http://localhost:8080/register -d '{"email":"test@example.com","password":"password123"}' -H 'Content-Type:`

`application/json'` → `201 {"user_id":"...","email":"test@example.com"}`.

Phase 4: `JWTMiddleware` Integration + `GET /me` (1h)

1. Verify `JWTMiddleware` in `shared/auth/middleware.go` correctly injects claims into context.
2. Verify `ClaimsFromContext` returns claims from context.
3. Wire `GET /me` handler using the middleware (done in Phase 3 above).
4. Test `GET /me` with a valid token and with an expired/invalid token. **Checkpoint:** Full manual flow:

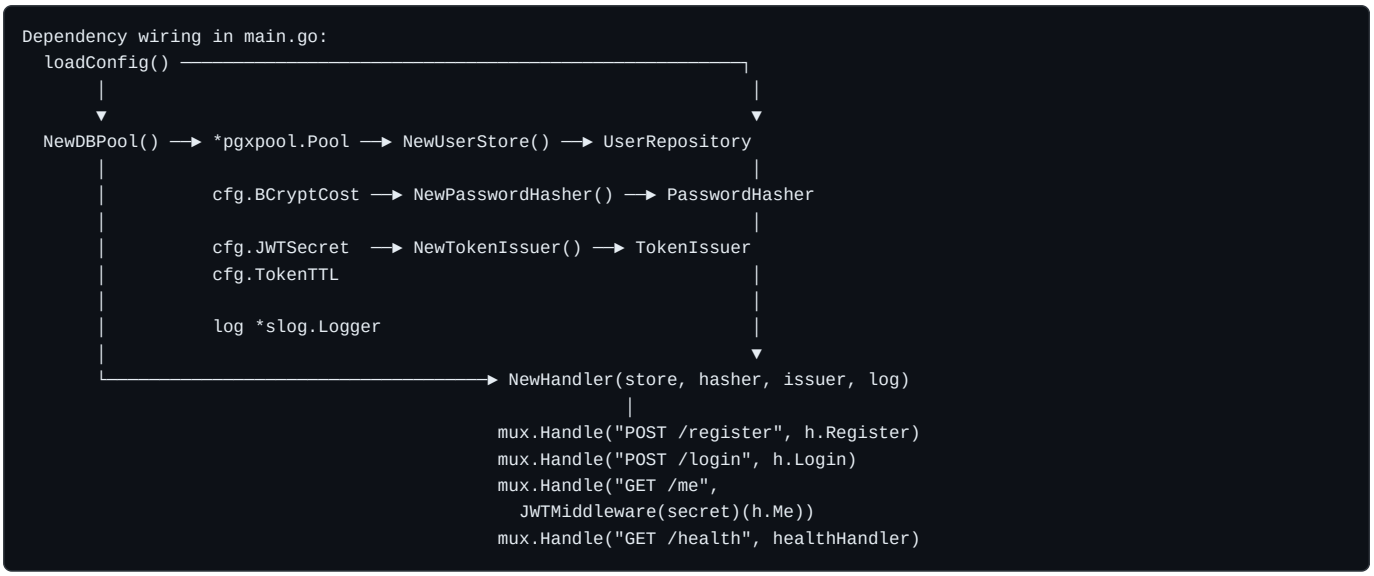
```bash
# Register
RESP=$(curl -s -X POST http://localhost:8080/register \
  -d '{"email":"user@example.com","password":"securepass"}' \
  -H 'Content-Type: application/json')
echo $RESP  # {"user_id":"<uuid>","email":"user@example.com"}
# Login
TOKEN_RESP=$(curl -s -X POST http://localhost:8080/login \
  -d '{"email":"user@example.com","password":"securepass"}' \
  -H 'Content-Type: application/json')
TOKEN=$(echo $TOKEN_RESP | python3 -c "import sys,json; print(json.load(sys.stdin)['token'])")
# Me
curl -s http://localhost:8080/me \
  -H "Authorization: Bearer $TOKEN"
# → {"user_id":"<uuid>","email":"user@example.com"}
```

Phase 5: Integration Test (1–2h)

Write `services/user-service/user_test.go` covering full round trip. See §10 for detailed test specifications. Run with Docker-started `user_db`:

```bash
DATABASE_URL="postgres://useruser:userpass@localhost:5434/userdb" \
JWT_SECRET="testsecret-at-least-32-chars-long-ok" \
go test -v -run TestIntegration ./services/user-service/...
```

Checkpoint: All integration tests PASS. No test leaks connections (each test uses `t.Cleanup`). {{DIAGRAM:tdd-diag-9}}


Dependency wiring in main.go:
loadConfig()
NewDBPool() ──▶ *pgxpool.Pool ──▶ NewUserStore() ──▶ UserRepository
cfg.BCryptCost ──▶ NewPasswordHasher() ──▶ PasswordHasher
cfg.JWTSecret ──▶ NewTokenIssuer() ──▶ TokenIssuer
cfg.TokenTTL
log *slog.Logger
NewHandler(store, hasher, issuer, log)
mux.Handle("POST /register", h.Register)
mux.Handle("POST /login", h.Login)
mux.Handle("GET /me",
JWTMiddleware(secret)(h.Me))
mux.Handle("GET /health", healthHandler)

10. Test Specification

10.1 Unit Tests — Validators (`validate_test.go`)

```go
func TestValidateEmail(t *testing.T) {
    cases := []struct {
        input   string
        wantErr bool
    }{
        {"user@example.com", false},
        {"user+tag@sub.domain.org", false},
        {"", true},               // empty
        {"notanemail", true},     // no @
        {"@nodomain.com", true},  // empty local part
        {"user@", true},          // no domain
        {"user @example.com", true}, // space in email
    }
    for _, tc := range cases {
        t.Run(tc.input, func(t *testing.T) {
            err := validateEmail(tc.input)
            if (err != nil) != tc.wantErr {
                t.Errorf("validateEmail(%q) err=%v, wantErr=%v", tc.input, err, tc.wantErr)
            }
        })
    }
}
func TestValidatePassword(t *testing.T) {
    cases := []struct {
        input   string
        wantErr bool
    }{
        {"12345678", false},    // exactly 8 chars
        {"longerpassword", false},
        {"1234567", true},      // 7 chars
        {"", true},             // empty
        {"       7", false},    // 8 spaces — valid (no complexity rules)
    }
    for _, tc := range cases {
```

```
        t.Run(fmt.Sprintf("len%d", len(tc.input)), func(t *testing.T) {
            err := validatePassword(tc.input)
            if (err != nil) != tc.wantErr {
                t.Errorf("validatePassword(%q) err=%v, wantErr=%v", tc.input, err, tc.wantErr)
            }
        })
    }
}
```

10.2 Unit Tests — `bcryptHasher`

```go
func TestBcryptHasher_HashAndVerify(t *testing.T) {
    h := NewPasswordHasher(bcrypt.MinCost) // cost=4 for test speed
    hash, err := h.Hash("mypassword")
    if err != nil {
        t.Fatalf("Hash: %v", err)
    }
    if hash == "" {
        t.Fatal("hash is empty")
    }
    if err := h.Verify("mypassword", hash); err != nil {
        t.Errorf("Verify correct password: %v", err)
    }
    if err := h.Verify("wrongpassword", hash); !errors.Is(err, ErrPasswordMismatch) {
        t.Errorf("Verify wrong password: got %v, want ErrPasswordMismatch", err)
    }
}
func TestBcryptHasher_DifferentHashesSamePassword(t *testing.T) {
    h := NewPasswordHasher(bcrypt.MinCost)
    hash1, _ := h.Hash("samepassword")
    hash2, _ := h.Hash("samepassword")
    if hash1 == hash2 {
        t.Error("bcrypt should produce different hashes (different salts)")
    }
    // Both should verify correctly
    if err := h.Verify("samepassword", hash1); err != nil {
        t.Errorf("hash1 verify: %v", err)
    }
    if err := h.Verify("samepassword", hash2); err != nil {
        t.Errorf("hash2 verify: %v", err)
    }
}
```

10.3 Unit Tests — `jwtTokenIssuer`

```go
func TestJWTTokenIssuer_IssueAndVerify(t *testing.T) {
    issuer := NewTokenIssuer("test-secret-32-chars-long-exactly", 24*time.Hour)
    tokenStr, expiresAt, err := issuer.Issue("user-uuid-123", "user@example.com")
    if err != nil {
        t.Fatalf("Issue: %v", err)
    }
    if tokenStr == "" {
        t.Fatal("token string is empty")
    }
    if time.Until(expiresAt) < 23*time.Hour {
        t.Errorf("expiresAt too soon: %v", expiresAt)
    }
    claims, err := issuer.Verify(tokenStr)
    if err != nil {
        t.Fatalf("Verify: %v", err)
    }
    if claims.Sub != "user-uuid-123" {
        t.Errorf("sub: got %q", claims.Sub)
    }
    if claims.Email != "user@example.com" {
        t.Errorf("email: got %q", claims.Email)
    }
    if claims.Iss != "url-shortener" {
        t.Errorf("iss: got %q", claims.Iss)
    }
}
func TestJWTTokenIssuer_Verify_InvalidSignature(t *testing.T) {
    issuer1 := NewTokenIssuer("secret-one-32-chars-long-exactly", 24*time.Hour)
    issuer2 := NewTokenIssuer("secret-two-32-chars-long-exactly", 24*time.Hour)
    tok, _, _ := issuer1.Issue("uid", "u@e.com")
    _, err := issuer2.Verify(tok)
    if !errors.Is(err, ErrTokenInvalid) {
        t.Errorf("wrong secret verify: got %v, want ErrTokenInvalid", err)
    }
}
```

```go
func TestJWTTokenIssuer_Verify_Expired(t *testing.T) {
    issuer := NewTokenIssuer("test-secret-32-chars-long-exactly", -1*time.Hour) // negative TTL → already expired
    tok, _, _ := issuer.Issue("uid", "u@e.com")
    _, err := issuer.Verify(tok)
    if !errors.Is(err, ErrTokenInvalid) {
        t.Errorf("expired token verify: got %v, want ErrTokenInvalid", err)
    }
}
func TestJWTTokenIssuer_Verify_Malformed(t *testing.T) {
    issuer := NewTokenIssuer("test-secret-32-chars-long-exactly", 24*time.Hour)
    _, err := issuer.Verify("not.a.jwt")
    if !errors.Is(err, ErrTokenInvalid) {
        t.Errorf("malformed token: got %v, want ErrTokenInvalid", err)
    }
}
```

10.4 Unit Tests — Handlers (with mock store)

```go
// Mock implementations for handler unit tests:

type mockStore struct {

    insertFn      func(ctx context.Context, email, hash string) (*User, error)

    findByEmailFn func(ctx context.Context, email string) (*User, error)

}

func (m *mockStore) Insert(ctx context.Context, email, hash string) (*User, error) {

    return m.insertFn(ctx, email, hash)

}

func (m *mockStore) FindByEmail(ctx context.Context, email string) (*User, error) {

    return m.findByEmailFn(ctx, email)

}

func TestRegisterHandler_Success(t *testing.T) {

    store := &mockStore{

        insertFn: func(ctx context.Context, email, hash string) (*User, error) {

            return &User{ID: "test-uuid", Email: email, CreatedAt: time.Now()}, nil

        },

    }

    issuer := NewTokenIssuer("test-secret-32-chars-long-exactly", time.Hour)

    h := NewHandler(store, NewPasswordHasher(bcrypt.MinCost), issuer, slog.Default())

    body := `{"email":"test@example.com","password":"securepass"}`

    req := httptest.NewRequest("POST", "/register", strings.NewReader(body))

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

    rec := httptest.NewRecorder()

    h.Register(rec, req)

    if rec.Code != 201 {

        t.Errorf("status: got %d want 201, body: %s", rec.Code, rec.Body.String())

    }

    var resp registerResponse

    json.Unmarshal(rec.Body.Bytes(), &resp)

    if resp.UserID != "test-uuid" {

        t.Errorf("user_id: got %q", resp.UserID)

    }

    // Ensure PasswordHash is NOT in response:

    if strings.Contains(rec.Body.String(), "password") {

        t.Error("response body must not contain password or password_hash")
```

```
    }
}
func TestRegisterHandler_DuplicateEmail(t *testing.T) {
    store := &mockStore{
        insertFn: func(_ context.Context, _, _ string) (*User, error) {
            return nil, ErrDuplicateEmail
        },
    }
    h := NewHandler(store, NewPasswordHasher(bcrypt.MinCost), nil, slog.Default())
    body := `{"email":"dup@example.com","password":"password123"}`
    req := httptest.NewRequest("POST", "/register", strings.NewReader(body))
    req.Header.Set("Content-Type", "application/json")
    rec := httptest.NewRecorder()
    h.Register(rec, req)
    if rec.Code != 409 {
        t.Errorf("status: got %d want 409", rec.Code)
    }
}
func TestRegisterHandler_ShortPassword(t *testing.T) {
    h := NewHandler(nil, nil, nil, slog.Default())
    body := `{"email":"user@example.com","password":"short"}`
    req := httptest.NewRequest("POST", "/register", strings.NewReader(body))
    req.Header.Set("Content-Type", "application/json")
    rec := httptest.NewRecorder()
    h.Register(rec, req)
    if rec.Code != 400 {
        t.Errorf("status: got %d want 400", rec.Code)
    }
    var errResp errorResponse
    json.Unmarshal(rec.Body.Bytes(), &errResp)
    if errResp.Field != "password" {
        t.Errorf("field: got %q want password", errResp.Field)
    }
}
func TestLoginHandler_InvalidCredentials_UnknownEmail(t *testing.T) {
    store := &mockStore{
```

```
        findByEmailFn: func(_ context.Context, _ string) (*User, error) {
            return nil, ErrUserNotFound
        },
    }
    h := NewHandler(store, NewPasswordHasher(bcrypt.MinCost), nil, slog.Default())
    body := `{"email":"nobody@example.com","password":"somepassword"}`
    req := httptest.NewRequest("POST", "/login", strings.NewReader(body))
    req.Header.Set("Content-Type", "application/json")
    rec := httptest.NewRecorder()
    h.Login(rec, req)
    if rec.Code != 401 {
        t.Errorf("status: got %d want 401", rec.Code)
    }
    // Verify response body is generic — no hint of user existence
    var errResp errorResponse
    json.Unmarshal(rec.Body.Bytes(), &errResp)
    if errResp.Error != "invalid credentials" {
        t.Errorf("error message: got %q, want opaque message", errResp.Error)
    }
}
func TestLoginHandler_WrongPassword(t *testing.T) {
    hash, _ := bcrypt.GenerateFromPassword([]byte("correctpassword"), bcrypt.MinCost)
    store := &mockStore{
        findByEmailFn: func(_ context.Context, _ string) (*User, error) {
            return &User{ID: "uid", Email: "u@e.com", PasswordHash: string(hash)}, nil
        },
    }
    h := NewHandler(store, NewPasswordHasher(bcrypt.MinCost), nil, slog.Default())
    body := `{"email":"u@e.com","password":"wrongpassword"}`
    req := httptest.NewRequest("POST", "/login", strings.NewReader(body))
    req.Header.Set("Content-Type", "application/json")
    rec := httptest.NewRecorder()
    h.Login(rec, req)
    if rec.Code != 401 {
        t.Errorf("status: got %d want 401", rec.Code)
    }
```

```go
}

func TestMeHandler_ValidToken(t *testing.T) {

    claims := &auth.Claims{Sub: "user-uuid", Email: "user@example.com", Iss: "url-shortener"}

    ctx := context.WithValue(context.Background(), auth.TestClaimsKey{}, claims)

    req := httptest.NewRequest("GET", "/me", nil).WithContext(ctx)

    rec := httptest.NewRecorder()

    h := NewHandler(nil, nil, nil, slog.Default())

    h.Me(rec, req)

    if rec.Code != 200 {

        t.Errorf("status: got %d", rec.Code)

    }

    var resp meResponse

    json.Unmarshal(rec.Body.Bytes(), &resp)

    if resp.UserID != "user-uuid" {

        t.Errorf("user_id: got %q", resp.UserID)

    }

}
```

Note on `TestClaimsKey`: Expose an exported key type from `shared/auth` for testing purposes only:

GO

```go
// shared/auth/auth.go — add alongside unexported claimsKey

// TestClaimsKey is exported ONLY for use in handler unit tests.

// Do not use in production code; use JWTMiddleware instead.

type TestClaimsKey = claimsKey
```

10.5 Integration Test — Full Round Trip

```go
// +build integration

// Run with: go test -tags integration -run TestIntegrationRoundTrip ./services/user-service/...

func TestIntegrationRoundTrip(t *testing.T) {

    // Requires running user_db at DATABASE_URL env var

    cfg, err := loadConfig()

    if err != nil {

        t.Skipf("config not available (set DATABASE_URL, JWT_SECRET): %v", err)

    }

    pool, err := NewDBPool(context.Background(), cfg.DatabaseURL, slog.Default())

    if err != nil {

        t.Fatalf("db: %v", err)

    }

    t.Cleanup(pool.Close)

    if err := runMigrations(context.Background(), pool, slog.Default()); err != nil {

        t.Fatalf("migrate: %v", err)

    }

    store := NewUserStore(pool)

    hasher := NewPasswordHasher(bcrypt.MinCost) // fast cost for tests

    issuer := NewTokenIssuer(cfg.JWTSecret, cfg.TokenTTL)

    h := NewHandler(store, hasher, issuer, slog.Default())

    mux := http.NewServeMux()

    mux.HandleFunc("POST /register", h.Register)

    mux.HandleFunc("POST /login", h.Login)

    mux.Handle("GET /me", auth.JWTMiddleware(cfg.JWTSecret)(http.HandlerFunc(h.Me)))

    srv := httptest.NewServer(mux)

    t.Cleanup(srv.Close)

    email := fmt.Sprintf("integration+%d@example.com", time.Now().UnixNano())

    password := "testpassword123"

    // Step 1: Register

    regBody, _ := json.Marshal(map[string]string{"email": email, "password": password})

    regResp, err := http.Post(srv.URL+"/register", "application/json", bytes.NewReader(regBody))

    if err != nil {

        t.Fatalf("register request: %v", err)

    }

    if regResp.StatusCode != 201 {
```

```
    body, _ := io.ReadAll(regResp.Body)
    t.Fatalf("register: got %d, body: %s", regResp.StatusCode, body)
}
var regResult registerResponse
json.NewDecoder(regResp.Body).Decode(&regResult)
if regResult.UserID == "" {
    t.Fatal("user_id is empty")
}
// Step 2: Login
loginBody, _ := json.Marshal(map[string]string{"email": email, "password": password})
loginResp, _ := http.Post(srv.URL+"/login", "application/json", bytes.NewReader(loginBody))
if loginResp.StatusCode != 200 {
    body, _ := io.ReadAll(loginResp.Body)
    t.Fatalf("login: got %d, body: %s", loginResp.StatusCode, body)
}
var loginResult loginResponse
json.NewDecoder(loginResp.Body).Decode(&loginResult)
if loginResult.Token == "" {
    t.Fatal("token is empty")
}
// Step 3: GET /me
meReq, _ := http.NewRequest("GET", srv.URL+"/me", nil)
meReq.Header.Set("Authorization", "Bearer "+loginResult.Token)
meResp, _ := http.DefaultClient.Do(meReq)
if meResp.StatusCode != 200 {
    body, _ := io.ReadAll(meResp.Body)
    t.Fatalf("me: got %d, body: %s", meResp.StatusCode, body)
}
var meResult meResponse
json.NewDecoder(meResp.Body).Decode(&meResult)
if meResult.UserID != regResult.UserID {
    t.Errorf("user_id mismatch: register=%q me=%q", regResult.UserID, meResult.UserID)
}
if meResult.Email != email {
    t.Errorf("email mismatch: got %q want %q", meResult.Email, email)
}
```

```
    // Step 4: Duplicate email returns 409
    regBody2, _ := json.Marshal(map[string]string{"email": email, "password": "differentpassword"})
    dupResp, _ := http.Post(srv.URL+"/register", "application/json", bytes.NewReader(regBody2))
    if dupResp.StatusCode != 409 {
        t.Errorf("duplicate email: got %d want 409", dupResp.StatusCode)
    }
    // Step 5: Wrong password returns 401 (same body as unknown email)
    badLoginBody, _ := json.Marshal(map[string]string{"email": email, "password": "wrongpassword"})
    badLoginResp, _ := http.Post(srv.URL+"/login", "application/json", bytes.NewReader(badLoginBody))
    if badLoginResp.StatusCode != 401 {
        t.Errorf("wrong password: got %d want 401", badLoginResp.StatusCode)
    }
    // Step 6: Unknown email returns 401 with identical body
    unknownLoginBody, _ := json.Marshal(map[string]string{"email": "nobody@example.com", "password": "whatever"})
    unknownLoginResp, _ := http.Post(srv.URL+"/login", "application/json", bytes.NewReader(unknownLoginBody))
    if unknownLoginResp.StatusCode != 401 {
        t.Errorf("unknown email: got %d want 401", unknownLoginResp.StatusCode)
    }
    // Verify response bodies are identical (anti-enumeration)
    badBody, _ := io.ReadAll(badLoginResp.Body)
    unknownBody, _ := io.ReadAll(unknownLoginResp.Body)
    if string(badBody) != string(unknownBody) {
        t.Errorf("401 bodies differ: wrong-password=%q unknown-email=%q", badBody, unknownBody)
    }
}
```

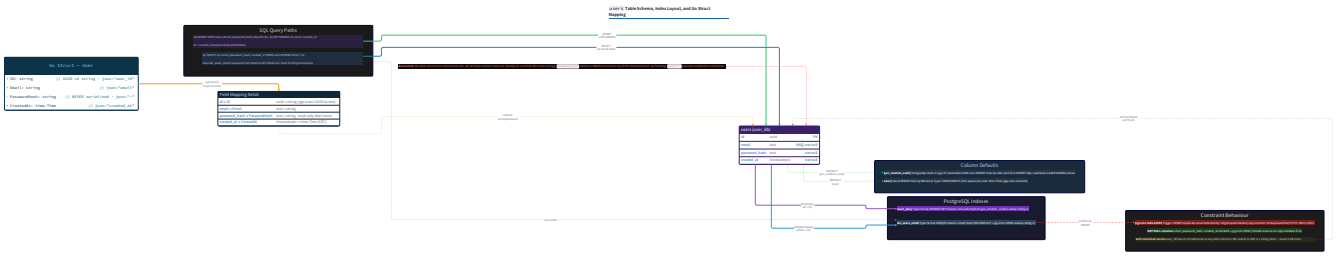
11. Performance Targets

Operation	Target	Measurement Command
`POST /register` (bcrypt cost=12)	< 400ms p99	`wrk -t4 -c4 -d30s -s register.lua http://localhost:8083/register` — check Latency 99th
`POST /login` (bcrypt compare cost=12 + DB)	< 500ms p99	`wrk -t4 -c4 -d30s -s login.lua http://localhost:8083/login`
`GET /me` (local JWT verify, no DB)	< 2ms p99	`wrk -t4 -c20 -d10s http://localhost:8083/me` with valid Bearer token in header via lua script
`GET /health`	< 10ms p99	`wrk -t1 -c10 -d10s http://localhost:8083/health`
`NewDBPool` startup ping	< 10s total (10s timeout)	Instrument with `time.Since` around `pool.Ping`; log as `Info`
bcrypt cost=12 single hash	< 400ms	`go test -bench=BenchmarkHash -benchtime=10s ./services/user-service/`

bcrypt benchmark:

```go
func BenchmarkBcryptHash(b *testing.B) {
    h := NewPasswordHasher(12)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        h.Hash("benchmarkpassword123")
    }
}
func BenchmarkBcryptVerify(b *testing.B) {
    h := NewPasswordHasher(12)
    hash, _ := h.Hash("benchmarkpassword123")
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        h.Verify("benchmarkpassword123", hash)
    }
}
```

Expected output: `BenchmarkBcryptHash-8 4 280000000 ns/op` (≈280ms per op at cost 12 on modern hardware).



```
Request latency breakdown:
POST /register:
  ┌──────────────────────────────────────────────────┐
  | Network + parse: ~1ms                            |
  | validateEmail + validatePassword: <1ms           |
  | bcrypt.GenerateFromPassword (cost=12): ~250-350ms | ← dominates
  | pgxpool.Exec INSERT: ~1-3ms                      |
  | JSON encode response: <1ms                       |
  | TOTAL: ~252-355ms (bcrypt is intentionally slow) |
  └──────────────────────────────────────────────────┘
POST /login:
  ┌──────────────────────────────────────────────────┐
  | Network + parse: ~1ms                            |
  | pgxpool.QueryRow SELECT: ~1-2ms                  |
  | bcrypt.CompareHashAndPassword (cost=12): ~250-350ms | ← dominates
  | jwt.Sign (HMAC-SHA256): <1ms                     |
  | JSON encode response: <1ms                       |
  | TOTAL: ~252-354ms                                |
  └──────────────────────────────────────────────────┘
GET /me:
  ┌──────────────────────────────────────────────────┐
  | Network + parse: ~0.5ms                          |
  | JWTMiddleware (jwt.Parse + HMAC-SHA256): <1ms    | ← pure computation
  | ClaimsFromContext: <0.1ms                        |
  | JSON encode response: <0.1ms                     |
  | TOTAL: ~1-2ms ✓ well within 2ms p99 target       |
  └──────────────────────────────────────────────────┘
```

12. `docker-compose.yml` Addition for JWT_SECRET

Add `JWT_SECRET` to the user-service environment block. The same value must be added to **all** services that consume `shared/auth` in later milestones:

```yaml
user-service:
  environment:
    DATABASE_URL: postgres://useruser:userpass@user_db:5432/userdb
    JWT_SECRET: "change-this-in-production-minimum-32-chars"
    PORT: "8080"
```

Add to `go.work` and `go.mod` replace directives for `shared/auth` in all services that will verify JWTs (url-service, analytics-service, notification-service, gateway) — even though those services don't use it until M3+, adding the dependency now prevents `go.work` synchronization issues

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

Module ID: `url-shortener-m3`

1. Module Charter

This module implements the core business logic of the URL shortener: accepting long URLs, generating collision-resistant 7-character base62 short codes, storing URL records in the `urls` table, serving redirects with a Redis read-through cache, and guaranteeing asynchronous event delivery to RabbitMQ via the outbox pattern. It extends the url-service binary established in M1 with the full HTTP surface (`POST /shorten` , `GET /:code` , `GET /urls` , `DELETE /urls/:code`) and a background outbox worker pool (1 coordinator + 3 workers). This module does **not** implement analytics — click events are published to RabbitMQ and consumed by analytics-service in M4. It does **not** send emails or touch the notification queue. It does **not** implement the API Gateway layer. It does **not** call any other service synchronously — the redirect hot path is entirely self-contained (DB + Redis). It does **not** run database migrations against any service's DB other than `url_db` . **Upstream dependencies:** `shared/events` (event struct definitions), `shared/auth` (JWT middleware + `ClaimsFromContext`), `shared/logger` (structured JSON logger), `pgxpool` (from M1 `NewDBPool`), `NewRedisClient` (from M1), `NewRabbitMQConn` (from M1), `url_db` PostgreSQL instance. **Downstream consumers:** analytics-service consumes `URLClickedEvent` from queue `analytics.clicks` . notification-service consumes `URLCreatedEvent` and `URLDeletedEvent` from queue `notifications.events` . The outbox pattern guarantees at-least-once delivery to both. **Invariants that must always hold:**

- Every `POST /shorten` and `DELETE /urls/:code` writes the URL row and its outbox event in a **single database transaction** — no partial writes ever occur.
- Redis is never authoritative: every cache miss falls through to PostgreSQL; Redis errors are non-fatal and logged at `Warn` level.
- Short codes use `crypto/rand` exclusively — `math/rand` is forbidden.
- Expired URLs (`expires_at < now()`) always return 410, never 404.
- Deactivated URLs (`is_active = false`) always return 410, never 404.
- URL ownership is verified on `DELETE` — mismatched `user_id` returns 403.
- Cache TTL is always `min(expires_at - now, 1h)` for URLs with expiry; `1h` for perpetual URLs; never unbounded.

2. File Structure

Create files in the numbered order below. Shared packages exist from M1/M2; only url-service files are new here.

```
url-shortener/
│
├── shared/                          ← unchanged from M1/M2
│   ├── events/events.go
│   ├── auth/auth.go + middleware.go
│   └── logger/logger.go
│
└── services/
    └── url-service/
        ├── (from M1, extended)
        │   ├── go.mod               ← add dependencies (see §6.3)
        │   ├── main.go              ← extend: wire all new components, start outbox
        │   ├── config.go            ← extend: add ShortURLBase, OutboxPollInterval
        │   ├── db.go                ← unchanged
        │   ├── redis.go             ← unchanged
        │   ├── rabbitmq.go          ← unchanged
        │   └── health.go            ← unchanged
        │
        ├── 1.  migration.sql        ← urls + outbox tables, all indexes
        ├── 2.  store.go             ← URLRepository interface + pgxURLStore
        ├── 3.  outbox_store.go      ← OutboxRepository interface + pgxOutboxStore
        ├── 4.  base62.go            ← Base62Encoder, alphabet constant
        ├── 5.  codegen.go           ← ShortCodeGenerator (crypto/rand, collision retry)
        ├── 6.  cache.go             ← RedisCache: Get/Set/Del with error isolation
        ├── 7.  publisher.go         ← RabbitMQPublisher interface + amqpPublisher
        ├── 8.  outbox.go            ← OutboxCoordinator + OutboxWorker
        ├── 9.  handler.go           ← Handler struct, Shorten/Redirect/ListURLs/DeleteURL
        ├── 10. validate.go          ← URL format validator (scheme + host check)
        ├── 11. errors.go            ← sentinel errors, writeError, writeJSON helpers
        ├── 12. url_test.go          ← unit tests (base62, codegen, handler, cache)
        └── 13. bench_test.go        ← redirect benchmark
```

3. Complete Data Model

3.1 PostgreSQL Schema (`migration.sql`)

```sql
-- ── urls table ──────────────────────────────────────────────────────────

CREATE TABLE IF NOT EXISTS urls (

    id           UUID         PRIMARY KEY DEFAULT gen_random_uuid(),

    short_code   VARCHAR(10)  UNIQUE 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

);

-- Redirect lookup: short_code → row. UNIQUE implies B-tree index;

-- explicit name for EXPLAIN ANALYZE verification.

CREATE UNIQUE INDEX IF NOT EXISTS idx_urls_short_code

    ON urls(short_code);

-- Paginated user URL list, newest-first.

-- Composite index supports: WHERE user_id = $1 AND id > $2 ORDER BY created_at DESC.

CREATE INDEX IF NOT EXISTS idx_urls_user_id_created

    ON urls(user_id, created_at DESC);

-- ── outbox table ────────────────────────────────────────────────────────

CREATE TABLE IF NOT EXISTS outbox (

    id           UUID         PRIMARY KEY DEFAULT gen_random_uuid(),

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

    payload      JSONB        NOT NULL,      -- full event struct serialized

    created_at   TIMESTAMPTZ  NOT NULL DEFAULT now(),

    published_at TIMESTAMPTZ  NULL           -- NULL = unpublished; set by worker on success

);

-- Outbox poller index: fetch only unpublished rows, oldest first.

-- Partial index omits published rows (WHERE published_at IS NULL) to stay small.

CREATE INDEX IF NOT EXISTS idx_outbox_unpublished

    ON outbox(created_at ASC)

    WHERE published_at IS NULL;
```

Column rationale:

Column	Type	Constraint	Rationale
`urls.id`	UUID	PK	Non-enumerable; used as cursor in pagination
`urls.short_code`	VARCHAR(10)	UNIQUE NOT NULL	7-char base62 + room for custom codes up to 10 chars
`urls.original_url`	TEXT	NOT NULL	Unbounded URL length; TEXT avoids artificial limit
`urls.user_id`	UUID	NOT NULL	Owner reference; checked on DELETE for 403
`urls.expires_at`	TIMESTAMPTZ	NULL	NULL = perpetual; compared with `now()` on redirect
`urls.is_active`	BOOLEAN	DEFAULT true	Soft delete; false → 410 Gone
`outbox.event_type`	TEXT	NOT NULL	RabbitMQ routing key; must match `events.EventType*` constants
`outbox.payload`	JSONB	NOT NULL	Full event JSON; worker deserializes to verify, then publishes as AMQP body
`outbox.published_at`	TIMESTAMPTZ	NULL	Poller criterion: `WHERE published_at IS NULL`; set atomically by worker

3.2 Go Structs

```go
// store.go

package urlservice

import "time"

// URLRecord is the domain object mapped from the urls table.

// Never returned directly in HTTP responses — projection structs handle serialization.

type URLRecord struct {

    ID          string     // UUID string

    ShortCode   string

    OriginalURL string

    UserID      string     // UUID string

    CreatedAt   time.Time

    ExpiresAt   *time.Time // nil if no expiry

    IsActive    bool

}

// OutboxRecord is the domain object mapped from the outbox table.

type OutboxRecord struct {

    ID          string     // UUID string

    EventType   string

    Payload     []byte     // raw JSONB bytes; sent as AMQP message body

    CreatedAt   time.Time

    PublishedAt *time.Time // nil if unpublished

}
```

```go
// cache.go

// CachedURL is the Redis-stored projection for the redirect path.

// Small enough to fit in a single Redis string value.

// Storing is_active avoids returning 301 for a deactivated URL cached before deletion.

type CachedURL struct {

    OriginalURL string     `json:"original_url"`

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

    IsActive    bool       `json:"is_active"`

}
```

```go
// handler.go — HTTP request/response types

type shortenRequest struct {

    URL        string  `json:"url"`

    CustomCode *string `json:"custom_code,omitempty"` // nil = auto-generate

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

}

type shortenResponse struct {

    ShortCode   string  `json:"short_code"`

    ShortURL    string  `json:"short_url"`    // cfg.ShortURLBase + "/" + short_code

    OriginalURL string  `json:"original_url"`

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

}

type urlListItem struct {

    ShortCode   string  `json:"short_code"`

    OriginalURL string  `json:"original_url"`

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

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

    IsActive    bool    `json:"is_active"`

}

type urlListResponse struct {

    URLs       []urlListItem `json:"urls"`

    NextCursor *string       `json:"next_cursor"` // null if no more pages

}
```

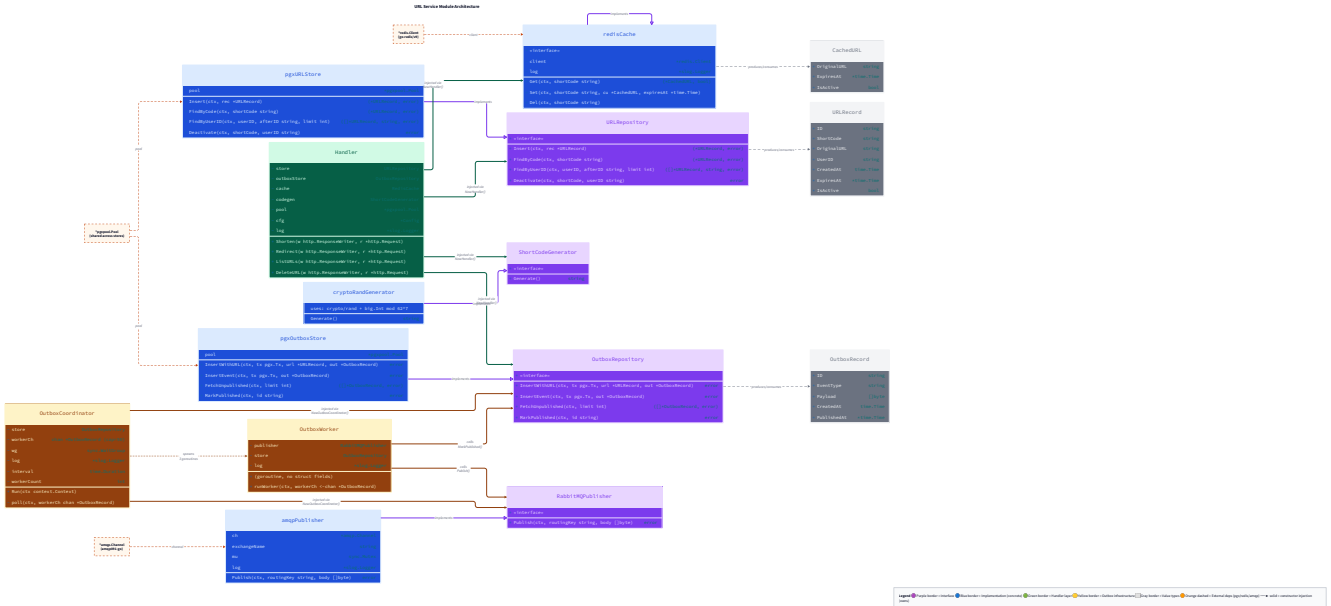
```go
// config.go (extended from M1)

type Config struct {

    DatabaseURL         string

    RedisURL            string

    RabbitMQURL         string

    Port                string

    ServiceName         string

    JWTSecret           string        // required; added in M2 for other services, now url-service needs it

    ShortURLBase        string        // e.g. "http://localhost:8080" or "https://sho.rt"

    OutboxPollInterval  time.Duration // default 2s; configurable via OUTBOX_POLL_INTERVAL_MS

    OutboxWorkerCount   int           // default 3; fixed per spec but readable from config

}
```

3.3 Repository Interfaces

```go
// store.go — URLRepository

type URLRepository interface {

    // Insert creates a new URL record. Returns ErrShortCodeConflict if short_code already exists.

    Insert(ctx context.Context, rec *URLRecord) (*URLRecord, error)

    // FindByCode retrieves a URL by short_code.

    // Returns ErrURLNotFound if no row exists (active or not).

    FindByCode(ctx context.Context, shortCode string) (*URLRecord, error)

    // FindByUserID returns paginated URLs for a user, ordered by created_at DESC.

    // afterID is the cursor: if non-empty, returns rows with id < afterID in created_at DESC order.

    // limit is the page size (max 50).

    // Returns slice (possibly empty) and the next cursor UUID string (empty if no more pages).

    FindByUserID(ctx context.Context, userID, afterID string, limit int) ([]*URLRecord, string, error)

    // Deactivate sets is_active=false for the given short_code owned by userID.

    // Returns ErrURLNotFound if short_code doesn't exist.

    // Returns ErrNotOwner if the row exists but user_id doesn't match.

    Deactivate(ctx context.Context, shortCode, userID string) error

}
```

```go
// outbox_store.go — OutboxRepository

type OutboxRepository interface {

    // InsertWithURL inserts a URL record and an outbox row atomically within tx.

    // Called only by the shorten handler via a transaction helper.

    // tx is a *pgx.Tx passed from the handler's transaction scope.

    InsertWithURL(ctx context.Context, tx pgx.Tx, urlRec *URLRecord, outboxRec *OutboxRecord) error

    // InsertEvent inserts a single outbox row (used for URLDeletedEvent, URLClickedEvent).

    // Also called within an existing transaction.

    InsertEvent(ctx context.Context, tx pgx.Tx, outboxRec *OutboxRecord) error

    // FetchUnpublished returns up to limit outbox rows with published_at IS NULL,

    // ordered by created_at ASC. Uses SELECT ... FOR UPDATE SKIP LOCKED to prevent

    // multiple coordinator instances from picking the same row (safe for future scaling).

    FetchUnpublished(ctx context.Context, limit int) ([]*OutboxRecord, error)

    // MarkPublished sets published_at = now() for the given outbox row ID.

    MarkPublished(ctx context.Context, id string) error

}
```

```go
// publisher.go — RabbitMQPublisher

type RabbitMQPublisher interface {

    // Publish sends a message to the exchange with the given routing key.

    // body is the raw JSON bytes (outbox.payload column value).

    // Returns error if the AMQP channel is closed or publish fails.

    Publish(ctx context.Context, routingKey string, body []byte) error

}
```



4. Interface Contracts

4.1 `URLRepository` — `pgxURLStore` Implementation

```go
// store.go

type pgxURLStore struct {

    pool *pgxpool.Pool

}

func NewURLStore(pool *pgxpool.Pool) URLRepository {

    return &pgxURLStore{pool: pool}

}
```

`Insert(ctx context.Context, rec *URLRecord) (*URLRecord, error)` Called within a transaction scope from the shorten handler. The `rec.ID` field is left empty on input — the DB generates it via `DEFAULT gen_random_uuid()`.

```
SQL:
  INSERT INTO urls (short_code, original_url, user_id, expires_at)
  VALUES ($1, $2, $3, $4)
  RETURNING id, short_code, original_url, user_id, created_at, expires_at, is_active
Parameters: rec.ShortCode, rec.OriginalURL, rec.UserID, rec.ExpiresAt (nil OK)
On success:  return populated *URLRecord (all fields from RETURNING clause)
On pgError 23505 (unique_violation on short_code): return nil, ErrShortCodeConflict
On other DB error: return nil, fmt.Errorf("insert url: %w", err)
```

`FindByCode(ctx context.Context, shortCode string) (*URLRecord, error)`

```
SQL:
  SELECT id, short_code, original_url, user_id, created_at, expires_at, is_active
  FROM urls
  WHERE short_code = $1
On pgx.ErrNoRows: return nil, ErrURLNotFound
On success: return *URLRecord with all fields
On other DB error: return nil, fmt.Errorf("find url by code: %w", err)
Note: This returns the row regardless of is_active and expires_at.
      The caller (handler) decides 404 vs 410 semantics.
```

`FindByUserID(ctx context.Context, userID, afterID string, limit int) ([]*URLRecord, string, error)` Implements cursor-based pagination. The cursor is the `id` (UUID) of the last item returned in the previous page. Because `created_at` is the sort column and UUIDs are unique, the cursor resolves to a created_at boundary using a subquery.

```
When afterID == "" (first page):
  SQL:
    SELECT id, short_code, original_url, user_id, created_at, expires_at, is_active
    FROM urls
    WHERE user_id = $1
    ORDER BY created_at DESC, id DESC
    LIMIT $2
When afterID != "" (subsequent pages):
  SQL:
    SELECT id, short_code, original_url, user_id, created_at, expires_at, is_active
    FROM urls
    WHERE user_id = $1
      AND (created_at, id) < (
          SELECT created_at, id FROM urls WHERE id = $2
      )
    ORDER BY created_at DESC, id DESC
    LIMIT $3
  Parameters: userID, afterID (cursor UUID), limit+1
After query: fetch limit+1 rows to determine if a next page exists.
  - If len(rows) == limit+1: nextCursor = rows[limit].ID; return rows[:limit]
  - If len(rows) <= limit:   nextCursor = ""; return rows as-is
  - Uses idx_urls_user_id_created; verify with EXPLAIN ANALYZE (checkpoint §9)
On success: return ([]*URLRecord, nextCursorString, nil)
  nextCursorString is "" if no more pages (JSON null in response)
On DB error: return nil, "", fmt.Errorf("find urls by user: %w", err)
```

`Deactivate(ctx context.Context, shortCode, userID string) error`

```
SQL:
  UPDATE urls
  SET is_active = false
  WHERE short_code = $1
  RETURNING user_id
On pgx.ErrNoRows from QueryRow: return ErrURLNotFound
On success scan: compare returned user_id with param userID
  - If different: return ErrNotOwner
  - If same: return nil
On other DB error: return fmt.Errorf("deactivate url: %w", err)
Note: Uses RETURNING user_id to do ownership check and update in one round-trip.
```

4.2 `OutboxRepository` — `pgxOutboxStore` Implementation

```go
// outbox_store.go

type pgxOutboxStore struct {

    pool *pgxpool.Pool

}

func NewOutboxStore(pool *pgxpool.Pool) OutboxRepository {

    return &pgxOutboxStore{pool: pool}

}
```

`InsertWithURL(ctx context.Context, tx pgx.Tx, urlRec *URLRecord, outboxRec *OutboxRecord) error` Both inserts execute on the same `pgx.Tx`. Called from the handler's transaction scope.

```go
// Implementation pattern — caller provides the tx:

func (s *pgxOutboxStore) InsertWithURL(ctx context.Context, tx pgx.Tx, urlRec *URLRecord, outboxRec *OutboxRecord) error {

    // Insert URL row

    row := tx.QueryRow(ctx,

        `INSERT INTO urls (short_code, original_url, user_id, expires_at)

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

         RETURNING id, short_code, original_url, user_id, created_at, expires_at, is_active`,

        urlRec.ShortCode, urlRec.OriginalURL, urlRec.UserID, urlRec.ExpiresAt,

    )

    if err := row.Scan(&urlRec.ID, &urlRec.ShortCode, &urlRec.OriginalURL,

        &urlRec.UserID, &urlRec.CreatedAt, &urlRec.ExpiresAt, &urlRec.IsActive); err != nil {

        if isPgUniqueViolation(err) {

            return ErrShortCodeConflict

        }

        return fmt.Errorf("insert url in tx: %w", err)

    }

    // Insert outbox row in same transaction

    return s.InsertEvent(ctx, tx, outboxRec)

}
```

`InsertEvent(ctx context.Context, tx pgx.Tx, outboxRec *OutboxRecord) error`

```
SQL (on tx):
  INSERT INTO outbox (event_type, payload)
  VALUES ($1, $2)
  RETURNING id, created_at
Parameters: outboxRec.EventType, outboxRec.Payload ([]byte)
On success: populate outboxRec.ID and outboxRec.CreatedAt from RETURNING clause
On error: return fmt.Errorf("insert outbox event: %w", err)
```

`FetchUnpublished(ctx context.Context, limit int) ([]*OutboxRecord, error)`

```
SQL:
  SELECT id, event_type, payload, created_at
  FROM outbox
  WHERE published_at IS NULL
  ORDER BY created_at ASC
  LIMIT $1
  FOR UPDATE SKIP LOCKED
Parameters: limit (always 50 per coordinator call)
Returns: slice of *OutboxRecord with ID, EventType, Payload, CreatedAt
On empty result: return empty slice, nil (not an error)
On DB error: return nil, fmt.Errorf("fetch unpublished outbox: %w", err)
Note: FOR UPDATE SKIP LOCKED is safe here even with a single coordinator instance.
      It prevents double-processing if a second coordinator is ever started.
      The coordinator runs this in a pgxpool.BeginTx context — see §5.5.
```

`MarkPublished(ctx context.Context, id string) error`

```
SQL:
  UPDATE outbox SET published_at = now() WHERE id = $1
On 0 rows affected: not an error (idempotent; already marked by another instance)
On DB error: return fmt.Errorf("mark outbox published: %w", err)
```

4.3 `RedisCache`

```go
// cache.go

type RedisCache struct {

    client *redis.Client

    log    *slog.Logger

}

func NewRedisCache(client *redis.Client, log *slog.Logger) *RedisCache

const cacheKeyPrefix = "url:"  // key format: "url:{short_code}"

const defaultCacheTTL = time.Hour
```

`Get(ctx context.Context, shortCode string) (*CachedURL, bool)`

```
key := "url:" + shortCode
val, err := c.client.Get(ctx, key).Bytes()
  - On redis.Nil (key missing): return nil, false   // cache miss — normal
  - On other error: c.log.Warn("redis get failed", "key", key, "error", err)
                    return nil, false               // treat as miss, fall to DB
  - On success: unmarshal val into *CachedURL
      - If unmarshal fails: c.log.Warn("redis unmarshal failed", "key", key, "error", err)
                            return nil, false        // treat as miss
      - If success: return &cachedURL, true          // cache hit
```

`Set(ctx context.Context, shortCode string, cu *CachedURL, expiresAt *time.Time)`

```
key := "url:" + shortCode
ttl := computeTTL(expiresAt)           // see algorithm below
body, err := json.Marshal(cu)
  - If marshal fails: c.log.Warn("redis marshal failed", "key", key, "error", err); return
val := c.client.Set(ctx, key, body, ttl)
  - If val.Err() != nil: c.log.Warn("redis set failed", "key", key, "error", val.Err())
  - Either way: return (no error propagated to caller)
computeTTL(expiresAt *time.Time) time.Duration:
  if expiresAt == nil:
      return defaultCacheTTL  // 1 hour
  remaining := time.Until(*expiresAt)
  if remaining <= 0:
      return 1 * time.Second  // will expire immediately; don't cache at all in practice
                              // (handler returns 410 before reaching Set; but guard here)
  if remaining < defaultCacheTTL:
      return remaining
  return defaultCacheTTL
```

`Del(ctx context.Context, shortCode string)`

```
key := "url:" + shortCode
err := c.client.Del(ctx, key).Err()
  - If err != nil: c.log.Warn("redis del failed", "key", key, "error", err)
  - Either way: return (best-effort; caller does not see error)
```

4.4 `Base62Encoder` and `ShortCodeGenerator`

GO

```go
// base62.go

const base62Alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

const shortCodeLength = 7

// Encode converts a big.Int to a base62 string of exactly shortCodeLength characters.

// Pads with '0' (alphabet[0]) on the left if the number requires fewer digits.

func Encode(n *big.Int) string

// Decode converts a base62 string back to a big.Int (used in tests for round-trip verification).

func Decode(s string) (*big.Int, error)
```

```go
// codegen.go

type ShortCodeGenerator struct{}

func NewShortCodeGenerator() *ShortCodeGenerator

// Generate creates a cryptographically random 7-character base62 short code.

// Uses crypto/rand to fill 5 bytes (40 bits → max value 2^40 = 1,099,511,627,776)

// then takes modulo 62^7 = 3,521,614,606,208 to map into base62 space.

//

// Probability analysis:

//   62^7 = 3.5 trillion codes.

//   After 1 million URLs: collision probability ≈ 1.4 × 10^-7 per attempt.

//   5-retry budget makes the probability of all 5 colliding negligible.

//

// Returns: 7-character string from base62Alphabet

// Never returns an error (crypto/rand failure panics — system entropy failure is unrecoverable)

func (g *ShortCodeGenerator) Generate() string
```

4.5 `RabbitMQPublisher` — `amqpPublisher` Implementation

```go
// publisher.go

type amqpPublisher struct {

    ch           *amqp.Channel

    exchangeName string

    log          *slog.Logger

}

func NewRabbitMQPublisher(ch *amqp.Channel, log *slog.Logger) RabbitMQPublisher {

    return &amqpPublisher{ch: ch, exchangeName: "url-shortener", log: log}

}
```

`Publish(ctx context.Context, routingKey string, body []byte) error`

```go
err := p.ch.PublishWithContext(ctx,
    p.exchangeName,  // exchange
    routingKey,      // routing key = event_type constant
    false,           // mandatory: false — drop if no queue bound (shouldn't happen)
    false,           // immediate: false — AMQP 0-9-1 doesn't support immediate
    amqp.Publishing{
        ContentType:  "application/json",
        DeliveryMode: amqp.Persistent, // survive broker restart
        Body:         body,
    },
)
if err != nil:
    return fmt.Errorf("amqp publish %s: %w", routingKey, err)
return nil
```

5. Algorithm Specification

5.1 `ShortCodeGenerator.Generate()` — Base62 Random Code

```
Input:  none
Output: 7-character string from base62Alphabet [0-9A-Za-z]
Step 1: Allocate 8-byte buffer
  buf := make([]byte, 8)
Step 2: Fill with crypto/rand
  if _, err := rand.Read(buf); err != nil {
      panic(fmt.Sprintf("crypto/rand read failed: %v", err))
  }
  // Panicking here is correct: crypto/rand failure means the OS entropy pool
  // is exhausted — no safe randomness is possible. The process must not continue.
Step 3: Convert to big.Int and reduce to base62 space
  n := new(big.Int).SetBytes(buf)
  // 62^7 = 3,521,614,606,208
  maxCode := new(big.Int).Exp(big.NewInt(62), big.NewInt(7), nil)
  n.Mod(n, maxCode)
Step 4: Encode to base62 string of length 7
  return Encode(n)  // left-pads with '0' if n encodes to fewer than 7 chars
Encode(n *big.Int) string:
  result := make([]byte, shortCodeLength)
  // Fill right-to-left:
  sixty2 := big.NewInt(62)
  mod := new(big.Int)
  for i := shortCodeLength - 1; i >= 0; i-- {
      n.DivMod(n, sixty2, mod)
      result[i] = base62Alphabet[mod.Int64()]
  }
  return string(result)
```

5.2 `POST /shorten` Handler — Full Algorithm

```
Input:  JWT-authenticated request; body: shortenRequest
Output: 201 {short_code, short_url, original_url, expires_at} | 400 | 409 | 422 | 503
Step 1: Extract JWT claims
  claims, ok := auth.ClaimsFromContext(r.Context())
  if !ok: writeError(w, 401, "unauthorized", ""); return
Step 2: Decode and validate request body
  Limit body: r.Body = http.MaxBytesReader(w, r.Body, 1<<20)
  var req shortenRequest
  if err := json.NewDecoder(r.Body).Decode(&req); err != nil:
      writeError(w, 400, "invalid request body", ""); return
Step 3: Validate URL
  if err := validateURL(req.URL); err != nil:
      writeError(w, 422, err.Error(), "url"); return
Step 4: Parse optional expires_at
  var expiresAt *time.Time
  if req.ExpiresAt != nil:
      t, err := time.Parse(time.RFC3339, *req.ExpiresAt)
      if err != nil:
          writeError(w, 422, "expires_at must be RFC3339 format", "expires_at"); return
      if t.Before(time.Now()):
          writeError(w, 422, "expires_at must be in the future", "expires_at"); return
      expiresAt = &t
Step 5: Determine short code
  var shortCode string
  if req.CustomCode != nil:
      shortCode = *req.CustomCode
      if len(shortCode) == 0 || len(shortCode) > 10:
          writeError(w, 422, "custom_code must be 1-10 characters", "custom_code"); return
      // Custom code: attempt insert directly; DB unique constraint handles conflict
      // (no retry loop for custom codes)
  else:
      shortCode = ""  // will be set in retry loop below
Step 6: Insert URL + outbox in single transaction (with collision retry)
  const maxRetries = 5
  var urlRec *URLRecord
  var insertErr error
  for attempt := 0; attempt < maxRetries; attempt++:
      if shortCode == "" {  // auto-generate
          shortCode = h.codegen.Generate()
      }
      urlRec = &URLRecord{
          ShortCode:   shortCode,
          OriginalURL: req.URL,
          UserID:      claims.Sub,
          ExpiresAt:   expiresAt,
      }
      // Build outbox record (payload serialized before tx opens)
      event := &events.URLCreatedEvent{
          BaseEvent: events.BaseEvent{
              EventType:     events.EventTypeURLCreated,
              OccurredAt:    time.Now().UTC(),
              CorrelationID: correlationIDFromContext(r.Context()),
              EventID:       newUUID(),
          },
          ShortCode:   shortCode,
          OriginalURL: req.URL,
          UserID:      claims.Sub,
          UserEmail:   claims.Email,
          ExpiresAt:   expiresAt,
      }
      payload, err := json.Marshal(event)
      if err != nil:
          log.Error("marshal URLCreatedEvent", "error", err)
          writeError(w, 500, "internal server error", ""); return
      outboxRec := &OutboxRecord{
          EventType: events.EventTypeURLCreated,
          Payload:   payload,
      }
      // Open transaction
      tx, err := h.pool.Begin(r.Context())
```

```
      if err != nil:
          log.Error("begin tx", "error", err)
          writeError(w, 500, "internal server error", ""); return
      insertErr = h.outboxStore.InsertWithURL(r.Context(), tx, urlRec, outboxRec)
      if insertErr == nil:
          if err := tx.Commit(r.Context()); err != nil:
              tx.Rollback(r.Context())
              log.Error("commit tx", "error", err)
              writeError(w, 500, "internal server error", ""); return
          break  // success
      tx.Rollback(r.Context())
      if errors.Is(insertErr, ErrShortCodeConflict):
          if req.CustomCode != nil:
              // Custom code conflict is permanent — do not retry
              writeError(w, 409, "short code already taken", "custom_code"); return
          // Auto-generated: reset and retry with new code
          shortCode = ""
          continue
      // Other DB error
      log.Error("insert url+outbox", "error", insertErr, "attempt", attempt)
      writeError(w, 500, "internal server error", ""); return
  if insertErr != nil:
      // Exhausted maxRetries on collision
      log.Error("short code collision exhausted", "attempts", maxRetries)
      writeError(w, 503, "service temporarily unavailable, try again", ""); return
Step 7: Write response
  var expiresAtStr *string
  if urlRec.ExpiresAt != nil:
      s := urlRec.ExpiresAt.Format(time.RFC3339)
      expiresAtStr = &s
  writeJSON(w, 201, shortenResponse{
      ShortCode:   urlRec.ShortCode,
      ShortURL:    h.cfg.ShortURLBase + "/" + urlRec.ShortCode,
      OriginalURL: urlRec.OriginalURL,
      ExpiresAt:   expiresAtStr,
  })
```

5.3 `GET /:code` — Read-Through Cache Redirect

```
Input:  path parameter: code string (from URL path)
Output: 301 Location | 404 | 410
Step 1: Extract short_code from path
  code := r.PathValue("code")   // Go 1.22+ net/http pattern: "GET /{code}"
  if code == "": writeError(w, 404, "not found", ""); return
Step 2: Try Redis cache
  cacheCtx, cancel := context.WithTimeout(r.Context(), 50*time.Millisecond)
  defer cancel()
  cached, hit := h.cache.Get(cacheCtx, code)
  if hit:
      if !cached.IsActive:
          writeError(w, 410, "url has been deactivated", ""); return
      if cached.ExpiresAt != nil && cached.ExpiresAt.Before(time.Now()):
          writeError(w, 410, "url has expired", ""); return
      http.Redirect(w, r, cached.OriginalURL, http.StatusMovedPermanently)  // 301
      return
Step 3: Cache miss — query PostgreSQL
  urlRec, err := h.urlStore.FindByCode(r.Context(), code)
  if errors.Is(err, ErrURLNotFound):
      writeError(w, 404, "short url not found", ""); return
  if err != nil:
      h.log.Error("find url by code", "code", code, "error", err)
      writeError(w, 500, "internal server error", ""); return
Step 4: Evaluate active + expiry state
  if !urlRec.IsActive:
      // Do NOT cache deactivated URLs
      writeError(w, 410, "url has been deactivated", ""); return
  if urlRec.ExpiresAt != nil && urlRec.ExpiresAt.Before(time.Now()):
      // Do NOT cache expired URLs (TTL would be negative)
      writeError(w, 410, "url has expired", ""); return
Step 5: Populate cache (non-blocking, errors swallowed inside Set)
  setCtx, cancel := context.WithTimeout(r.Context(), 100*time.Millisecond)
  defer cancel()
  h.cache.Set(setCtx, code, &CachedURL{
      OriginalURL: urlRec.OriginalURL,
      ExpiresAt:   urlRec.ExpiresAt,
      IsActive:    urlRec.IsActive,
  }, urlRec.ExpiresAt)
Step 6: Write click event to outbox (same transaction as redirect is impractical —
        redirect is a read, not a write. Click event MUST be in its own transaction
        to be atomic. If service crashes between redirect and click insert, event
        is lost — this is acceptable per spec: the redirect happened; the click
        event may be delayed or lost. The outbox pattern provides at-least-once
        for the write path, not for reads.)
  clickEvent := &events.URLClickedEvent{
      BaseEvent: events.BaseEvent{
          EventType:     events.EventTypeURLClicked,
          OccurredAt:    time.Now().UTC(),
          CorrelationID: correlationIDFromContext(r.Context()),
          EventID:       newUUID(),
      },
      ShortCode: code,
      IPHash:    hashIP(r.RemoteAddr),  // SHA-256(IP + salt), see §5.6
      UserAgent: r.Header.Get("User-Agent"),
      Referer:   r.Header.Get("Referer"),
      ClickedAt: time.Now().UTC(),
  }
  payload, err := json.Marshal(clickEvent)
  if err == nil:
      tx, err := h.pool.Begin(r.Context())
      if err == nil:
          outboxRec := &OutboxRecord{EventType: events.EventTypeURLClicked, Payload: payload}
          if err := h.outboxStore.InsertEvent(r.Context(), tx, outboxRec); err != nil:
              tx.Rollback(r.Context())
              h.log.Warn("insert click event outbox", "code", code, "error", err)
          else:
              if err := tx.Commit(r.Context()); err != nil:
                  h.log.Warn("commit click event tx", "error", err)
      else:
          h.log.Warn("begin click tx", "error", err)
```

```
  // Click event failure is always non-fatal — redirect proceeds regardless
Step 7: Redirect
  http.Redirect(w, r, urlRec.OriginalURL, http.StatusMovedPermanently)  // 301
```

Critical design note on click atomicity: The M3 spec states the outbox write must be atomic. For `URLClickedEvent`, this is written in its own transaction immediately after the redirect decision is made. If the process crashes after writing the redirect response but before committing the outbox row, the click is lost — this is the acceptable trade-off documented in the spec. The alternative (writing the event before sending the redirect) risks a phantom event if the redirect fails. The outbox guarantees at-least-once for events that do commit; it cannot guarantee events for HTTP responses already sent.

5.4 `DELETE /urls/:code` Handler

```
Input:  JWT-authenticated; path param: code
Output: 204 | 403 | 404 | 500
Step 1: Extract claims
  claims, ok := auth.ClaimsFromContext(r.Context())
  if !ok: writeError(w, 401, "unauthorized", ""); return
Step 2: Deactivate in DB (ownership check inside store)
  tx, err := h.pool.Begin(r.Context())
  if err != nil: writeError(w, 500, "internal server error", ""); return
  // Must read the URL before deactivating to build the event payload
  urlRec, err := h.urlStore.FindByCode(r.Context(), r.PathValue("code"))
  if errors.Is(err, ErrURLNotFound): tx.Rollback(r.Context()); writeError(w, 404, ...); return
  if err != nil: tx.Rollback(r.Context()); writeError(w, 500, ...); return
  if urlRec.UserID != claims.Sub: tx.Rollback(r.Context()); writeError(w, 403, "forbidden", ""); return
  // Deactivate (UPDATE within same tx)
  // Execute: UPDATE urls SET is_active = false WHERE short_code = $1 AND user_id = $2
  _, err = tx.Exec(r.Context(),
      `UPDATE urls SET is_active = false WHERE short_code = $1 AND user_id = $2`,
      r.PathValue("code"), claims.Sub)
  if err != nil: tx.Rollback(r.Context()); writeError(w, 500, ...); return
Step 3: Insert URLDeletedEvent into outbox (same tx)
  event := &events.URLDeletedEvent{
      BaseEvent: events.BaseEvent{
          EventType:     events.EventTypeURLDeleted,
          OccurredAt:    time.Now().UTC(),
          CorrelationID: correlationIDFromContext(r.Context()),
          EventID:       newUUID(),
      },
      ShortCode: r.PathValue("code"),
      UserID:    claims.Sub,
      UserEmail: claims.Email,
  }
  payload, _ := json.Marshal(event)
  outboxRec := &OutboxRecord{EventType: events.EventTypeURLDeleted, Payload: payload}
  if err := h.outboxStore.InsertEvent(r.Context(), tx, outboxRec); err != nil:
      tx.Rollback(r.Context())
      h.log.Error("insert delete event outbox", "error", err)
      writeError(w, 500, "internal server error", ""); return
Step 4: Commit
  if err := tx.Commit(r.Context()); err != nil:
      h.log.Error("commit delete tx", "error", err)
      writeError(w, 500, "internal server error", ""); return
Step 5: Invalidate Redis cache (best-effort, after commit)
  delCtx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
  defer cancel()
  h.cache.Del(delCtx, r.PathValue("code"))
  // h.cache.Del always returns void; errors are logged inside Del
Step 6: Write 204 No Content
  w.WriteHeader(http.StatusNoContent)
```

5.5 Outbox Coordinator + Worker Pool

```
Outbox architecture:
  main.go
    | go outboxCoordinator.Run(ctx)
    |
  OutboxCoordinator
    | every 2s: FetchUnpublished(limit=50)
    | for each row:
    |   workerCh <- row                    ← buffered channel, cap=50
    |
  OutboxWorker × 3 (goroutines)
    | for row := range workerCh:
    |   publisher.Publish(row.EventType, row.Payload)
    |   outboxStore.MarkPublished(row.ID)
    |
  Graceful shutdown:
    ctx cancelled → coordinator stops polling → workerCh drained → workers exit
```

```go
// outbox.go

type OutboxCoordinator struct {

    store     OutboxRepository

    workerCh  chan *OutboxRecord

    log       *slog.Logger

    interval  time.Duration

    workerCount int

}

func NewOutboxCoordinator(

    store OutboxRepository,

    publisher RabbitMQPublisher,

    log *slog.Logger,

    interval time.Duration,

    workerCount int,

) *OutboxCoordinator

// Run starts the coordinator and worker goroutines.

// Blocks until ctx is cancelled.

// Called with `go coordinator.Run(ctx)` in main.go.

func (c *OutboxCoordinator) Run(ctx context.Context)
```

`OutboxCoordinator.Run(ctx context.Context)` Algorithm:

```
Step 1: Create buffered worker channel
  workerCh := make(chan *OutboxRecord, 50)
Step 2: Start worker goroutines
  var wg sync.WaitGroup
  for i := 0; i < c.workerCount; i++:
      wg.Add(1)
      go func():
          defer wg.Done()
          c.runWorker(ctx, workerCh)
      ()
Step 3: Poll loop
  ticker := time.NewTicker(c.interval)
  defer ticker.Stop()
  for:
      select:
      case <-ctx.Done():
          close(workerCh)   // signal workers to drain and exit
          wg.Wait()         // wait for all workers to finish in-flight publishes
          c.log.Info("outbox coordinator stopped")
          return
      case <-ticker.C:
          c.poll(ctx, workerCh)
poll(ctx context.Context, workerCh chan *OutboxRecord):
  rows, err := c.store.FetchUnpublished(ctx, 50)
  if err != nil:
      c.log.Warn("outbox fetch failed", "error", err); return
  for _, row := range rows:
      select:
      case workerCh <- row:  // send to available worker
      case <-ctx.Done():
          return              // coordinator is shutting down
runWorker(ctx context.Context, workerCh <-chan *OutboxRecord):
  for row := range workerCh:   // exits when channel is closed
      publishCtx, cancel := context.WithTimeout(ctx, 5*time.Second)
      err := c.publisher.Publish(publishCtx, row.EventType, row.Payload)
      cancel()
      if err != nil:
          c.log.Warn("outbox publish failed",
              "event_type", row.EventType,
              "outbox_id", row.ID,
              "error", err)
          // Leave published_at NULL — coordinator will re-fetch on next poll
          continue
      if err := c.store.MarkPublished(context.Background(), row.ID); err != nil:
          c.log.Warn("outbox mark published failed",
              "outbox_id", row.ID,
              "error", err)
          // RabbitMQ got the message but DB update failed.
          // Next poll will re-fetch this row and re-publish.
          // Consumer must handle duplicate events (analytics-service does this in M4).
```

Why `FOR UPDATE SKIP LOCKED` matters: If `MarkPublished` fails after a successful publish, the row remains in `published_at IS NULL` state. The next poll cycle picks it up and publishes again. The consumer (analytics-service) deduplicates by `EventID`. This is the at-least-once guarantee.

5.6 IP Hashing (`hashIP`)

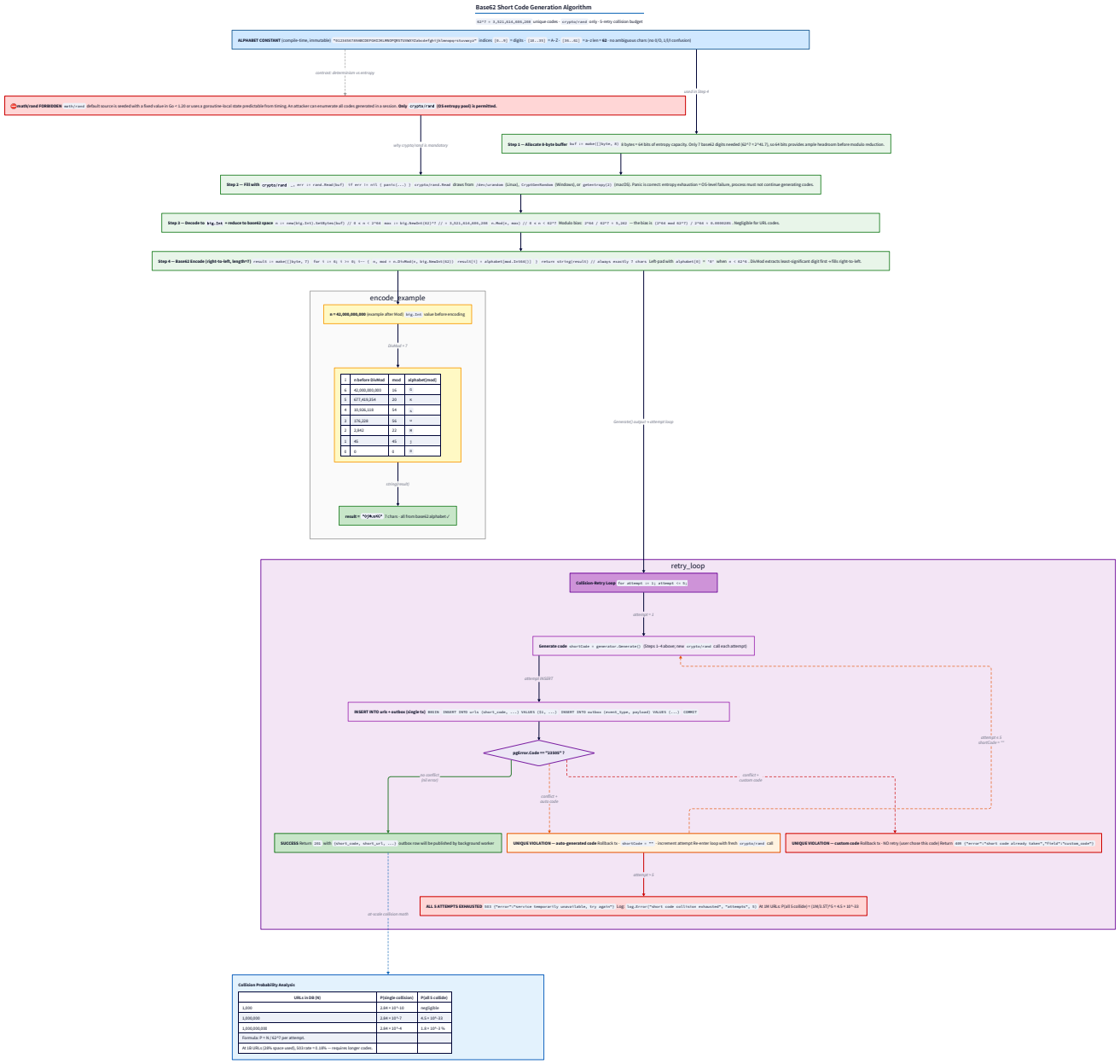
```go
// handler.go helper

import (

    "crypto/sha256"

    "fmt"

    "net"

    "os"

)

var ipHashSalt = os.Getenv("IP_HASH_SALT") // set in docker-compose env; empty string is acceptable fallback

func hashIP(remoteAddr string) string {

    ip, _, err := net.SplitHostPort(remoteAddr)

    if err != nil {

        ip = remoteAddr // fallback: use raw if port parsing fails

    }

    h := sha256.Sum256([]byte(ip + ipHashSalt))

    return fmt.Sprintf("%x", h)

}
```

5.7 Startup Migration

```GO
// main.go — after NewDBPool succeeds

const urlServiceSchema = `

CREATE TABLE IF NOT EXISTS urls (

    id           UUID         PRIMARY KEY DEFAULT gen_random_uuid(),

    short_code   VARCHAR(10)  UNIQUE NOT NULL,

    original_url TEXT         NOT NULL,

    user_id      UUID         NOT NULL,

    created_at   TIMESTAMPTZ  NOT NULL DEFAULT now(),

    expires_at   TIMESTAMPTZ  NULL,

    is_active    BOOLEAN      NOT NULL DEFAULT true

);

CREATE UNIQUE INDEX IF NOT EXISTS idx_urls_short_code ON urls(short_code);

CREATE INDEX IF NOT EXISTS idx_urls_user_id_created ON urls(user_id, created_at DESC);

CREATE TABLE IF NOT EXISTS outbox (

    id           UUID         PRIMARY KEY DEFAULT gen_random_uuid(),

    event_type   TEXT         NOT NULL,

    payload      JSONB        NOT NULL,

    created_at   TIMESTAMPTZ  NOT NULL DEFAULT now(),

    published_at TIMESTAMPTZ  NULL

);

CREATE INDEX IF NOT EXISTS idx_outbox_unpublished

    ON outbox(created_at ASC) WHERE published_at IS NULL;

`
```

5.8 Route Registration in `main.go`

```go
// main.go (extended from M1)

mux := http.NewServeMux()

mux.HandleFunc("GET /health", NewHealthHandler(cfg.ServiceName))

// Protected routes (JWT required)

authMw := auth.JWTMiddleware(cfg.JWTSecret)

mux.Handle("POST /shorten",       authMw(http.HandlerFunc(h.Shorten)))

mux.Handle("GET /urls",           authMw(http.HandlerFunc(h.ListURLs)))

mux.Handle("DELETE /urls/{code}", authMw(http.HandlerFunc(h.DeleteURL)))

// Anonymous route

mux.HandleFunc("GET /{code}", h.Redirect)

// Start outbox coordinator (before HTTP server, after RabbitMQ connected)

appCtx, appCancel := context.WithCancel(context.Background())

coordinator := NewOutboxCoordinator(

    outboxStore,

    NewRabbitMQPublisher(mq.Channel, log),

    log,

    cfg.OutboxPollInterval,

    cfg.OutboxWorkerCount,

)

go coordinator.Run(appCtx)

// On SIGTERM/SIGINT: appCancel() before pool.Close()
```


Base62 Short Code Generation Algorithm
62^7 = 3,521,614,606,208 unique codes · crypto/rand only · 5-retry collision budget
ALPHABET CONSTANT (compile-time, immutable) "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" indices [0..9] = digits · [10..35] = A-Z · [36..61] = a-z len = 62 · no ambiguous chars (no 0/O, 1/l/I confusion)
contrast: deterministic vs entropy
seed() · Step 4
math/rand FORBIDDEN math/rand default source is seeded with a fixed value in Go < 1.20 or uses a goroutine-local state predictable from timing. An attacker can enumerate all codes generated in a session. Only crypto/rand (OS entropy pool) is permitted.
why crypto/rand is mandatory
Step 1 — Allocate 8-byte buffer buf := make([]byte, 8) 8 bytes = 64 bits of entropy capacity. Only 7 base62 digits needed (62^7 ≈ 2^41.7), so 64 bits provides ample headroom before modulo reduction.
Step 2 — Fill with crypto/rand _, err := rand.Read(buf) if err != nil { panic(...) } crypto/rand.Read draws from /dev/urandom (Linux), CryptGenRandom (Windows), or getentropy(2) (macOS). Panic is correct: entropy exhaustion = OS-level failure, process must not continue generating codes.
Step 3 — Decode to big.Int + reduce to base62 space n := new(big.Int).SetBytes(buf) // 0 ≤ n < 2^64 max := big.NewInt(62)^7 // = 3,521,614,606,208 n.Mod(n, max) // 0 ≤ n < 62^7 Modulo bias: 2^64 / 62^7 = 5,242 — the bias is (2^64 mod 62^7) / 2^64 ≈ 0.0000286 . Negligible for URL codes.
Step 4 — Base62 Encode (right-to-left, length=7) result := make([]byte, 7) for i := 6; i >= 0; i-- { n, mod = n.DivMod(n, big.NewInt(62)) result[i] = alphabet[mod.Int64()] } return string(result) // always exactly 7 chars Left-pad with alphabet[0] = '0' when n < 62^6 . DivMod extracts least-significant digit first → fills right-to-left.
encode_example
n = 42,000,000,000 (example after Mod) big.Int value before encoding
DivMod n 7
i n before DivMod rem alphabet[rem]
6 42,000,000,000 56 q
5 677,419,354 20 k
4 10,926,118 54 u
3 176,228 52 s
2 2,842 42 g
1 45 45 t
0 0 0 0
string(result)
result = "0jkuSG" ? check - all from base62 alphabet ✓
Generate() · outer + attempt loop
retry_loop
Collision-Retry Loop for attempt := 1; attempt <= 5;
attempt = 1
Generate code shortCode = generator.Generate() (Steps 1–4 above) new crypto/rand call each attempt
attempt loop ends
INSERT INTO urls + outbox (single tx) BEGIN INSERT INTO urls (short_code, ...) VALUES ($1, ...) INSERT INTO outbox (event_type, payload) VALUES (...) COMMIT
pgError.Code == "23505" ?
no conflict (nil error)
conflict + custom code
conflict + auto code
attempt < 5 shortCode = ""
SUCCESS Return 201 with {short_code, short_url, ...} outbox row will be published by background worker
UNIQUE VIOLATION — auto-generated code Rollback tx · shortCode = "" · increment attempt Re-enter loop with fresh crypto/rand call
UNIQUE VIOLATION — custom code Rollback tx · NO retry (user chose this code) Return 409 {"error":"short code already taken","field":"custom_code"}
attempt = 5
ALL 5 ATTEMPTS EXHAUSTED 503 {"error":"service temporarily unavailable, try again"} Log: log.Error("short code collision exhausted", "attempts", 5) At 1M URLs: P(all 5 collide) ≈ (1M/3.5T)^5 ≈ 4.5 × 10^-33
at-scale collision math
Collision Probability Analysis
URLs in DB (N) P(single collision) P(all 5 collide)
1,000 2.84 × 10^-10 negligible
1,000,000 2.84 × 10^-7 4.5 × 10^-33
1,000,000,000 2.84 × 10^-4 1.6 × 10^-17 %
Formula: P = N / 62^7 per attempt.
At 1B URLs (28% space used), 503-rate ≈ 0.28% — requires longer codes.

```
main.go startup sequence (url-service, M3 extended):
  loadConfig()
       | ← fatal if DATABASE_URL | RABBITMQ_URL | REDIS_URL | JWT_SECRET missing
       ▼
  logger.New("url-service")
       ▼
  NewDBPool() ── fatal on error
       ▼
  runMigrations(pool) ── fatal on error
       ▼
  NewRedisClient() ── warn+continue on error
       ▼
  NewRabbitMQConn() ── fatal after 10 attempts
       ▼
  NewURLStore(pool)
  NewOutboxStore(pool)
  NewShortCodeGenerator()
  NewRedisCache(redisClient, log)
  NewRabbitMQPublisher(mq.Channel, log)
  NewHandler(...)
       ▼
  coordinator := NewOutboxCoordinator(outboxStore, publisher, log, 2s, 3)
  go coordinator.Run(appCtx)     ← background goroutine
       ▼
  mux := http.NewServeMux()
  mux.Handle(...)                ← all routes registered
       ▼
  srv.ListenAndServe()           ← blocks
       |
       | SIGTERM received
       ▼
  appCancel()   ← stops coordinator and workers
  srv.Shutdown(ctx)
  mq.Close()
  redisClient.Close()
  pool.Close()
```

5.9 `validateURL` Function

```go
// validate.go

func validateURL(rawURL string) error {

    if rawURL == "" {

        return errors.New("url is required")

    }

    u, err := url.Parse(rawURL)

    if err != nil {

        return fmt.Errorf("invalid url: %w", err)

    }

    if u.Scheme == "" {

        return errors.New("url must include a scheme (http:// or https://)")

    }

    if u.Scheme != "http" && u.Scheme != "https" {

        return fmt.Errorf("url scheme must be http or https, got %q", u.Scheme)

    }

    if u.Host == "" {

        return errors.New("url must include a host")

    }

    return nil

}
```

5.10 Correlation ID Propagation

```go
// handler.go helper — reads X-Correlation-ID from request context or header
// The correlation ID is injected by the gateway in M5; for direct calls it may be absent.
type correlationKey struct{}
func correlationIDFromContext(ctx context.Context) string {
    if id, ok := ctx.Value(correlationKey{}).(string); ok && id != "" {
        return id
    }
    return newUUID() // generate one if missing (direct service call without gateway)
}
// Middleware to read X-Correlation-ID header and inject into context.
// Applied to all routes in main.go before auth middleware.
func CorrelationIDMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        id := r.Header.Get("X-Correlation-ID")
        if id == "" {
            id = newUUID()
        }
        ctx := context.WithValue(r.Context(), correlationKey{}, id)
        w.Header().Set("X-Correlation-ID", id) // echo back to client
        next.ServeHTTP(w, r.WithContext(ctx))
    })
}
// newUUID returns a random UUID v4 string using crypto/rand.
func newUUID() string {
    b := make([]byte, 16)
    rand.Read(b)
    b[6] = (b[6] & 0x0f) | 0x40 // version 4
    b[8] = (b[8] & 0x3f) | 0x80 // variant bits
    return fmt.Sprintf("%08x-%04x-%04x-%04x-%012x",
        b[0:4], b[4:6], b[6:8], b[8:10], b[10:16])
}
```

6. Error Handling Matrix

Error Scenario	Detected By	HTTP Status	Response Body	Log Level	Notes
Missing JWT / invalid token	`JWTMiddleware`	401	`{"error":"unauthorized"}`	none	Handled before handler
Invalid JSON request body	`json.Decode`	400	`{"error":"invalid request body"}`	Warn	Log path + size
URL missing scheme	`validateURL`	422	`{"error":"url must include a scheme...","field":"url"}`	none	
URL non-http scheme	`validateURL`	422	`{"error":"url scheme must be http or https...","field":"url"}`	none	
URL missing host	`validateURL`	422	`{"error":"url must include a host","field":"url"}`	none	
`expires_at` malformed	RFC3339 parse	422	`{"error":"expires_at must be RFC3339 format","field":"expires_at"}`	none	
`expires_at` in past	time comparison	422	`{"error":"expires_at must be in the future","field":"expires_at"}`	none	
Custom code already taken	`ErrShortCodeConflict`, `custom_code != nil`	409	`{"error":"short code already taken","field":"custom_code"}`	Info	
Auto-code collision × 5	`ErrShortCodeConflict` × 5, attempts exhausted	503	`{"error":"service temporarily unavailable, try again"}`	Error	Include attempt count
DB error on INSERT urls	`InsertWithURL` non-conflict	500	`{"error":"internal server error"}`	Error	Log full error
DB begin tx error	`pool.Begin`	500	`{"error":"internal server error"}`	Error	
DB commit error	`tx.Commit`	500	`{"error":"internal server error"}`	Error	
`GET /:code` not found	`ErrURLNotFound`	404	`{"error":"short url not found"}`	none	
`GET /:code` is_active=false	`urlRec.IsActive == false`	410	`{"error":"url has been deactivated"}`	none	
`GET /:code` expired	`expires_at < now()`	410	`{"error":"url has expired"}`	none	
Redis GET error	`client.Get` non-nil error	cache miss (fall through)	none (no HTTP error)	Warn	`"redis get failed"`
Redis SET error	`client.Set` non-nil error	none (redirect succeeds)	none	Warn	`"redis set failed"`
Redis DEL error	`client.Del` non-nil error	none (204 proceeds)	none	Warn	`"redis del failed"`
`GET /urls` DB error	`FindByUserID`	500	`{"error":"internal server error"}`	Error	

Error Scenario	Detected By	HTTP Status	Response Body	Log Level	Notes
`GET /urls` invalid cursor	UUID parse fail on `after` param	400	`{"error":"invalid cursor","field":"after"}`	none	
`DELETE` URL not found	`ErrURLNotFound`	404	`{"error":"short url not found"}`	none	
`DELETE` wrong owner	`urlRec.UserID != claims.Sub`	403	`{"error":"forbidden"}`	Info	Log user_id + owner
`DELETE` DB error	any non-sentinel DB err	500	`{"error":"internal server error"}`	Error	
Outbox publish failure	`amqpPublisher.Publish`	no HTTP (background)	—	Warn	`"outbox_id"`, `"event_type"`, `"error"`
Outbox MarkPublished failure	`MarkPublished`	no HTTP (background)	—	Warn	Same fields; row re-published next cycle
Outbox FetchUnpublished failure	`FetchUnpublished`	no HTTP (background)	—	Warn	`"error"`; coordinator continues next tick
Click event outbox insert failure	tx.Exec or InsertEvent	no HTTP (redirect proceeds)	—	Warn	`"code"`, `"error"`
`crypto/rand.Read` failure	`ShortCodeGenerator.Generate`	process panic	—	—	Unrecoverable; OS entropy exhausted

`writeError` helper:

```go
// errors.go

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

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

    w.WriteHeader(status)

    resp := struct {

        Error string `json:"error"`

        Field string `json:"field,omitempty"`

    }{Error: message, Field: field}

    json.NewEncoder(w).Encode(resp)

}

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

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

    w.WriteHeader(status)

    json.NewEncoder(w).Encode(v)

}
```

Sentinel errors:

```go
// errors.go

var (

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

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

    ErrNotOwner          = errors.New("not the url owner")

)
```

7. Concurrency Specification

Component	Goroutine	Shared State	Thread Safety
HTTP handlers	per-request (net/http)	`*pgxpool.Pool`, `*RedisCache`, `*OutboxStore`	✅ All safe: pgxpool is goroutine-safe; RedisCache is stateless; OutboxStore holds only pool
`OutboxCoordinator.Run`	1 dedicated goroutine	`workerCh chan *OutboxRecord`, `OutboxRepository`	✅ Channel is the only shared state; only coordinator sends
`OutboxWorker.runWorker`	3 dedicated goroutines	`workerCh` (read-only), `RabbitMQPublisher`, `OutboxRepository`	✅ Workers only read from channel; pool is goroutine-safe
`amqpPublisher.Publish`	called from worker goroutines	`*amqp.Channel`	⚠️ **NOT safe** for concurrent calls — see below
`RedisCache`	called from HTTP goroutines	`*redis.Client`	✅ go-redis client is goroutine-safe

AMQP Channel concurrency: `amqp091-go` channels are **not** goroutine-safe for concurrent publishes. Since 3 workers share one publisher, the publisher must protect the channel with a mutex:

```go
// publisher.go

type amqpPublisher struct {

    mu           sync.Mutex

    ch           *amqp.Channel

    exchangeName string

    log          *slog.Logger

}

func (p *amqpPublisher) Publish(ctx context.Context, routingKey string, body []byte) error {

    p.mu.Lock()

    defer p.mu.Unlock()

    // ... PublishWithContext call ...

}
```

Alternatively, give each worker its own AMQP channel (3 channels from one connection). The mutex approach is simpler for this project. For higher throughput, switch to per-worker channels in a production system. **Context cancellation flow:**

```
main.go: SIGTERM received
  → appCancel()
  → coordinator's ctx.Done() fires in select loop
  → coordinator closes workerCh
  → workers drain workerCh and exit (range loop terminates on close)
  → wg.Wait() in coordinator returns
  → coordinator returns
  → main.go proceeds to srv.Shutdown()
```

Timeout contexts for Redis:

- All Redis operations must use a `context.WithTimeout` with 50ms for GET and 100ms for SET/DEL.
- Never use `r.Context()` directly for Redis — if the HTTP client disconnects, the context is cancelled and the cache operation is aborted, but the handler continues regardless.

---

8. Implementation Sequence with Checkpoints

Phase 1: DB Schema — `urls` + `outbox` Tables, Migrations, Indexes (1h)

1. Write `migration.sql` as specified in §3.1.
2. Extend `main.go` to call `runMigrations` (add `urlServiceSchema` const, `runMigrations` func as in §5.7).
3. Extend `config.go` to add `JWTSecret`, `ShortURLBase`, `OutboxPollInterval`, `OutboxWorkerCount`.
4. Run against Docker `url_db`:

```bash
docker compose up url_db -d

DATABASE_URL="postgres://urluser:urlpass@localhost:5432/urldb" \

JWT_SECRET="testjwtsecret-minimum-32-chars-long" \

SHORT_URL_BASE="http://localhost:8081" \

go run ./services/url-service/
```

1. Verify schema:

```bash
psql -h localhost -p 5432 -U urluser -d urldb \

  -c "\d urls; \d outbox; \di idx_urls_*; \di idx_outbox_*;"
```

Checkpoint: Tables `urls` and `outbox` exist with correct columns and types. All four indexes visible in `\di` output. Service exits 0 after migration.

Phase 2: Base62 Encoder + `ShortCodeGenerator` with Collision Retry (1–1.5h)

1. Write `base62.go`: `base62Alphabet` constant, `Encode(*big.Int) string`, `Decode(string) (*big.Int, error)`.
2. Write `codegen.go`: `ShortCodeGenerator`, `Generate() string`.
3. Write initial tests in `url_test.go` (§10.1):
   - `TestBase62RoundTrip`
   - `TestBase62Length`
   - `TestBase62Alphabet`
   - `TestShortCodeUniqueness`

```bash
go test ./services/url-service/ -run TestBase62 -v

go test ./services/url-service/ -run TestShortCode -v
```

Checkpoint: All base62 and codegen tests pass. `Generate()` always returns exactly 7 characters from the base62 alphabet.

Phase 3: `POST /shorten` Handler + Atomic URL+Outbox Insert (2–2.5h)

1. Write `errors.go`: sentinels, `writeError`, `writeJSON`, `isPgUniqueViolation`.
2. Write `validate.go`: `validateURL`.
3. Write `store.go`: `URLRecord`, `URLRepository` interface, `pgxURLStore` with `Insert` (unique violation detection).
4. Write `outbox_store.go`: `OutboxRecord`, `OutboxRepository` interface, `pgxOutboxStore` with `InsertWithURL` and `InsertEvent`.
5. Write `handler.go`: `Handler` struct, `NewHandler`, `Shorten` method (full algorithm from §5.2).
6. Wire in `main.go`: construct all dependencies, register `POST /shorten` route with `JWTMiddleware`.
7. Add `CorrelationIDMiddleware` and wrap the mux with it.

```bash
# Start infra
docker compose up url_db rabbitmq redis -d

# Start service with env vars
DATABASE_URL=... RABBITMQ_URL=... REDIS_URL=... JWT_SECRET=... \
SHORT_URL_BASE="http://localhost:8081" \
go run ./services/url-service/

# Test (get a token from user-service first, or generate a test JWT)
curl -X POST http://localhost:8081/shorten \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"url":"https://example.com/very/long/path"}'
```

Checkpoint: `POST /shorten` returns `201` with `{short_code, short_url, original_url}`. PostgreSQL `urls` table has 1 row. `outbox` table has 1 row with `published_at = NULL`. Duplicate URL returns 201 with a different short code. Invalid URL returns 422.

Phase 4: Redis Cache Helpers + `GET /:code` Redirect (2–2.5h)

1. Write `cache.go`: `CachedURL`, `RedisCache`, `Get/Set/Del` methods with full error isolation.
2. Write `publisher.go`: `RabbitMQPublisher` interface, `amqpPublisher` with mutex.
3. Add `Redirect` method to `handler.go` (full algorithm from §5.3).
4. Register `GET /{code}` route in `main.go` (anonymous, no auth middleware).

```bash
# After POST /shorten returned {"short_code":"abc1234", ...}
curl -v http://localhost:8081/abc1234
# Expect: HTTP/1.1 301 Moved Permanently
#         Location: https://example.com/very/long/path

# Second request (cache hit)
curl -v http://localhost:8081/abc1234
# Expect: same 301, served from Redis in < 5ms

# Redis verification
redis-cli -p 6379 GET "url:abc1234"
# Expect: {"original_url":"https://example.com/very/long/path","is_active":true}
```

Checkpoint: First redirect: EXPLAIN ANALYZE on `SELECT ... WHERE short_code = $1` uses `idx_urls_short_code`. Second redirect: Redis `GET url:<code>` returns JSON (cache hit). `outbox` table gains 1 row with `event_type = "url.clicked"` and `published_at = NULL`. Non-existent code returns 404. Redis key for expired URL has TTL set correctly.

Phase 5: `GET /urls` + `DELETE /urls/:code` (1.5–2h)

1. Add `FindByUserID` to `pgxURLStore` (cursor pagination with `limit+1` trick).
2. Add `Deactivate` to `pgxURLStore`.
3. Add `ListURLs` and `DeleteURL` methods to `Handler`.
4. Register routes in `main.go`.

```bash
# List URLs
curl http://localhost:8081/urls \
  -H "Authorization: Bearer $TOKEN"
# Expect: {"urls":[...],"next_cursor":null} (first page)
# Pagination test (create 10+ URLs first)
curl "http://localhost:8081/urls?after=<uuid_from_previous_response>"
# Delete
curl -X DELETE http://localhost:8081/urls/abc1234 \
  -H "Authorization: Bearer $TOKEN"
# Expect: 204 No Content
# Verify deactivated
curl http://localhost:8081/abc1234
# Expect: 410 Gone
# Verify Redis cleared
redis-cli -p 6379 EXISTS "url:abc1234"
# Expect: (integer) 0
# Delete wrong owner
curl -X DELETE http://localhost:8081/urls/abc1234 \
  -H "Authorization: Bearer $OTHER_USER_TOKEN"
# Expect: 403 Forbidden
# Verify outbox has URLDeletedEvent
psql -h localhost -p 5432 -U urluser -d urldb \
  -c "SELECT event_type, payload FROM outbox WHERE event_type = 'url.deleted';"
```

Checkpoint: `EXPLAIN ANALYZE` on `GET /urls` query shows `Index Scan using idx_urls_user_id_created`. Pagination cursor works correctly. DELETE sets `is_active=false` in DB. Outbox has `url.deleted` row. Redis key deleted. 403 returned for non-owner.

Phase 6: Outbox Coordinator + 3-Worker Pool + Graceful Shutdown (2–2.5h)

1. Write `outbox.go`: `OutboxCoordinator`, `runWorker`, coordinator poll loop (algorithm from §5.5).
2. Start coordinator in `main.go` before HTTP server.
3. Wire `appCancel` to OS signal handler.
4. Start `analytics-service` + `notification-service` (M1 skeletons) to receive messages.

```bash
docker compose up --build -d

# Shorten a URL
curl -X POST http://localhost:8081/shorten \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"url":"https://example.com"}' \
  -H "Content-Type: application/json"

# Wait 3s for outbox poll
sleep 3

# Verify published
psql -h localhost -p 5432 -U urluser -d urldb \
  -c "SELECT event_type, published_at FROM outbox ORDER BY created_at DESC LIMIT 5;"

# Expect: url.created row with published_at != NULL

# Check RabbitMQ management UI

# http://localhost:15672/#/queues → analytics.clicks and notifications.events

# should show message counts incrementing
```

Checkpoint: Outbox rows transition from `published_at = NULL` to `published_at = <timestamp>` within 2-5 seconds of creation. RabbitMQ management UI shows messages delivered to queues. Coordinator logs `"outbox coordinator stopped"` on SIGTERM. All workers exit cleanly (no goroutine leak).

Phase 7: Tests + Benchmark (1–1.5h)

1. Complete `url_test.go` with all test cases from §10.
2. Write `bench_test.go` with the redirect benchmark.
3. Run all tests:

```bash
go test ./services/url-service/... -v -count=1
```

1. Run redirect benchmark:

```bash
go test ./services/url-service/ -bench=BenchmarkRedirectCacheHit -benchtime=30s -benchmem
```

Checkpoint: All unit tests pass. Redirect benchmark reports p99 < 5ms under 100 concurrent

simulated requests (see §9 for wrk command)

9. Test Specification

9.1 Base62 Tests (`url_test.go`)

```go
func TestBase62RoundTrip(t *testing.T) {
    cases := []int64{0, 1, 61, 62, 3521614606207} // 0, 1, last single-digit, first two-digit, max 7-char
    for _, n := range cases {
        t.Run(fmt.Sprintf("n=%d", n), func(t *testing.T) {
            encoded := Encode(big.NewInt(n))
            decoded, err := Decode(encoded)
            if err != nil {
                t.Fatalf("Decode: %v", err)
            }
            if decoded.Int64() != n {
                t.Errorf("round trip: got %d want %d", decoded.Int64(), n)
            }
        })
    }
}
func TestBase62Length(t *testing.T) {
    // All encoded values must be exactly 7 chars (left-padded)
    for _, n := range []*big.Int{big.NewInt(0), big.NewInt(1), big.NewInt(61)} {
        s := Encode(n)
        if len(s) != 7 {
            t.Errorf("Encode(%v) = %q len=%d, want 7", n, s, len(s))
        }
    }
}
func TestBase62Alphabet(t *testing.T) {
    if len(base62Alphabet) != 62 {
        t.Errorf("alphabet length: got %d want 62", len(base62Alphabet))
    }
    seen := make(map[byte]bool)
    for i := 0; i < len(base62Alphabet); i++ {
        c := base62Alphabet[i]
        if seen[c] {
```

```
            t.Errorf("duplicate character %q at index %d", c, i)
        }
        seen[c] = true
    }
    // Must start with digits, then uppercase, then lowercase
    if base62Alphabet[0] != '0' || base62Alphabet[9] != '9' {
        t.Error("first 10 chars must be 0-9")
    }
    if base62Alphabet[10] != 'A' || base62Alphabet[35] != 'Z' {
        t.Error("chars 10-35 must be A-Z")
    }
    if base62Alphabet[36] != 'a' || base62Alphabet[61] != 'z' {
        t.Error("chars 36-61 must be a-z")
    }
}
func TestShortCodeLength(t *testing.T) {
    g := NewShortCodeGenerator()
    for i := 0; i < 1000; i++ {
        code := g.Generate()
        if len(code) != 7 {
            t.Fatalf("generated code %q has length %d, want 7", code, len(code))
        }
    }
}
func TestShortCodeAlphabet(t *testing.T) {
    g := NewShortCodeGenerator()
    valid := make(map[byte]bool)
    for i := 0; i < len(base62Alphabet); i++ {
        valid[base62Alphabet[i]] = true
    }
    for i := 0; i < 100; i++ {
        code := g.Generate()
        for _, c := range []byte(code) {
            if !valid[c] {
                t.Fatalf("invalid char %q in code %q", c, code)
            }
```

```
        }
    }
}
func TestShortCodeUniqueness(t *testing.T) {
    g := NewShortCodeGenerator()
    seen := make(map[string]bool)
    // Generate 10,000 codes; collision probability at 10k ≈ 1.4% — acceptable for test
    // (we're testing the generator produces varied output, not that it's collision-free)
    for i := 0; i < 10_000; i++ {
        code := g.Generate()
        seen[code] = true
    }
    // Expect at least 9990 unique codes out of 10,000
    if len(seen) < 9990 {
        t.Errorf("too many collisions: got %d unique codes from 10,000 generates", len(seen))
    }
}
```

9.2 `validateURL` Tests

```go
func TestValidateURL(t *testing.T) {

    cases := []struct {

        input   string

        wantErr bool

    }{

        {"https://example.com", false},

        {"http://example.com/path?q=1", false},

        {"https://sub.domain.org/a/b/c", false},

        {"", true},                          // empty

        {"not-a-url", true},                 // no scheme

        {"ftp://example.com", true},         // non-http scheme

        {"http://", true},                   // scheme but no host

        {"//example.com", true},             // no scheme

        {"javascript:alert(1)", true},       // non-http scheme

    }

    for _, tc := range cases {

        t.Run(tc.input, func(t *testing.T) {

            err := validateURL(tc.input)

            if (err != nil) != tc.wantErr {

                t.Errorf("validateURL(%q) err=%v wantErr=%v", tc.input, err, tc.wantErr)

            }

        })

    }

}
```

9.3 Handler Unit Tests (with mock stores)

```go
// Mock implementations

type mockURLStore struct {

    insertFn       func(ctx context.Context, rec *URLRecord) (*URLRecord, error)

    findByCodeFn   func(ctx context.Context, code string) (*URLRecord, error)

    findByUserIDFn func(ctx context.Context, userID, afterID string, limit int) ([]*URLRecord, string, error)

    deactivateFn   func(ctx context.Context, code, userID string) error

}

// (implement URLRepository interface — each method delegates to respective Fn)

type mockOutboxStore struct {

    insertWithURLFn func(ctx context.Context, tx pgx.Tx, u *URLRecord, o *OutboxRecord) error

    insertEventFn   func(ctx context.Context, tx pgx.Tx, o *OutboxRecord) error

    fetchFn         func(ctx context.Context, limit int) ([]*OutboxRecord, error)

    markFn          func(ctx context.Context, id string) error

}

func TestShortenHandler_Success(t *testing.T) {

    // Verify: 201 response, short_code 7 chars, short_url has base prefix

}

func TestShortenHandler_InvalidURL(t *testing.T) {

    // Input: {"url": "not-a-url"}

    // Expect: 422 with field="url"

}

func TestShortenHandler_ExpiredExpiresAt(t *testing.T) {

    // Input: expires_at = one hour ago

    // Expect: 422

}

func TestShortenHandler_CustomCodeConflict(t *testing.T) {

    // mockOutboxStore.insertWithURLFn returns ErrShortCodeConflict

    // req.custom_code = "mycode"

    // Expect: 409, no retry

}

func TestShortenHandler_AutoCodeCollisionExhausted(t *testing.T) {

    // insertWithURLFn always returns ErrShortCodeConflict

    // req has no custom_code

    // Expect: 503 after 5 attempts (verify mock called exactly 5 times)

}
```

```go
func TestRedirectHandler_CacheHit_Active(t *testing.T) {
    // mockCache.Get returns &CachedURL{OriginalURL:"https://x.com", IsActive:true}
    // Expect: 301 Location: https://x.com; no DB call (mockURLStore.findByCode not called)
}
func TestRedirectHandler_CacheHit_Inactive(t *testing.T) {
    // cached.IsActive = false
    // Expect: 410
}
func TestRedirectHandler_CacheMiss_Active(t *testing.T) {
    // cache returns miss; store returns active URLRecord
    // Expect: 301; cache.Set called
}
func TestRedirectHandler_CacheMiss_NotFound(t *testing.T) {
    // cache miss; store returns ErrURLNotFound
    // Expect: 404
}
func TestRedirectHandler_CacheMiss_Expired(t *testing.T) {
    // cache miss; store returns URLRecord with ExpiresAt = 1 hour ago
    // Expect: 410; cache.Set NOT called
}
func TestDeleteHandler_Success(t *testing.T) {
    // store returns owned URLRecord; deactivate succeeds; outbox insert succeeds
    // Expect: 204; cache.Del called
}
func TestDeleteHandler_NotFound(t *testing.T) {
    // deactivate returns ErrURLNotFound
    // Expect: 404
}
func TestDeleteHandler_WrongOwner(t *testing.T) {
    // urlRec.UserID != claims.Sub
    // Expect: 403
}
func TestListURLsHandler_EmptyPage(t *testing.T) {
    // store.FindByUserID returns empty slice, ""
    // Expect: 200 {"urls":[],"next_cursor":null}
}
```

```
func TestListURLsHandler_WithNextCursor(t *testing.T) {

    // store returns 20 items and nextCursor = "some-uuid"

    // Expect: 200 {"urls":[...],"next_cursor":"some-uuid"}

}
```

9.4 `RedisCache` Unit Tests

```go
func TestRedisCache_GetMiss(t *testing.T) {

    // Use miniredis (github.com/alicebob/miniredis/v2) for in-process Redis

    mr := miniredis.RunT(t)

    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})

    cache := NewRedisCache(client, slog.Default())

    result, hit := cache.Get(context.Background(), "nonexistent")

    if hit || result != nil {

        t.Error("expected cache miss")

    }

}

func TestRedisCache_SetThenGet(t *testing.T) {

    mr := miniredis.RunT(t)

    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})

    cache := NewRedisCache(client, slog.Default())

    cu := &CachedURL{OriginalURL: "https://example.com", IsActive: true}

    cache.Set(context.Background(), "abc1234", cu, nil)

    result, hit := cache.Get(context.Background(), "abc1234")

    if !hit {

        t.Fatal("expected cache hit")

    }

    if result.OriginalURL != "https://example.com" {

        t.Errorf("url: got %q", result.OriginalURL)

    }

}

func TestRedisCache_Del(t *testing.T) {

    mr := miniredis.RunT(t)

    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})

    cache := NewRedisCache(client, slog.Default())

    cu := &CachedURL{OriginalURL: "https://example.com", IsActive: true}

    cache.Set(context.Background(), "abc1234", cu, nil)

    cache.Del(context.Background(), "abc1234")

    _, hit := cache.Get(context.Background(), "abc1234")

    if hit {

        t.Error("expected cache miss after Del")

    }
```

```go
}

func TestRedisCache_TTL_WithExpiry(t *testing.T) {

    mr := miniredis.RunT(t)

    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})

    cache := NewRedisCache(client, slog.Default())

    // ExpiresAt 30 minutes from now → TTL should be ≈ 30min (< 1h)

    expiresAt := time.Now().Add(30 * time.Minute)

    cache.Set(context.Background(), "code1", &CachedURL{OriginalURL: "x", IsActive: true}, &expiresAt)

    // miniredis doesn't check exact TTL but we can verify key exists

    _, hit := cache.Get(context.Background(), "code1")

    if !hit {

        t.Error("expected hit for unexpired key")

    }

    // Fast-forward 31 minutes in miniredis

    mr.FastForward(31 * time.Minute)

    _, hit = cache.Get(context.Background(), "code1")

    if hit {

        t.Error("expected cache miss after TTL expiry")

    }

}

func TestRedisCache_ErrorIsolation(t *testing.T) {

    // Closed Redis client — verify Get returns (nil, false), not panics

    client := redis.NewClient(&redis.Options{Addr: "localhost:1"}) // nothing listening

    cache := NewRedisCache(client, slog.Default())

    result, hit := cache.Get(context.Background(), "code")

    if hit || result != nil {

        t.Error("expected miss for unreachable Redis")

    }

    // Del should not panic

    cache.Del(context.Background(), "code")

}
```

9.5 Outbox Worker Test

```go
func TestOutboxWorker_PublishAndMark(t *testing.T) {
    published := make([]string, 0)
    marked := make([]string, 0)
    var mu sync.Mutex
    publisher := &mockPublisher{
        publishFn: func(ctx context.Context, routingKey string, body []byte) error {
            mu.Lock(); published = append(published, routingKey); mu.Unlock()
            return nil
        },
    }
    store := &mockOutboxStore{
        markFn: func(ctx context.Context, id string) error {
            mu.Lock(); marked = append(marked, id); mu.Unlock()
            return nil
        },
    }
    ch := make(chan *OutboxRecord, 1)
    ch <- &OutboxRecord{ID: "outbox-1", EventType: "url.created", Payload: []byte(`{}`)}
    close(ch)
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        defer wg.Done()
        // Simulate a single worker run
        ctx := context.Background()
        for row := range ch {
            publisher.Publish(ctx, row.EventType, row.Payload)
            store.MarkPublished(ctx, row.ID)
        }
    }()
    wg.Wait()
    if len(published) != 1 || published[0] != "url.created" {
        t.Errorf("published: %v", published)
    }
    if len(marked) != 1 || marked[0] != "outbox-1" {
```

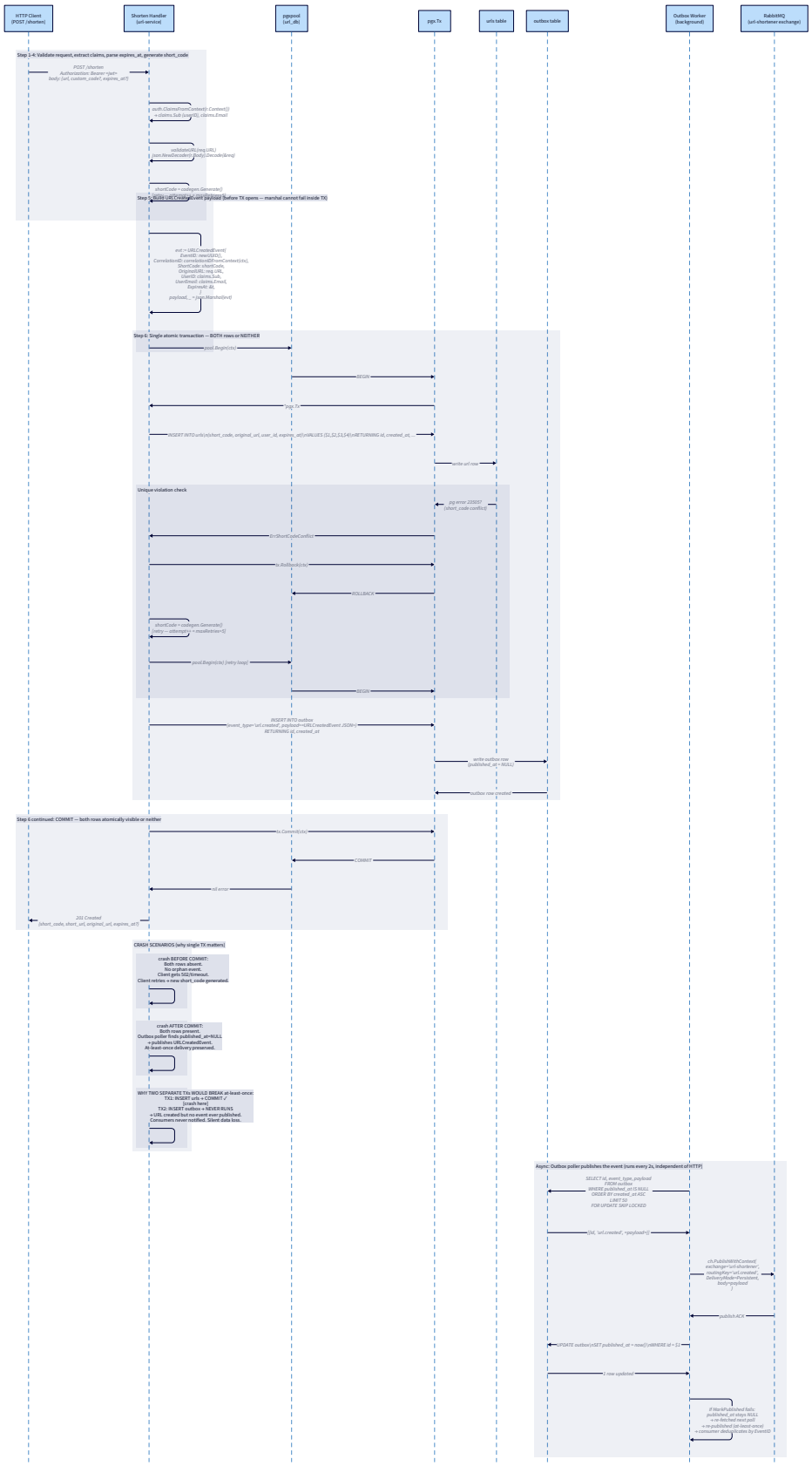
```
        t.Errorf("marked: %v", marked)
    }
}
func TestOutboxWorker_PublishFail_NoMark(t *testing.T) {
    // If publish fails, MarkPublished must NOT be called (row stays unpublished for retry)
    markCalled := false
    publisher := &mockPublisher{publishFn: func(_ context.Context, _ string, _ []byte) error {
        return errors.New("amqp error")
    }}
    store := &mockOutboxStore{markFn: func(_ context.Context, _ string) error {
        markCalled = true; return nil
    }}
    // ... run worker with one row, publish fails ...
    if markCalled {
        t.Error("MarkPublished must not be called when Publish fails")
    }
}
```

9.6 Benchmark (`bench_test.go`)

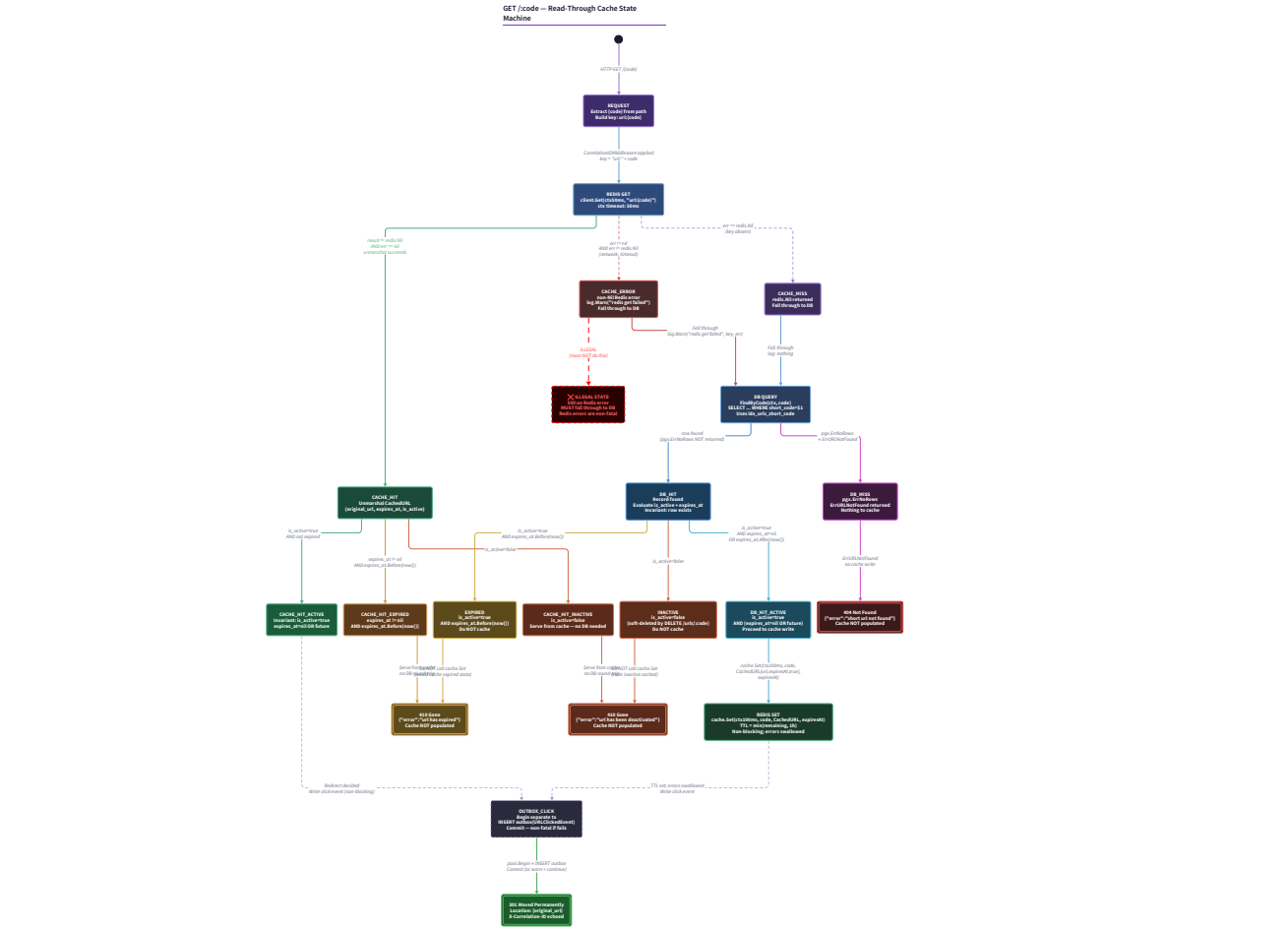
```go
// +build integration

// Run: DATABASE_URL=... REDIS_URL=... go test -bench=BenchmarkRedirectCacheHit -benchtime=30s

func BenchmarkRedirectCacheHit(b *testing.B) {

    // Setup: insert a URL record and warm the Redis cache

    // ...

    b.ResetTimer()

    b.SetParallelism(100)

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

        for pb.Next() {

            req := httptest.NewRequest("GET", "/"+shortCode, nil)

            rec := httptest.NewRecorder()

            handler.Redirect(rec, req)

            if rec.Code != 301 {

                b.Errorf("expected 301, got %d", rec.Code)

            }

        }

    })

}
```

wrk-based end-to-end benchmark:

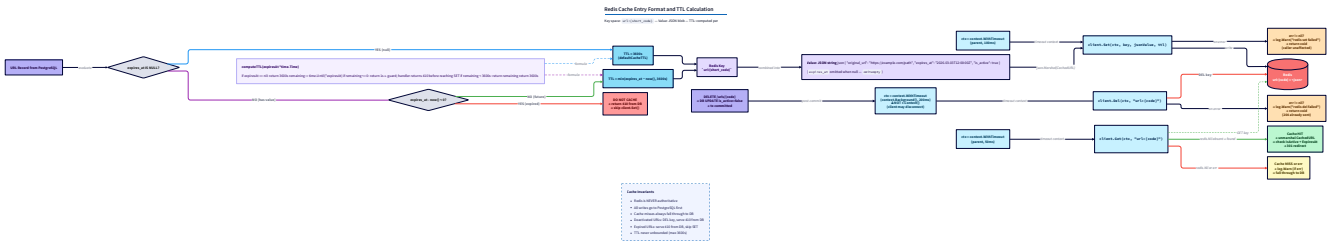
```bash
# Warm the cache first:

curl http://localhost:8081/<code> > /dev/null

# Then benchmark:

wrk -t4 -c100 -d30s http://localhost:8081/<code>

# Target: Latency 99% < 5ms in "Latency Distribution" output
```


HTTP Client (POST /shorten)
Shorten Handler (url-service)
pgxpool (url_db)
pgx.Tx
urls table
outbox table
Outbox Worker (background)
RabbitMQ (url-shortener exchange)
Step 1-4: Validate request, extract claims, parse expires_at, generate short_code
POST /shorten
Authorization: Bearer <jwt>
body: {url, custom_code?, expires_at?}
auth.ClaimsFromContext(r.Context())
→ claims.Sub (userID), claims.Email
validateURL(req.URL)
json.NewDecoder(r.Body).Decode(&req)
shortCode = codegen.Generate()
Step 5: Build URLCreatedEvent payload (before TX opens — marshal cannot fail inside TX)
evt := URLCreatedEvent{
EventID: new(UUIDV1),
CorrelationID: correlationIDFromContext(ctx),
ShortCode: shortCode,
OriginalURL: req.URL,
UserID: claims.Sub,
UserEmail: claims.Email,
ExpiresAt: &t,
}
payload, _ := evt.Marshal(evt)
Step 6: Single atomic transaction — BOTH rows or NEITHER
pool.Begin(ctx)
BEGIN
tx pgx.Tx
INSERT INTO urls (id,short_code, original_url, user_id, expires_at) VALUES ($1,$2,$3,$4) RETURNING id, created_at,
write url row
Unique violation check
pg error 23505?
short_code conflict
23505 ShortCodeConflict
tx.Rollback(ctx)
ROLLBACK
shortCode = codegen.Generate()
retry — attempt++ < maxRetries=5
pool.Begin(ctx) (retry loop)
BEGIN
INSERT INTO outbox
(event_type='url.created', payload=<URLCreatedEvent JSON>)
RETURNING id, created_at
write outbox row
(published_at = NULL)
outbox row created
Step 6 continued: COMMIT — both rows atomically visible or neither
tx.Commit(ctx)
COMMIT
nil error
201 Created
{short_code, short_url, original_url, expires_at?}
CRASH SCENARIOS (why single TX matters)
crash BEFORE COMMIT:
Both rows absent.
No orphan event.
Client gets 502/timeout.
Client retries → new short_code generated.
crash AFTER COMMIT:
Both rows present.
Outbox poller finds published_at=NULL
→ publishes URLCreatedEvent.
At-least-once delivery preserved.
WHY TWO SEPARATE TXs WOULD BREAK at-least-once:
TX1: INSERT urls → COMMIT ✓
[crash here]
TX2: INSERT outbox → NEVER RUNS
→ URL created but no event ever published.
Consumers never notified. Silent data loss.
Async: Outbox poller publishes the event (runs every 2s, independent of HTTP)
SELECT id, event_type, payload
FROM outbox
WHERE published_at IS NULL
ORDER BY created_at ASC
LIMIT 50
FOR UPDATE SKIP LOCKED
[id, 'url.created', <payload>]
ch.PublishWithContext(
exchange='url-shortener',
routingKey='url.created',
DeliveryMode=Persistent,
body=payload
)
publish ACK
UPDATE outbox SET published_at = now() WHERE id = $1
1 row updated
If MarkPublished fails:
published_at stays NULL
→ re-fetched next poll
→ re-published (at-least-once)
→ consumer deduplicates by EventID

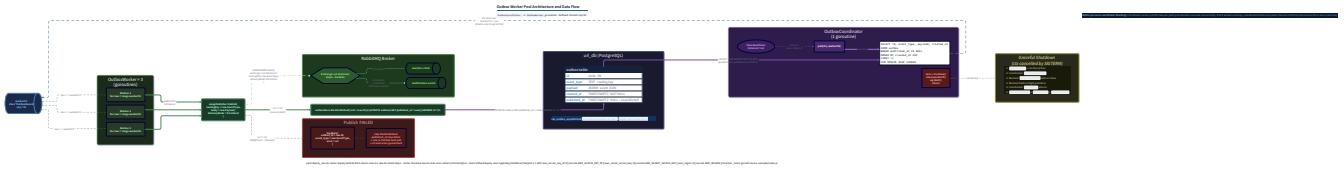
```
Request lifecycle for GET /:code (cache hit path):
  Client ──► url-service handler
                |
                ▼
        CorrelationIDMiddleware (read/generate X-Correlation-ID)
                |
                ▼
        RedisCache.Get("url:<code>") ──► Redis (50ms timeout ctx)
                |                              |
          cache hit                      cache miss or error
                |                              |
                ▼                              ▼
        Check IsActive, ExpiresAt     pgxURLStore.FindByCode()
                |                              |
          expired/inactive                active URL
                |                              |
                ▼                              ▼
              410                     RedisCache.Set() (100ms timeout ctx)
                                               |
                                               ▼
                                     Insert URLClickedEvent to outbox
                                     (separate tx, non-fatal)
                                               |
                                               ▼
                                      301 Location: <original_url>
```



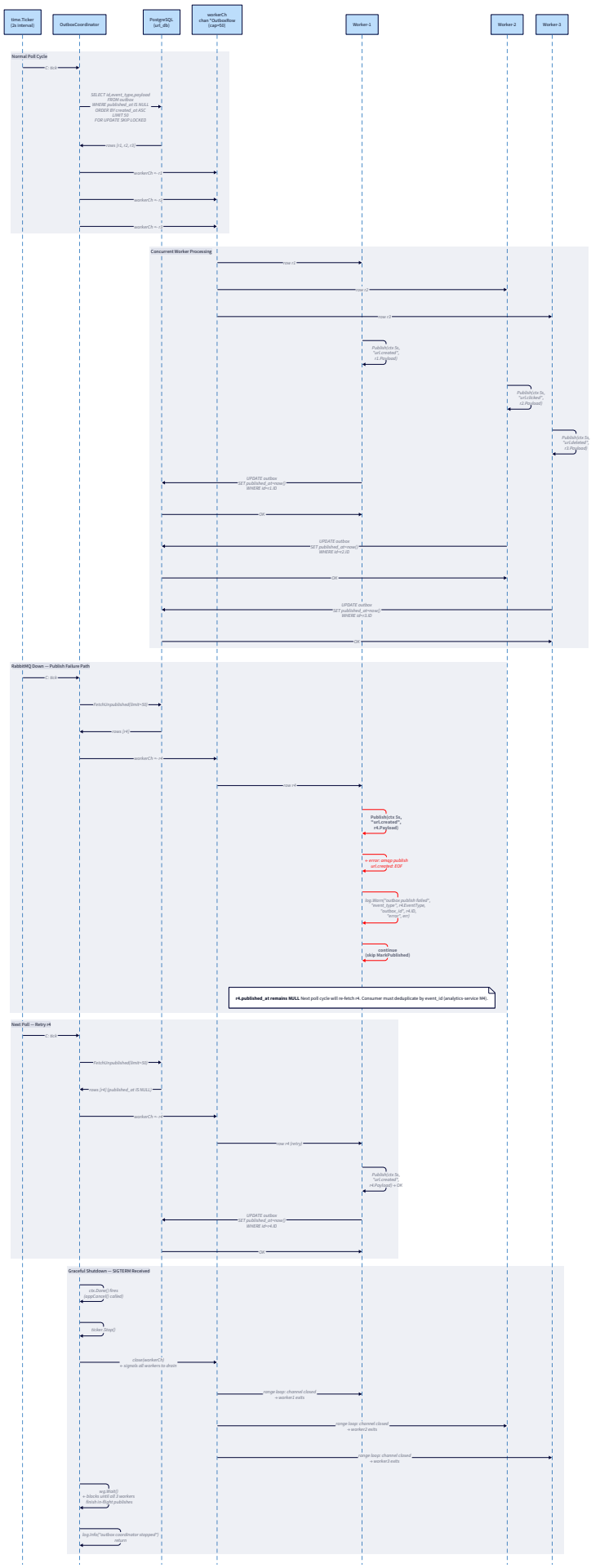
```
Transaction boundaries:
  POST /shorten:
  ┌──────────────────────────────────┐
  | BEGIN                            |
  |   INSERT INTO urls (...)         |  ← URLRecord populated from RETURNING
  |   INSERT INTO outbox (...)       |  ← URLCreatedEvent payload
  | COMMIT  ──▶ or ROLLBACK          |
  └──────────────────────────────────┘
  (one transaction, two tables, same url_db)
  GET /:code (click event):
  ┌──────────────────────────────────┐
  | BEGIN                            |
  |   INSERT INTO outbox (...)       |  ← URLClickedEvent
  | COMMIT                           |
  └──────────────────────────────────┘
  (separate transaction; redirect response sent regardless of outcome)
  DELETE /urls/:code:
  ┌──────────────────────────────────┐
  | BEGIN                            |
  |   SELECT ... FROM urls           |  ← read for ownership check
  |   UPDATE urls SET is_active=false|
  |   INSERT INTO outbox (...)       |  ← URLDeletedEvent
  | COMMIT                           |
  └──────────────────────────────────┘
  Post-commit: Redis DEL (best-effort, outside tx)
```




```
Cursor-based pagination for GET /urls:
  First page (afterID = ""):
    SELECT ... FROM urls
    WHERE user_id = $1
    ORDER BY created_at DESC, id DESC
    LIMIT 21   ← limit+1 to detect next page
    If 21 rows returned: nextCursor = rows[20].ID, return rows[:20]
    If ≤20 rows:          nextCursor = "",          return all rows
  Second page (afterID = "<uuid>"):
    SELECT ... FROM urls
    WHERE user_id = $1
      AND (created_at, id) < (SELECT created_at, id FROM urls WHERE id = $2)
    ORDER BY created_at DESC, id DESC
    LIMIT 21
  Index used: idx_urls_user_id_created ON urls(user_id, created_at DESC)
  PostgreSQL can use this index for the WHERE user_id = $1 + ORDER BY created_at DESC.
  The subquery for cursor resolution performs a single PK lookup.
```



```
Outbox coordinator poll cycle:
  t=0s:  FetchUnpublished(limit=50)
         rows = [r1, r2, r3] (3 unpublished events)
         workerCh <- r1  ──► Worker-1 picks up r1
         workerCh <- r2  ──► Worker-2 picks up r2
         workerCh <- r3  ──► Worker-3 picks up r3
  t=0.1s: Workers publish to RabbitMQ concurrently
           Worker-1: Publish("url.created", r1.Payload) → OK → MarkPublished(r1.ID)
           Worker-2: Publish("url.clicked", r2.Payload) → OK → MarkPublished(r2.ID)
           Worker-3: Publish("url.deleted", r3.Payload) → FAIL → log + skip mark
  t=2s:  FetchUnpublished(limit=50)
         rows = [r3] (r3 still has published_at = NULL)
         workerCh <- r3  ──► Worker-1 picks up r3 (retry)
         Worker-1: Publish("url.deleted", r3.Payload) → OK → MarkPublished(r3.ID)
```


time.Ticker (2s interval)
OutboxCoordinator
PostgreSQL (svc_db)
workerCh chan *OutboxRow (cap=50)
Worker-1
Worker-2
Worker-3
Normal Poll Cycle
C: tick
SELECT id,event_type,payload FROM outbox WHERE published_at IS NULL ORDER BY created_at ASC LIMIT 50 FOR UPDATE SKIP LOCKED
rows [r1, r2, r3]
workerCh <- r1
workerCh <- r2
workerCh <- r3
Concurrent Worker Processing
row r1
row r2
row r3
Publish(ctx St, "url.created", r1.Payload)
Publish(ctx St, "url.clicked", r2.Payload)
Publish(ctx St, "url.deleted", r3.Payload)
UPDATE outbox SET published_at=now() WHERE id=r1.ID
OK
UPDATE outbox SET published_at=now() WHERE id=r2.ID
OK
UPDATE outbox SET published_at=now() WHERE id=r3.ID
OK
RabbitMQ Down — Publish Failure Path
C: tick
FetchUnpublished(limit=50)
rows [r4]
workerCh <- r4
row r4
Publish(ctx St, "url.created", r4.Payload)
← error: amqp publish url.created: EOF
log.Warn("outbox publish failed", "event_type", r4.EventType, "outbox_id", r4.ID, "error", err)
continue (skip MarkPublished)
r4.published_at remains NULL Next poll cycle will re-fetch r4. Consumer must deduplicate by event_id (analytics-service M4).
Next Poll — Retry r4
C: tick
FetchUnpublished(limit=50)
rows [r4] (published_at IS NULL)
workerCh <- r4
row r4 (retry)
Publish(ctx St, "url.created", r4.Payload) → OK
UPDATE outbox SET published_at=now() WHERE id=r4.ID
OK
Graceful Shutdown — SIGTERM Received
stk.Done() fires (appCancel() called)
ticker.Stop()
close(workerCh) → signals all workers to drain
range loop: channel closed → worker1 exits
range loop: channel closed → worker2 exits
range loop: channel closed → worker3 exits
wg.Wait() → blocks until all 3 workers finish in-flight publishes
log.Info("outbox coordinator stopped") return

```
Short code generation probability analysis:
  Base62 space: 62^7 = 3,521,614,606,208 ≈ 3.5 trillion codes
  After N URLs inserted, collision probability on single attempt:
    P(collision) ≈ N / 62^7
  At N = 1,000,000 (1M URLs):
    P(single collision) ≈ 2.84 × 10^-7 per attempt
    P(all 5 attempts collide) ≈ (2.84e-7)^5 ≈ 4.5 × 10^-33
  At N = 1,000,000,000 (1B URLs, 28.4% of space):
    P(single collision) ≈ 0.284
    P(all 5 attempts collide) ≈ 0.284^5 ≈ 0.18% → 503 error rate 0.18%
    (At this scale, a longer code or different strategy is warranted)
  For the scope of this project (< 1M URLs): 5 retries is more than sufficient.
```

10. Performance Targets

Operation	Target	How to Measure
`GET /:code` cache hit	< 5ms p99 under 100 concurrent requests	`wrk -t4 -c100 -d30s http://localhost:8081/<warm-code>` → Latency 99th < 5ms
`GET /:code` cache miss	< 20ms p99	Flush Redis key, then `wrk -t4 -c10 -d10s` (low concurrency to force misses)
`POST /shorten` (single tx: URL + outbox)	< 50ms p99	`wrk -t2 -c10 -d30s -s shorten.lua http://localhost:8081/shorten`
`GET /urls` page 1 (cold, 20 results)	< 10ms p99	`EXPLAIN ANALYZE SELECT ... LIMIT 21` must show Index Scan on idx_urls_user_id_created
Outbox poll-to-publish latency	< 5s	Create URL → poll RabbitMQ management API until queue depth increases; time delta
`DELETE /urls/:code` (tx + cache del)	< 20ms p99	Curl timing: `time curl -X DELETE ...`
Short code generation (single)	< 1ms	`go test -bench=BenchmarkGenerate -benchtime=10s` → ns/op < 1,000,000
Outbox coordinator memory	< 50MB steady-state	`docker stats url-service-1` after 10 minutes of load

EXPLAIN ANALYZE verification:

```bash
psql -h localhost -p 5432 -U urluser -d urldb -c "

EXPLAIN ANALYZE

SELECT id, short_code, original_url, user_id, created_at, expires_at, is_active

FROM urls

WHERE user_id = 'some-uuid'

ORDER BY created_at DESC, id DESC

LIMIT 21;

"

# Must show: Index Scan using idx_urls_user_id_created on urls

# Must NOT show: Seq Scan
```

Short code generation benchmark:

```go
func BenchmarkShortCodeGenerate(b *testing.B) {
    g := NewShortCodeGenerator()
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        g.Generate()
    }
}
// Expected: ~1-5 µs/op (crypto/rand + big.Int arithmetic)
```

Cursor-Based Pagination Query Plan

`GET /urls?after=\<uuid\>` — Index Scan Backward on `idx_urls_user_id_created`

OFFSET-based (ANTI-PATTERN)

```
SELECT ... FROM urls
WHERE user_id = $1
ORDER BY created_at DESC
LIMIT 20 OFFSET 1000
```

EXPLAIN ANALYZE shows Seq Scan / large rows_removed

Cost: O(N)
PostgreSQL must scan & discard 1000 rows before returning 20. At 100k rows = full table scan likely.

Replace with

Cursor-based (CORRECT)

```
-- Page 1 (afterID = "")
SELECT id, short_code, original_url,
       created_at, expires_at, is_active
FROM urls
WHERE user_id = $1
ORDER BY created_at DESC, id DESC
LIMIT 21

-- Page 2+ (afterID = "<uuid>")
SELECT id, short_code, original_url,
       created_at, expires_at, is_active
FROM urls
WHERE user_id = $1
  AND (created_at, id) < (
      SELECT created_at, id
      FROM urls WHERE id = $2   -- PK lookup
  )
ORDER BY created_at DESC, id DESC
LIMIT 21
```

EXPLAIN ANALYZE shows Index Scan Backward using idx_urls_user_id_created

Cost: O(log N)
B-tree seek directly to cursor position. Fetch next 21 rows from leaf chain. Scalable at any N.

implements

Algorithm Steps

Step 1 — Parse `afterID`

`afterID := r.URL.Query().Get("after")`

- `""` → first page
- `"<uuid>"` → subsequent page

afterID parsed

Step 2 — Execute SQL *(LIMIT = 21, not 20)*

If `afterID == ""` → Page-1 query else → Cursor query with subquery

Subquery `WHERE id = $2` is a **PRIMARY KEY** lookup. Row comparison `(created_at,id) < (val)` is handled via index.

rows fetched

len(rows) == 21

NO (≤20 rows)

YES (21 rows)

Step 4a — Next page exists

`nextCursor = rows[20].ID` (21st row) `return rows[:20]` (trim to 20)

Step 4b — No next page

`nextCursor = ""` (JSON null) `return rows` (all ≤ 20 rows)

Step 5 — Serialize response `go writeJSON(w, 200, urlListResponse{ URLs: items, NextCursor: &nextCursor, })`

validated by

EXPLAIN ANALYZE (expected output)

```
EXPLAIN ANALYZE
SELECT id, short_code, original_url,
       created_at, expires_at, is_active
FROM urls
WHERE user_id = 'some-uuid'
  AND (created_at, id) < ('2026-03-01 12:00:00+00', 'cursor-uuid')
ORDER BY created_at DESC, id DESC
LIMIT 21;
-- QUERY PLAN ------------------------------------------------------------
Limit  (cost=0.43..12.87 rows=21 width=120)
       (actual time=0.041..0.182 rows=21 loops=1)
  ->  Index Scan Backward using idx_urls_user_id_created on urls
        (cost=0.43..1248.56 rows=2103 width=120)
        (actual time=0.038..0.174 rows=21 loops=1)
      Index Cond: (user_id = 'some-uuid'::uuid)
      Filter: ((created_at, id) < ('2026-03-01...', 'cursor-uuid'))
      Rows Removed by Filter: 0
Planning Time:  0.120 ms
Execution Time: 0.210 ms    ← O(log N); scales to millions
```

uses

Index: idx_urls_user_id_created

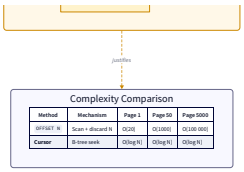
```
CREATE INDEX idx_urls_user_id_created
    ON urls(user_id, created_at DESC);
```

B-Tree Leaf Chain (sorted)

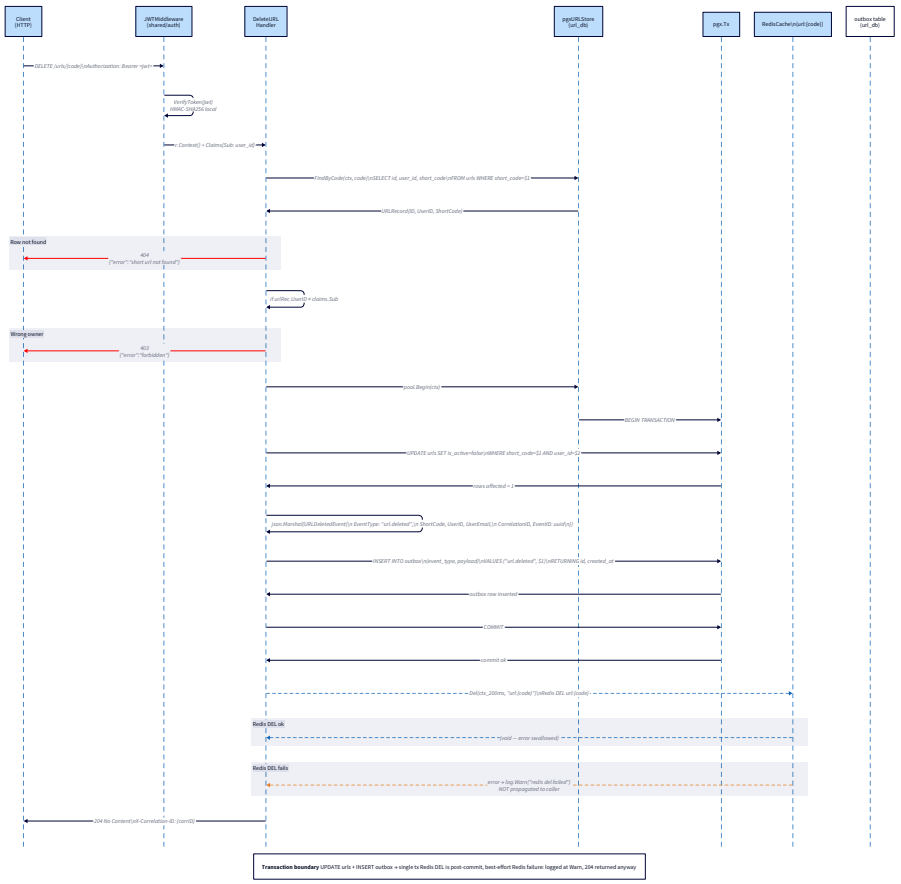
- (user_A, 2026-03-02, id-7)
- (user_A, 2026-03-01, id-3)
- (user_A, 2026-02-28, id-9)
- (user_B, 2026-03-02, id-1)

seek here

Cursor seek:
B-tree descends to O(log N). Then follows leaf chain.




```
Performance budget for GET /:code cache hit path:
  ┌──────────────────────────────────────────────────────┐
  | Network recv + HTTP parse:          ~0.1ms           |
  | CorrelationIDMiddleware:            <0.1ms           |
  | RedisCache.Get (50ms ctx):          ~0.5ms (LAN Redis)  | ← dominates
  | JSON unmarshal CachedURL:           <0.1ms           |
  | is_active + expiry check:           <0.01ms          |
  | http.Redirect (write headers):      ~0.1ms           |
  | Network send:                       ~0.1ms           |
  | TOTAL:                              ~1ms typical     |
  |                                     <5ms p99 ✓       |
  └──────────────────────────────────────────────────────┘
Performance budget for GET /:code cache miss path:
  ┌──────────────────────────────────────────────────────┐
  | RedisCache.Get (miss):              ~0.5ms           |
  | pgxpool.Acquire:                    ~0.1ms (warm pool)  |
  | pgx QueryRow (idx_urls_short_code): ~1-3ms (LAN PG)    | ← dominates
  | RedisCache.Set:                     ~0.5ms (async-ish)  |
  | Begin outbox tx + Insert + Commit:  ~2-5ms           |
  | http.Redirect:                      ~0.1ms           |
  | TOTAL:                              ~5-10ms typical  |
  |                                     <20ms p99 ✓      |
  └──────────────────────────────────────────────────────┘
```



```
go.mod additions for url-service (M3):
  (from M1): github.com/jackc/pgx/v5 v5.6.0
             github.com/redis/go-redis/v9 v9.6.0
             github.com/rabbitmq/amqp091-go v1.10.0
  (from M2): github.com/yourhandle/url-shortener/shared/auth
             github.com/golang-jwt/jwt/v5 v5.2.1
  (new M3):  math/big — stdlib, no dependency
             crypto/rand — stdlib, no dependency
             github.com/alicebob/miniredis/v2 v2.32.0  (test only)
  All require in go.mod:
    require (
        github.com/yourhandle/url-shortener/shared/auth v0.0.0
        github.com/yourhandle/url-shortener/shared/events v0.0.0
        github.com/yourhandle/url-shortener/shared/logger v0.0.0
        github.com/jackc/pgx/v5 v5.6.0
        github.com/redis/go-redis/v9 v9.6.0
        github.com/rabbitmq/amqp091-go v1.10.0
        github.com/golang-jwt/jwt/v5 v5.2.1
        github.com/alicebob/miniredis/v2 v2.32.0
    )
  docker-compose.yml additions for url-service environment:
    JWT_SECRET: "change-this-in-production-minimum-32-chars"
    SHORT_URL_BASE: "http://localhost:8080"   ← or gateway URL in M5
    IP_HASH_SALT: "change-this-random-salt"
```

11. Anti-Pattern Guard

The following checks must pass before this module is considered complete:

```bash
# 1. No shared DB: url-service must ONLY connect to url_db

grep -r "analytics_db\|user_db\|notification_db" services/url-service/ || echo "PASS"

# 2. No synchronous analytics call

grep -r "analytics-service\|analytics_service\|8082" services/url-service/ || echo "PASS"

# 3. crypto/rand only (no math/rand in production code)

grep -r "math/rand" services/url-service/*.go | grep -v "_test.go" || echo "PASS"

# 4. No unbounded Redis TTL

# Verify Set always calls with non-zero TTL:

grep -A5 "client.Set" services/url-service/cache.go

# Must show TTL parameter is never 0 or redis.KeepTTL except the 1s guard case

# 5. Outbox and URL write in same transaction (POST /shorten)

# Review handler.go: BEGIN → InsertWithURL (urls + outbox) → COMMIT

# Both tables must appear inside the same pool.Begin() ... tx.Commit() scope
```

Analytics Service: Click Ingestion + Stats API

Module ID: `url-shortener-m4`

1. Module Charter

This module implements the Analytics Service as a self-contained read-model consumer. It owns the `clicks`, `milestones`, and `processed_events` tables in its dedicated PostgreSQL instance (`analytics_db`), subscribes to the `analytics.clicks` queue on RabbitMQ, processes `URLClickedEvent` messages with at-least-once semantics, deduplicates by `event_id`, inserts anonymized click records, checks milestone thresholds (10, 100, 1000 clicks per short code), publishes `MilestoneReachedEvent` back to RabbitMQ when thresholds are crossed, and exposes two public (unauthenticated) HTTP endpoints for click statistics. This module does **not** call the URL Service to validate `short_code` — it trusts the event. It does **not** store user PII — IP addresses are stored as SHA-256 hashes, never raw. It does **not** issue or verify JWTs — both `/stats` endpoints are public. It does **not** write to any database other than `analytics_db`. It does **not** implement the outbox pattern for `MilestoneReachedEvent` — publish failure is logged and the milestone row remains committed; the event may be re-triggered if the next click re-checks the threshold (thresholds are idempotent via `milestones` table). It does **not** implement a worker pool — a single sequential consumer goroutine with AMQP `prefetch=1` provides serialized processing without race conditions. **Upstream dependencies:** `shared/events` (event struct definitions), `shared/logger` (structured JSON logger), `NewRabbitMQConn` and `DeclareAnalyticsQueue` (from M1 `analytics-service/rabbitmq.go`), `analytics_db` PostgreSQL instance, `NewDBPool` (from M1). **Downstream consumers:** `notification-service` consumes `MilestoneReachedEvent` from `notifications.events` queue (bound to routing key `milestone.reached`). The analytics service publishes this event directly on the AMQP channel — not via the outbox — because the analytics DB is the authoritative source and the milestone row is already committed before publish is attempted. **Invariants that must always hold:**

- A `URLClickedEvent` with a previously seen `event_id` is **silently acknowledged and discarded** — click count never increments more than once per event.
- Click insert, processed_event insert, and milestone insert (when triggered) all occur in **one database transaction** — no partial state.
- A malformed/unparseable AMQP message body is **acknowledged** (not nacked) to prevent infinite requeue loops; it is logged as a poison message.
- The `/health` endpoint returns `200 {"status":"ok"}` even when the RabbitMQ consumer is temporarily paused due to connection drop.
- `MilestoneReachedEvent` publish failure never causes the consumer to crash or nack the originating click message — the milestone row is already durable.
- `ip_hash` is always `SHA-256(raw_ip + salt)` — the raw IP never enters the database.

2. File Structure

Create files in the numbered order below. Shared packages exist from M1/M2/M3; only analytics-service files are new here.

```
url-shortener/
│
├── shared/                                ← unchanged from prior milestones
│   ├── events/events.go
│   ├── auth/auth.go + middleware.go
│   └── logger/logger.go
│
└── services/
    └── analytics-service/
        ├── (from M1, extended)
        │   ├── go.mod                     ← add dependencies (see §6.3 / §10)
        │   ├── main.go                    ← extend: wire consumer, start goroutine
        │   ├── config.go                  ← extend: add IPHashSalt, MilestoneThresholds
        │   ├── db.go                      ← unchanged from M1 (NewDBPool reused)
        │   ├── rabbitmq.go                ← extend: add DeclareAnalyticsQueue (already in M1)
        │   └── health.go                  ← unchanged from M1
        │
        ├── 1.  migration.sql              ← clicks, milestones, processed_events + indexes
        ├── 2.  store.go                   ← ClickRepository, MilestoneRepository, DeduplicationRepository interfaces + pgx
implementations
        ├── 3.  consumer.go                ← ClickConsumer struct, Run(), message loop, panic recovery
        ├── 4.  milestone.go               ← MilestoneChecker, CheckAndPublish, threshold constants
        ├── 5.  publisher.go               ← AnalyticsPublisher interface + amqpAnalyticsPublisher (mirrors url-
service/publisher.go pattern)
        ├── 6.  handler.go                 ← StatsHandler, TimelineHandler, route registration helper
        ├── 7.  errors.go                  ← writeError, writeJSON helpers, sentinel errors
        ├── 8.  haship.go                  ← hashIP(remoteAddr, salt) string
        └── 9.  analytics_test.go          ← unit + integration tests
```

3. Complete Data Model

3.1 PostgreSQL Schema (`migration.sql`)

```sql
-- ── clicks table ──────────────────────────────────────────────────────────

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(raw_ip + salt); never raw IP

    user_agent  TEXT        NOT NULL DEFAULT '',

    referer     TEXT        NULL        -- NULL when no Referer header was present

);

-- Primary query path: aggregate clicks per code in time windows.

-- date_trunc bucketing and COUNT(*) both benefit from this index.

CREATE INDEX IF NOT EXISTS idx_clicks_short_code_time

    ON clicks(short_code, clicked_at DESC);

-- Top-referers query: partial index omits rows with no referer (saves space).

CREATE INDEX IF NOT EXISTS idx_clicks_referer

    ON clicks(short_code, referer)

    WHERE referer IS NOT NULL;

-- ── milestones table ──────────────────────────────────────────────────────

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)        -- exactly one row per (code, threshold)

);

-- Lookup: has this code already crossed a given milestone?

CREATE UNIQUE INDEX IF NOT EXISTS idx_milestones_code_milestone

    ON milestones(short_code, milestone);

-- ── processed_events table (deduplication) ────────────────────────────────

CREATE TABLE IF NOT EXISTS processed_events (

    event_id    TEXT        PRIMARY KEY,  -- events.BaseEvent.EventID (UUID v4 string)

    processed_at TIMESTAMPTZ NOT NULL DEFAULT now()

);

-- No additional index needed — PRIMARY KEY gives the O(log n) lookup we need.
```

```
-- Retention: rows here grow indefinitely for this project. A production system

-- would partition by processed_at and drop old partitions after TTL.
```

Column rationale:

Column	Type	Constraint	Rationale
`clicks.id`	UUID	PK	Unique row identity; not exposed in API
`clicks.short_code`	TEXT	NOT NULL	Denormalized from event; analytics DB has no FK to url_db (no cross-DB joins)
`clicks.clicked_at`	TIMESTAMPTZ	NOT NULL	From `URLClickedEvent.ClickedAt`; used for time-window aggregation and `date_trunc`
`clicks.ip_hash`	TEXT	NOT NULL	SHA-256(IP+salt) — 64-char hex string; GDPR-safe; salted to prevent rainbow table lookup
`clicks.user_agent`	TEXT	NOT NULL DEFAULT ''	From event; empty string when absent (not NULL — avoids NULL handling complexity in aggregations)
`clicks.referer`	TEXT	NULL	NULL when absent; partial index on non-NULL rows for top-referers query
`milestones.short_code`	TEXT	NOT NULL	Denormalized; no FK
`milestones.milestone`	INT	NOT NULL	Must be one of {10, 100, 1000}; enforced in application layer, not DB constraint
`milestones.UNIQUE(short_code, milestone)`	—	—	Prevents double-recording the same milestone; INSERT ... ON CONFLICT DO NOTHING is idempotent
`processed_events.event_id`	TEXT	PK	EventID from BaseEvent; TEXT not UUID to avoid parsing overhead on lookup; PRIMARY KEY gives unique constraint
`processed_events.processed_at`	TIMESTAMPTZ	NOT NULL DEFAULT now()	Audit trail; useful for future TTL-based cleanup

3.2 Go Structs

```go
// store.go

package analytics

import "time"

// ClickRecord maps to one row in the clicks table.

// Populated from URLClickedEvent fields.

type ClickRecord struct {

    ID        string    // UUID, generated by DB (gen_random_uuid())

    ShortCode string

    ClickedAt time.Time

    IPHash    string    // SHA-256(raw_ip + salt); 64-char hex

    UserAgent string

    Referer   string    // empty string when absent (not nil pointer)

}

// MilestoneRecord maps to one row in the milestones table.

type MilestoneRecord struct {

    ID          string

    ShortCode   string

    Milestone   int       // 10 | 100 | 1000

    TriggeredAt time.Time

}

// StatsResult is the aggregated response for GET /stats/:code.

// Built from multiple DB queries in the handler.

type StatsResult struct {

    ShortCode      string

    TotalClicks    int64

    ClicksLast24h  int64

    ClicksLast7d   int64

    TopReferers    []RefererCount

}

// RefererCount holds one entry in the top-referers list.

type RefererCount struct {

    Referer string `json:"referer"`

    Count   int64  `json:"count"`

}

// TimelinePoint holds one bucket in the timeline response.
```

```
type TimelinePoint struct {

    Period string `json:"period"` // RFC3339 of the bucket start

    Clicks int64  `json:"clicks"`

}
```

GO

```go
// config.go (extended from M1)

package analytics

import (

    "fmt"

    "os"

    "time"

)

type Config struct {

    DatabaseURL        string

    RabbitMQURL        string

    Port               string

    ServiceName        string

    IPHashSalt         string        // read from IP_HASH_SALT env var; empty string is acceptable

    AMQPPrefetchCount  int           // always 1 for this service (not configurable — fixed per spec)

    OutboxPollInterval time.Duration // unused in analytics; kept for config struct symmetry

}

func loadConfig() (*Config, error) {

    cfg := &Config{

        DatabaseURL:       os.Getenv("DATABASE_URL"),

        RabbitMQURL:       os.Getenv("RABBITMQ_URL"),

        Port:              envOrDefault("PORT", "8080"),

        ServiceName:       "analytics-service",

        IPHashSalt:        os.Getenv("IP_HASH_SALT"),

        AMQPPrefetchCount: 1,

    }

    if cfg.DatabaseURL == "" {

        return nil, fmt.Errorf("DATABASE_URL is required")

    }

    if cfg.RabbitMQURL == "" {

        return nil, fmt.Errorf("RABBITMQ_URL is required")

    }

    return cfg, nil

}

func envOrDefault(key, def string) string {

    if v := os.Getenv(key); v != "" {

        return v

    }
```

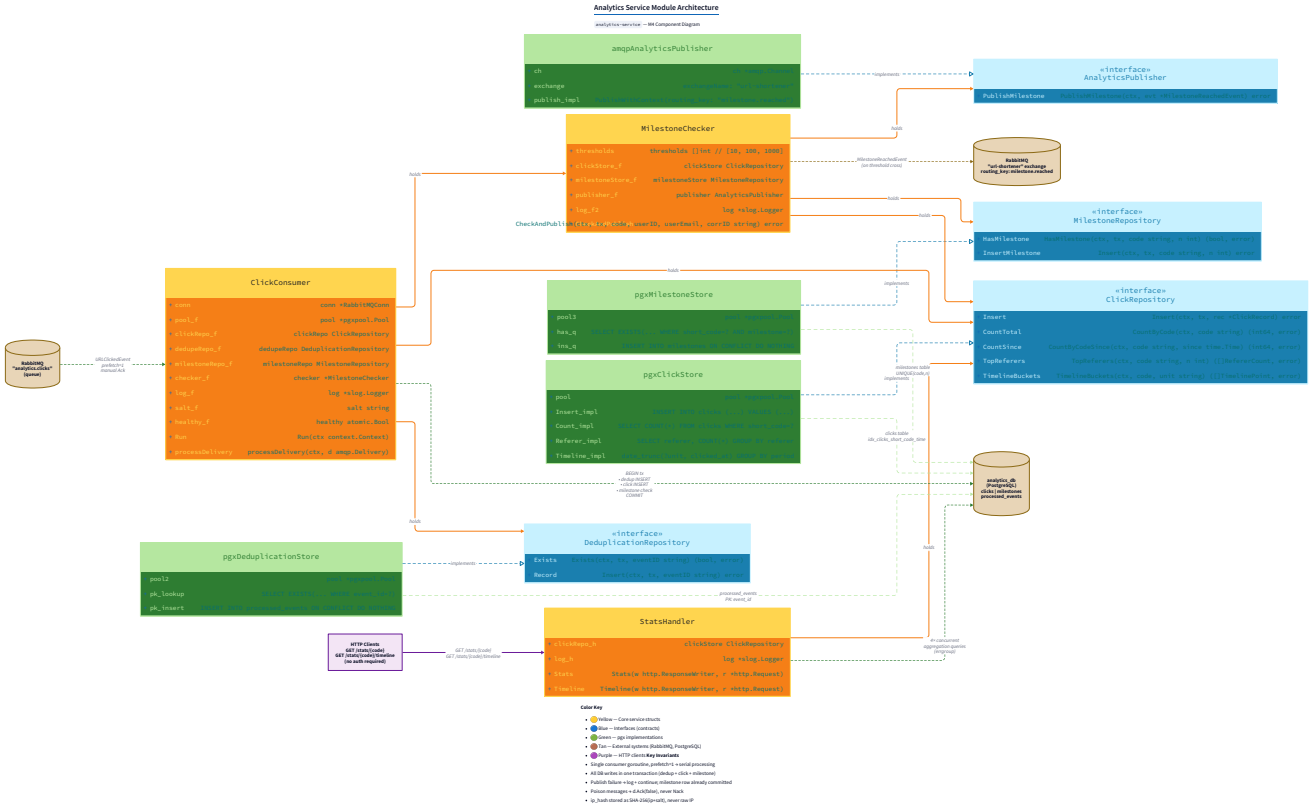
```
    return def
}
```

```go
// handler.go — HTTP request/response types

package analytics

// statsResponse is the JSON shape for 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   []RefererCount `json:"top_referers"` // always array, never null
}

// timelineResponse is the JSON shape for GET /stats/:code/timeline.
type timelineResponse struct {
    ShortCode string          `json:"short_code"`
    Interval  string          `json:"interval"`  // "day" | "hour" echoed back
    Points    []TimelinePoint `json:"points"`    // always array, never null
}
```

3.3 Repository Interfaces

```go
// store.go

// ClickRepository abstracts all DB operations on the clicks table.
type ClickRepository interface {
    // Insert creates a new click row. Called within a transaction.
    // rec.ID is populated by the DB (gen_random_uuid()); caller passes zero value.
    Insert(ctx context.Context, tx pgx.Tx, rec *ClickRecord) error
    // CountByCode returns total click count for a short_code.
    CountByCode(ctx context.Context, shortCode string) (int64, error)
    // CountByCodeSince returns click count for a short_code after the given time.
    CountByCodeSince(ctx context.Context, shortCode string, since time.Time) (int64, error)
    // TopReferers returns the top N referers for a short_code by count, descending.
    // Only non-NULL referer rows are considered (partial index).
    // n is always 5 in this project.
    TopReferers(ctx context.Context, shortCode string, n int) ([]RefererCount, error)
    // TimelineBuckets returns click counts bucketed by the given PostgreSQL truncation unit.
    // truncUnit is "day" or "hour" — validated by caller before reaching here.
    // Returns buckets ordered by period ASC.
    TimelineBuckets(ctx context.Context, shortCode string, truncUnit string) ([]TimelinePoint, error)
}
// MilestoneRepository abstracts all DB operations on the milestones table.
type MilestoneRepository interface {
    // HasMilestone returns true if the given (shortCode, milestone) row exists.
    HasMilestone(ctx context.Context, tx pgx.Tx, shortCode string, milestone int) (bool, error)
    // Insert records that a milestone was reached. Uses ON CONFLICT DO NOTHING for idempotency.
    // Called within a transaction.
    Insert(ctx context.Context, tx pgx.Tx, shortCode string, milestone int) error
}
// DeduplicationRepository abstracts all DB operations on the processed_events table.
type DeduplicationRepository interface {
    // Exists checks if the given event_id has already been processed.
    // Called within a transaction (holds a row-level lock on the insert for the duration).
    Exists(ctx context.Context, tx pgx.Tx, eventID string) (bool, error)
    // Insert records an event_id as processed. Called within the same transaction as Exists.
    // On conflict (race between two consumer instances): ON CONFLICT DO NOTHING.
    Insert(ctx context.Context, tx pgx.Tx, eventID string) error
```

```
}
```




```
Data ownership and flow:
RabbitMQ                analytics-service                analytics_db
queue:                  ┌──────────────────────────┐     ┌──────────────────┐
"analytics.clicks"      │  ClickConsumer goroutine │     │  clicks          │
       │                │                          │     │  milestones      │
       │ AMQP Deliver   │  1. decode JSON          │     │  processed_events│
       └──────────────► │  2. dedup check (tx)     │     └──────────────────┘
                        │  3. insert click (same tx)│           ▲
                        │  4. milestone check (tx) │           │
                        │  5. commit tx            │───────────┘
                        │  6. maybe publish        │
                        │     MilestoneReachedEvent│
                        │  7. d.Ack()              │
                        │                          │
HTTP (public)           │  StatsHandler            │
GET /stats/:code ──────►│    CountByCode           │
GET /stats/.../timeline►│    CountByCodeSince      │
                        │    TopReferers           │
                        │    TimelineBuckets       │
                        └──────────────────────────┘
                                    │
                                RabbitMQ publish
                                routing key:
                                "milestone.reached"
                                    │
                                    ▼
                             notifications.events queue
                             (notification-service M5)
```

4. Interface Contracts

4.1 `ClickRepository` — `pgxClickStore` Implementation

```go
// store.go

type pgxClickStore struct {

    pool *pgxpool.Pool

}

func NewClickStore(pool *pgxpool.Pool) ClickRepository {

    return &pgxClickStore{pool: pool}

}
```

`Insert(ctx context.Context, tx pgx.Tx, rec *ClickRecord) error`

```
SQL (executed on tx):
  INSERT INTO clicks (short_code, clicked_at, ip_hash, user_agent, referer)
  VALUES ($1, $2, $3, $4, $5)
Parameters:
  $1 rec.ShortCode
  $2 rec.ClickedAt    (time.Time; pgx maps to TIMESTAMPTZ)
  $3 rec.IPHash       (64-char hex string)
  $4 rec.UserAgent    (empty string acceptable; NOT NULL DEFAULT '')
  $5 rec.Referer      (empty string → NULL in DB via nilable helper; see note)
Referer NULL handling:
  Pass nil to pgx when rec.Referer == "" to store SQL NULL (allows partial index).
  Use pgtype.Text{String: rec.Referer, Valid: rec.Referer != ""} or convert to *string.
On success: return nil
On DB error: return fmt.Errorf("insert click: %w", err)
```

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

```
SQL:
  SELECT COUNT(*) FROM clicks WHERE short_code = $1
Uses idx_clicks_short_code_time (leading column = short_code).
On success: return count, nil
On DB error: return 0, fmt.Errorf("count clicks by code: %w", err)
```

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

```
SQL:
  SELECT COUNT(*) FROM clicks
  WHERE short_code = $1 AND clicked_at >= $2
Parameters: shortCode, since (time.Time)
Since values in use:
  24h window: time.Now().UTC().Add(-24 * time.Hour)
   7d window: time.Now().UTC().Add(-7 * 24 * time.Hour)
Uses idx_clicks_short_code_time (composite: short_code + clicked_at DESC).
On success: return count, nil
On DB error: return 0, fmt.Errorf("count clicks since: %w", err)
```

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

```
SQL:
  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
Parameters: shortCode, n (always 5)
Uses idx_clicks_referer (partial index WHERE referer IS NOT NULL).
On 0 rows: return empty []RefererCount{}, nil — NOT an error
On DB error: return nil, fmt.Errorf("top referers: %w", err)
Note: Caller is responsible for initializing []RefererCount{} (not nil) so
      JSON serializes as [] rather than null.
```

`TimelineBuckets(ctx context.Context, shortCode string, truncUnit string) ([]TimelinePoint, error)`

```
SQL:
  SELECT
    date_trunc($1, clicked_at AT TIME ZONE 'UTC') AS period,
    COUNT(*) AS clicks
  FROM clicks
  WHERE short_code = $2
  GROUP BY period
  ORDER BY period ASC
Parameters:
  $1 truncUnit ("day" or "hour") — caller validates before passing
  $2 shortCode
On success: scan each row into TimelinePoint{Period: period.Format(time.RFC3339), Clicks: count}
  period is a time.Time from PostgreSQL; format with .UTC().Format(time.RFC3339)
On 0 rows: return empty []TimelinePoint{}, nil
On DB error: return nil, fmt.Errorf("timeline buckets: %w", err)
```

4.2 `MilestoneRepository` — `pgxMilestoneStore` Implementation

GO

```go
type pgxMilestoneStore struct {

    pool *pgxpool.Pool

}

func NewMilestoneStore(pool *pgxpool.Pool) MilestoneRepository {

    return &pgxMilestoneStore{pool: pool}

}
```

`HasMilestone(ctx context.Context, tx pgx.Tx, shortCode string, milestone int) (bool, error)`

```
SQL (on tx):
  SELECT EXISTS(
    SELECT 1 FROM milestones
    WHERE short_code = $1 AND milestone = $2
  )
Parameters: shortCode, milestone
On success: return bool, nil
On DB error: return false, fmt.Errorf("has milestone: %w", err)
```

`Insert(ctx context.Context, tx pgx.Tx, shortCode string, milestone int) error`

```
SQL (on tx):
  INSERT INTO milestones (short_code, milestone)
  VALUES ($1, $2)
  ON CONFLICT (short_code, milestone) DO NOTHING
Parameters: shortCode, milestone
ON CONFLICT DO NOTHING: safe for concurrent consumers or replay scenarios.
On success (insert or no-op): return nil
On DB error (non-conflict): return fmt.Errorf("insert milestone: %w", err)
```

4.3 `DeduplicationRepository` — `pgxDeduplicationStore` Implementation

```go
type pgxDeduplicationStore struct {

    pool *pgxpool.Pool

}

func NewDeduplicationStore(pool *pgxpool.Pool) DeduplicationRepository {

    return &pgxDeduplicationStore{pool: pool}

}
```

`Exists(ctx context.Context, tx pgx.Tx, eventID string) (bool, error)`

```
SQL (on tx):
  SELECT EXISTS(SELECT 1 FROM processed_events WHERE event_id = $1)
Parameters: eventID
On success: return bool, nil
On DB error: return false, fmt.Errorf("dedup exists: %w", err)
Note: Running inside the same transaction as Insert below ensures
      that a concurrent consumer seeing the same event_id will either:
      (a) see Exists=true (our tx already committed), or
      (b) hit a PRIMARY KEY violation on Insert (our tx committed first).
      Single consumer with prefetch=1 makes this moot in practice,
      but the tx boundary is required for correctness.
```

`Insert(ctx context.Context, tx pgx.Tx, eventID string) error`

```
SQL (on tx):
  INSERT INTO processed_events (event_id)
  VALUES ($1)
  ON CONFLICT (event_id) DO NOTHING
Parameters: eventID
On success (insert or no-op conflict): return nil
On DB error (non-conflict): return fmt.Errorf("dedup insert: %w", err)
```

4.4 `AnalyticsPublisher` Interface

```go
// publisher.go

type AnalyticsPublisher interface {

    // PublishMilestone sends a MilestoneReachedEvent to the "url-shortener" exchange

    // with routing key "milestone.reached".

    // Returns error if publish fails; caller logs and continues.

    PublishMilestone(ctx context.Context, evt *events.MilestoneReachedEvent) error

}

type amqpAnalyticsPublisher struct {

    ch           *amqp.Channel

    exchangeName string

    log          *slog.Logger

}

func NewAnalyticsPublisher(ch *amqp.Channel, log *slog.Logger) AnalyticsPublisher {

    return &amqpAnalyticsPublisher{ch: ch, exchangeName: "url-shortener", log: log}

}
```

`PublishMilestone(ctx context.Context, evt *events.MilestoneReachedEvent) error`

```go
func (p *amqpAnalyticsPublisher) PublishMilestone(ctx context.Context, evt *events.MilestoneReachedEvent) error {
    body, err := json.Marshal(evt)
    if err != nil {
        return fmt.Errorf("marshal MilestoneReachedEvent: %w", err)
    }
    err = p.ch.PublishWithContext(ctx,
        p.exchangeName,          // exchange
        events.EventTypeMilestoneReached, // routing key: "milestone.reached"
        false,                   // mandatory
        false,                   // immediate
        amqp.Publishing{
            ContentType:  "application/json",
            DeliveryMode: amqp.Persistent,
            Body:         body,
        },
    )
    if err != nil {
        return fmt.Errorf("amqp publish milestone: %w", err)
    }
    return nil
}
```

Note on channel safety: `amqp091-go` channels are not safe for concurrent use. Since the consumer goroutine is the only goroutine calling `PublishMilestone`, no mutex is required here (unlike the url-service 3-worker case). If HTTP handlers ever need to publish, a mutex must be added.

4.5 `MilestoneChecker`

```go
// milestone.go

const (

    MilestoneThreshold10   = 10

    MilestoneThreshold100  = 100

    MilestoneThreshold1000 = 1000

)

var MilestoneThresholds = []int{MilestoneThreshold10, MilestoneThreshold100, MilestoneThreshold1000}

type MilestoneChecker struct {

    clickStore     ClickRepository

    milestoneStore MilestoneRepository

    publisher      AnalyticsPublisher

    log            *slog.Logger

}

func NewMilestoneChecker(

    clickStore ClickRepository,

    milestoneStore MilestoneRepository,

    publisher AnalyticsPublisher,

    log *slog.Logger,

) *MilestoneChecker

// CheckAndPublish is called inline in the consumer goroutine after a click is inserted.

// It reads total_clicks for the short_code (within the same tx to get the post-insert count),

// checks each threshold, and for any newly crossed threshold: inserts a milestone row

// and publishes MilestoneReachedEvent.

//

// Parameters:

//   ctx       - request context with deadline

//   tx        - the active transaction (click + dedup already inserted within it)

//   shortCode - the code that just received a click

//   userID    - from URLClickedEvent (may be empty if not in event; use "unknown")

//   userEmail - from URLClickedEvent (may be empty; use "" for MilestoneReachedEvent)

//   corrID    - correlation ID for event propagation

//

// Returns: error only for fatal DB errors — publish failures are logged and swallowed.

func (m *MilestoneChecker) CheckAndPublish(

    ctx context.Context,
```

```
    tx pgx.Tx,

    shortCode, userID, userEmail, corrID string,

) error
```

5. Algorithm Specification

5.1 Consumer Startup and AMQP Prefetch

```
// consumer.go — called in main.go
type ClickConsumer struct {
    conn       *RabbitMQConn        // from M1 factory
    pool       *pgxpool.Pool
    clickStore ClickRepository
    milestoneStore MilestoneRepository
    dedupStore DeduplicationRepository
    checker    *MilestoneChecker
    log        *slog.Logger
    salt       string               // IP_HASH_SALT from config
    healthy    atomic.Bool          // true when consumer loop is running
}
func NewClickConsumer(
    conn *RabbitMQConn,
    pool *pgxpool.Pool,
    clickStore ClickRepository,
    milestoneStore MilestoneRepository,
    dedupStore DeduplicationRepository,
    checker *MilestoneChecker,
    log *slog.Logger,
    salt string,
) *ClickConsumer
// Run starts the consumer loop. Blocks until ctx is Done.
// Called with: go consumer.Run(ctx) in main.go.
func (c *ClickConsumer) Run(ctx context.Context)
```

`Run(ctx context.Context)` Algorithm:

```
Step 1: Set quality-of-service (prefetch)
  err := c.conn.Channel.Qos(
      1,     // prefetchCount: process one message at a time
      0,     // prefetchSize: no size limit
      false, // global: apply per-consumer not per-channel
  )
  if err != nil:
      c.log.Error("set qos failed", "error", err)
      os.Exit(1)  // fatal: cannot guarantee serial processing without prefetch
Step 2: Register consumer
  msgs, err := c.conn.Channel.Consume(
      "analytics.clicks", // queue (declared in M1 DeclareAnalyticsQueue)
      "",                 // consumer tag: auto-generated
      false,              // autoAck: false — we ack manually after processing
      false,              // exclusive: false
      false,              // noLocal: false
      false,              // noWait: false
      nil,                // args
  )
  if err != nil:
      c.log.Error("register consumer failed", "error", err)
      os.Exit(1)  // fatal: cannot function without consumer registration
Step 3: Mark healthy
  c.healthy.Store(true)
  c.log.Info("consumer started", "queue", "analytics.clicks")
Step 4: Message loop
  for:
      select:
      case <-ctx.Done():
          c.healthy.Store(false)
          c.log.Info("consumer stopped")
          return
      case d, ok := <-msgs:
          if !ok:
              // Channel closed (RabbitMQ connection dropped)
              c.healthy.Store(false)
              c.log.Warn("amqp channel closed, consumer paused")
              // Do NOT os.Exit — /health must remain 200
              // Block until ctx is done (reconnect logic is future work; in-scope for M5)
              <-ctx.Done()
              return
          c.processDelivery(ctx, d)
```

`processDelivery(ctx context.Context, d amqp.Delivery)` Algorithm:

```
Step 1: Panic recovery (poison message guard)
  defer func() {
      if r := recover(); r != nil:
          c.log.Error("consumer panic recovered",
              "panic", fmt.Sprintf("%v", r),
              "body_preview", truncate(string(d.Body), 200),
          )
          d.Ack(false)  // ack to remove from queue; do not requeue a panicking message
  }()
Step 2: Decode JSON
  var evt events.URLClickedEvent
  if err := json.Unmarshal(d.Body, &evt); err != nil:
      c.log.Error("poison message: JSON unmarshal failed",
          "error", err,
          "body_preview", truncate(string(d.Body), 200),
      )
      d.Ack(false)  // ack (NOT nack) to drain poison messages without requeue
      return
Step 3: Validate required fields
  if evt.EventID == "" || evt.ShortCode == "":
      c.log.Error("poison message: missing required fields",
          "event_id", evt.EventID,
          "short_code", evt.ShortCode,
      )
      d.Ack(false)
      return
Step 4: Open transaction
  tx, err := c.pool.Begin(ctx)
  if err != nil:
      c.log.Error("begin tx failed", "error", err, "event_id", evt.EventID)
      d.Nack(false, true)  // nack + requeue — DB might recover
      return
Step 5: Idempotency check
  exists, err := c.dedupStore.Exists(ctx, tx, evt.EventID)
  if err != nil:
      tx.Rollback(ctx)
      c.log.Error("dedup exists check failed", "error", err, "event_id", evt.EventID)
      d.Nack(false, true)
      return
  if exists:
      tx.Rollback(ctx)
      c.log.Info("duplicate event skipped", "event_id", evt.EventID)
      d.Ack(false)
      return
Step 6: Record event_id as processed
  if err := c.dedupStore.Insert(ctx, tx, evt.EventID); err != nil:
      tx.Rollback(ctx)
      c.log.Error("dedup insert failed", "error", err, "event_id", evt.EventID)
      d.Nack(false, true)
      return
Step 7: Build and insert click record
  rec := &ClickRecord{
      ShortCode: evt.ShortCode,
      ClickedAt: evt.ClickedAt,
      IPHash:    evt.IPHash,   // already hashed by url-service; do NOT re-hash
      UserAgent: evt.UserAgent,
      Referer:   evt.Referer,
  }
  if err := c.clickStore.Insert(ctx, tx, rec); err != nil:
      tx.Rollback(ctx)
      c.log.Error("click insert failed", "error", err, "event_id", evt.EventID)
      d.Nack(false, true)
      return
Step 8: Milestone check (inline, same tx)
  corrID := evt.CorrelationID
  if corrID == "" { corrID = evt.EventID }
  if err := c.checker.CheckAndPublish(ctx, tx, evt.ShortCode, "", "", corrID); err != nil:
      tx.Rollback(ctx)
      c.log.Error("milestone check failed", "error", err, "event_id", evt.EventID)
      d.Nack(false, true)
      return
Step 9: Commit transaction
  if err := tx.Commit(ctx); err != nil:
```

```
      c.log.Error("commit failed", "error", err, "event_id", evt.EventID)
      d.Nack(false, true)
      return
Step 10: Acknowledge message
  d.Ack(false)
  c.log.Info("click processed",
      "event_id", evt.EventID,
      "short_code", evt.ShortCode,
      "correlation_id", corrID,
  )
```

5.2 `MilestoneChecker.CheckAndPublish` Algorithm

```
Input:
  ctx       context.Context
  tx        pgx.Tx          (click + dedup already inserted, tx not yet committed)
  shortCode string
  userID    string          (from URLClickedEvent; may be empty — use "" in event payload)
  userEmail string          (from URLClickedEvent; may be empty)
  corrID    string
Step 1: Count total clicks for this shortCode within the transaction
  // Use pool directly (not tx) to get committed count + current tx count.
  // IMPORTANT: Use tx.QueryRow so the count includes the click we just inserted
  //            (the INSERT is visible within this transaction but not yet committed).
  var totalClicks int64
  row := tx.QueryRow(ctx,
      "SELECT COUNT(*) FROM clicks WHERE short_code = $1",
      shortCode,
  )
  if err := row.Scan(&totalClicks); err != nil:
      return fmt.Errorf("count clicks for milestone: %w", err)
Step 2: Check each threshold in ascending order
  for _, threshold := range MilestoneThresholds:  // [10, 100, 1000]
      if totalClicks < int64(threshold):
          continue  // not yet reached
      // Has it already been recorded?
      alreadyRecorded, err := m.milestoneStore.HasMilestone(ctx, tx, shortCode, threshold)
      if err != nil:
          return fmt.Errorf("has milestone %d: %w", threshold, err)
      if alreadyRecorded:
          continue  // already crossed and recorded
      // New milestone! Record it.
      if err := m.milestoneStore.Insert(ctx, tx, shortCode, threshold); err != nil:
          return fmt.Errorf("insert milestone %d: %w", threshold, err)
      // Publish MilestoneReachedEvent (outside tx — publish after milestone committed is
      // safer; but here we publish before commit. If commit fails, we have a phantom event.
      // This is acceptable: milestone.reached is informational; notification-service
      // deduplication via processed_events in M5 handles replays.)
      //
      // Build and publish after tx.Commit() in the caller would be ideal but requires
      // returning the milestone data from this function. For simplicity, publish here
      // before commit. If publish fails, we still commit (milestone row is durable).
      evt := &events.MilestoneReachedEvent{
          BaseEvent: events.BaseEvent{
              EventType:     events.EventTypeMilestoneReached,
              OccurredAt:    time.Now().UTC(),
              CorrelationID: corrID,
              EventID:       newUUID(),
          },
          ShortCode:   shortCode,
          UserID:      userID,
          UserEmail:   userEmail,
          MilestoneN:  threshold,
          TotalClicks: totalClicks,
      }
      publishCtx, cancel := context.WithTimeout(ctx, 3*time.Second)
      if publishErr := m.publisher.PublishMilestone(publishCtx, evt); publishErr != nil:
          m.log.Warn("milestone event publish failed",
              "threshold", threshold,
              "short_code", shortCode,
              "error", publishErr,
          )
          // Do NOT return error — milestone row is committed; publish failure is non-fatal
      cancel()
Step 3: Return nil (all thresholds checked)
  return nil
```

Design note on publish-before-commit: Publishing `MilestoneReachedEvent` inside the transaction scope means the event goes out before the milestone row is committed. If the commit subsequently fails, a phantom notification is sent. The alternative (publishing post-commit) requires returning milestone data from `CheckAndPublish` and restructuring the consumer. For this project's scope, the phantom-event risk is accepted because milestone notifications are informational, not transactional. A production system would use the outbox pattern here too.

5.3 `GET /stats/:code` Handler

```
Input:  path parameter: code string
Output: 200 statsResponse | 404 | 500
No authentication required.
Step 1: Extract short_code from path
  code := r.PathValue("code")
  if code == "":
      writeError(w, 400, "short_code is required"); return
Step 2: Execute aggregation queries (concurrent with errgroup for performance)
  g, gCtx := errgroup.WithContext(r.Context())
  var total, last24h, last7d int64
  var topRefs []RefererCount
  g.Go(func() error {
      var err error
      total, err = h.clickStore.CountByCode(gCtx, code)
      return err
  })
  g.Go(func() error {
      var err error
      last24h, err = h.clickStore.CountByCodeSince(gCtx, code, time.Now().UTC().Add(-24*time.Hour))
      return err
  })
  g.Go(func() error {
      var err error
      last7d, err = h.clickStore.CountByCodeSince(gCtx, code, time.Now().UTC().Add(-7*24*time.Hour))
      return err
  })
  g.Go(func() error {
      var err error
      topRefs, err = h.clickStore.TopReferers(gCtx, code, 5)
      if topRefs == nil { topRefs = []RefererCount{} }  // ensure JSON []
      return err
  })
  if err := g.Wait(); err != nil:
      h.log.Error("stats query failed", "code", code, "error", err)
      writeError(w, 500, "internal server error"); return
Step 3: If total == 0 AND we want to return 404 for unknown codes...
  Note: The spec says stats are public and does NOT require the code to exist in url_db
  (analytics trusts events). A short_code with 0 clicks returns 200 with zeroed stats.
  We do NOT call url-service to check existence. Return 200 with zeros.
Step 4: Write response
  writeJSON(w, 200, statsResponse{
      ShortCode:     code,
      TotalClicks:   total,
      ClicksLast24h: last24h,
      ClicksLast7d:  last7d,
      TopReferers:   topRefs,
  })
```

5.4 `GET /stats/:code/timeline` Handler

```
Input:  path parameter: code; query param: interval=day|hour
Output: 200 timelineResponse | 400 | 500
Step 1: Extract and validate params
  code := r.PathValue("code")
  interval := r.URL.Query().Get("interval")
  if interval != "day" && interval != "hour":
      writeError(w, 400, `interval must be "day" or "hour"`); return
Step 2: Map interval to PostgreSQL date_trunc unit
  // "day" → "day", "hour" → "hour"
  // Both are valid PostgreSQL date_trunc units; no remapping needed.
  truncUnit := interval
Step 3: Query buckets
  points, err := h.clickStore.TimelineBuckets(r.Context(), code, truncUnit)
  if err != nil:
      h.log.Error("timeline query failed", "code", code, "error", err)
      writeError(w, 500, "internal server error"); return
  if points == nil { points = []TimelinePoint{} }
Step 4: Write response
  writeJSON(w, 200, timelineResponse{
      ShortCode: code,
      Interval:  interval,
      Points:    points,
  })
```

5.5 Schema Migration on Startup

```go
// main.go — after NewDBPool succeeds, before consumer or HTTP server start

const analyticsSchema = `

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,

    user_agent  TEXT        NOT NULL DEFAULT '',

    referer     TEXT        NULL

);

CREATE INDEX IF NOT EXISTS idx_clicks_short_code_time

    ON clicks(short_code, clicked_at DESC);

CREATE INDEX IF NOT EXISTS idx_clicks_referer

    ON clicks(short_code, referer)

    WHERE referer IS NOT NULL;

CREATE TABLE IF NOT EXISTS milestones (

    id           UUID  PRIMARY KEY DEFAULT gen_random_uuid(),

    short_code   TEXT  NOT NULL,

    milestone    INT   NOT NULL,

    triggered_at TIMESTAMPTZ NOT NULL DEFAULT now(),

    UNIQUE (short_code, milestone)

);

CREATE UNIQUE INDEX IF NOT EXISTS idx_milestones_code_milestone

    ON milestones(short_code, milestone);

CREATE TABLE IF NOT EXISTS processed_events (

    event_id     TEXT PRIMARY KEY,

    processed_at TIMESTAMPTZ NOT NULL DEFAULT now()

);

`

func runMigrations(ctx context.Context, pool *pgxpool.Pool, log *slog.Logger) error {

    _, err := pool.Exec(ctx, analyticsSchema)

    if err != nil {

        return fmt.Errorf("run analytics migrations: %w", err)

    }

    log.Info("analytics migrations applied")
```

```
    return nil
}
```

5.6 `hashIP` Function (`haship.go`)

```go
// haship.go

package analytics

import (

    "crypto/sha256"

    "fmt"

    "net"

)

// hashIP extracts the IP from remoteAddr and returns SHA-256(ip + salt) as a hex string.

// If port splitting fails, uses the raw remoteAddr as the input.

// This function is NOT called in the analytics consumer — the ip_hash is already computed

// by url-service before being embedded in URLClickedEvent.IPHash.

// Provided here for local test utilities and future analytics-internal use.

func hashIP(remoteAddr, salt string) string {

    ip, _, err := net.SplitHostPort(remoteAddr)

    if err != nil {

        ip = remoteAddr

    }

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

    return fmt.Sprintf("%x", h)

}
```

Critical: The analytics consumer reads `evt.IPHash` directly from the `URLClickedEvent`. It does **not** receive a raw IP. The `hashIP` function in analytics-service is present only for test utilities. The ip_hash field in the click row is always the value from the event payload, which was already hashed by url-service.

5.7 `newUUID` Helper

```go
// consumer.go or a shared helpers.go within analytics package

import (

    "crypto/rand"

    "fmt"

)

func newUUID() string {

    b := make([]byte, 16)

    rand.Read(b)

    b[6] = (b[6] & 0x0f) | 0x40

    b[8] = (b[8] & 0x3f) | 0x80

    return fmt.Sprintf("%08x-%04x-%04x-%04x-%012x",

        b[0:4], b[4:6], b[6:8], b[8:10], b[10:16])

}
```

5.8 Route and Consumer Registration in `main.go`

```go
// services/analytics-service/main.go

package main

import (

    "context"

    "fmt"

    "net/http"

    "os"

    "os/signal"

    "syscall"

    "github.com/yourhandle/url-shortener/shared/logger"

)

func main() {

    cfg, err := loadConfig()

    if err != nil {

        fmt.Fprintf(os.Stderr, "config error: %v\n", err)

        os.Exit(1)

    }

    log := logger.New(cfg.ServiceName)

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

    defer cancel()

    pool, err := NewDBPool(ctx, cfg.DatabaseURL, log)

    if err != nil {

        log.Error("db unreachable", "error", err)

        os.Exit(1)

    }

    defer pool.Close()

    if err := runMigrations(ctx, pool, log); err != nil {

        log.Error("migration failed", "error", err)

        os.Exit(1)

    }

    mq, err := NewRabbitMQConn(ctx, cfg.RabbitMQURL, log, 10)

    if err != nil {

        log.Error("rabbitmq unreachable", "error", err)

        os.Exit(1)

    }
```

```
    defer mq.Close()

    if err := DeclareAnalyticsQueue(mq.Channel); err != nil {
        log.Error("declare analytics queue failed", "error", err)
        os.Exit(1)
    }

    // Construct repositories
    clickStore     := NewClickStore(pool)
    milestoneStore := NewMilestoneStore(pool)
    dedupStore     := NewDeduplicationStore(pool)
    publisher      := NewAnalyticsPublisher(mq.Channel, log)
    checker        := NewMilestoneChecker(clickStore, milestoneStore, publisher, log)
    consumer := NewClickConsumer(
        mq, pool, clickStore, milestoneStore, dedupStore, checker, log, cfg.IPHashSalt,
    )

    // Construct HTTP handler
    statsHandler := NewStatsHandler(clickStore, log)
    mux := http.NewServeMux()
    mux.HandleFunc("GET /health", NewHealthHandler(cfg.ServiceName))
    mux.HandleFunc("GET /stats/{code}", statsHandler.Stats)
    mux.HandleFunc("GET /stats/{code}/timeline", statsHandler.Timeline)

    // Start consumer goroutine
    go consumer.Run(ctx)
    srv := &http.Server{Addr: ":" + cfg.Port, Handler: mux}
    go func() {
        log.Info("server listening", "port", cfg.Port)
        if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Error("server error", "error", err)
            os.Exit(1)
        }
    }()
    quit := make(chan os.Signal, 1)
    signal.Notify(quit, syscall.SIGTERM, syscall.SIGINT)
    <-quit
    log.Info("shutting down analytics-service")
    cancel()
    srv.Shutdown(context.Background())
```

```
}
```


```
Startup sequence — analytics-service:
  loadConfig() ── fatal if DATABASE_URL or RABBITMQ_URL missing
       |
       ▼
  logger.New("analytics-service")
       |
       ▼
  NewDBPool() ── fatal on error
       |
       ▼
  runMigrations() ── fatal on error
       |
       ▼
  NewRabbitMQConn() ── fatal after 10 attempts; declares exchange
       |
       ▼
  DeclareAnalyticsQueue() ── fatal on error
       |              declares queue "analytics.clicks"
       |              binds routing key "url.clicked"
       |
       ▼
  Construct: clickStore, milestoneStore, dedupStore, publisher, checker, consumer
       |
       ▼
  go consumer.Run(ctx) ── background goroutine
       |       sets prefetch=1, registers consumer, marks healthy
       |
       ▼
  mux.HandleFunc(...) ── /health, /stats/{code}, /stats/{code}/timeline
       |
       ▼
  srv.ListenAndServe() ── blocks
       |
       | SIGTERM
       ▼
  cancel() → consumer.Run exits → srv.Shutdown()
```

6. Error Handling Matrix

Error Scenario	Detected By	Recovery	HTTP Status / AMQP Action	Log Level	Notes
Missing `DATABASE_URL`	`loadConfig()`	`fmt.Fprintf(stderr)` + `os.Exit(1)`	— (pre-HTTP)	Error (stderr)	Name the missing var
Missing `RABBITMQ_URL`	`loadConfig()`	`os.Exit(1)`	— (pre-HTTP)	Error	
DB unreachable at startup	`NewDBPool` / `pool.Ping`	`os.Exit(1)`	— (pre-HTTP)	Error	Log masked DSN
Migration SQL fails	`runMigrations`	`os.Exit(1)`	— (pre-HTTP)	Error	
RabbitMQ unreachable (all 10 attempts)	`NewRabbitMQConn`	`os.Exit(1)`	— (pre-HTTP)	Error	
Queue declare fails	`DeclareAnalyticsQueue`	`os.Exit(1)`	— (pre-HTTP)	Error	
Consumer QoS set fails	`ch.Qos(1,0,false)`	`os.Exit(1)`	— (pre-HTTP)	Error	Serial guarantees lost without prefetch=1
Consumer registration fails	`ch.Consume(...)`	`os.Exit(1)`	— (pre-HTTP)	Error	
AMQP channel closed mid-run	`<-msgs` ok=false	Mark unhealthy, block on `<-ctx.Done()`	—	Warn	`/health` still returns 200 (healthcheck only checks HTTP)
Malformed JSON in message body	`json.Unmarshal`	`d.Ack(false)`	—	Error	`"poison message"` + body preview (truncated 200 chars)
Missing `event_id` or `short_code`	Field check after decode	`d.Ack(false)`	—	Error	`"poison message: missing required fields"`
Consumer panic	`defer recover()`	`d.Ack(false)`	—	Error	Log panic value + body preview
DB `pool.Begin` fails	`pool.Begin`	`d.Nack(false, true)` — requeue	—	Error	`"begin tx failed"`
`Exists` (dedup check) DB error	`dedupStore.Exists`	Rollback + `d.Nack(false, true)`	—	Error	
`Insert` (dedup) DB error	`dedupStore.Insert`	Rollback + `d.Nack(false, true)`	—	Error	
Duplicate `event_id`	`exists == true`	Rollback + `d.Ack(false)`	—	Info	`"duplicate event skipped"`

Error Scenario	Detected By	Recovery	HTTP Status / AMQP Action	Log Level	Notes
Click insert DB error	`clickStore.Insert`	Rollback + `d.Nack(false, true)`	—	Error	`"click insert failed"`
Milestone count DB error	`tx.QueryRow` COUNT	Rollback + `d.Nack(false, true)`	—	Error	
`HasMilestone` DB error	`milestoneStore.HasMilestone`	Rollback + `d.Nack(false, true)`	—	Error	
`Insert` milestone DB error	`milestoneStore.Insert`	Rollback + `d.Nack(false, true)`	—	Error	
`MilestoneReachedEvent` publish fails	`publisher.PublishMilestone`	Log + continue (tx still commits)	—	Warn	`"milestone event publish failed"` + threshold + code
`tx.Commit` fails	`tx.Commit`	`d.Nack(false, true)`	—	Error	
`GET /stats/:code` DB error (any query)	`errgroup.Wait`	500 response	500	Error	
`GET /stats/.../timeline` invalid interval	param validation	400 response	400	none	
`GET /stats/.../timeline` DB error	`clickStore.TimelineBuckets`	500 response	500	Error	

`writeError` and `writeJSON` helpers (`errors.go`):

```go
// errors.go

package analytics

import (

    "encoding/json"

    "net/http"

)

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

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

    w.WriteHeader(status)

    json.NewEncoder(w).Encode(struct {

        Error string `json:"error"`

    }{Error: message})

}

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

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

    w.WriteHeader(status)

    json.NewEncoder(w).Encode(v)

}

// truncate returns at most n bytes of s for safe logging of untrusted bodies.

func truncate(s string, n int) string {

    if len(s) <= n {

        return s

    }

    return s[:n] + "..."

}
```

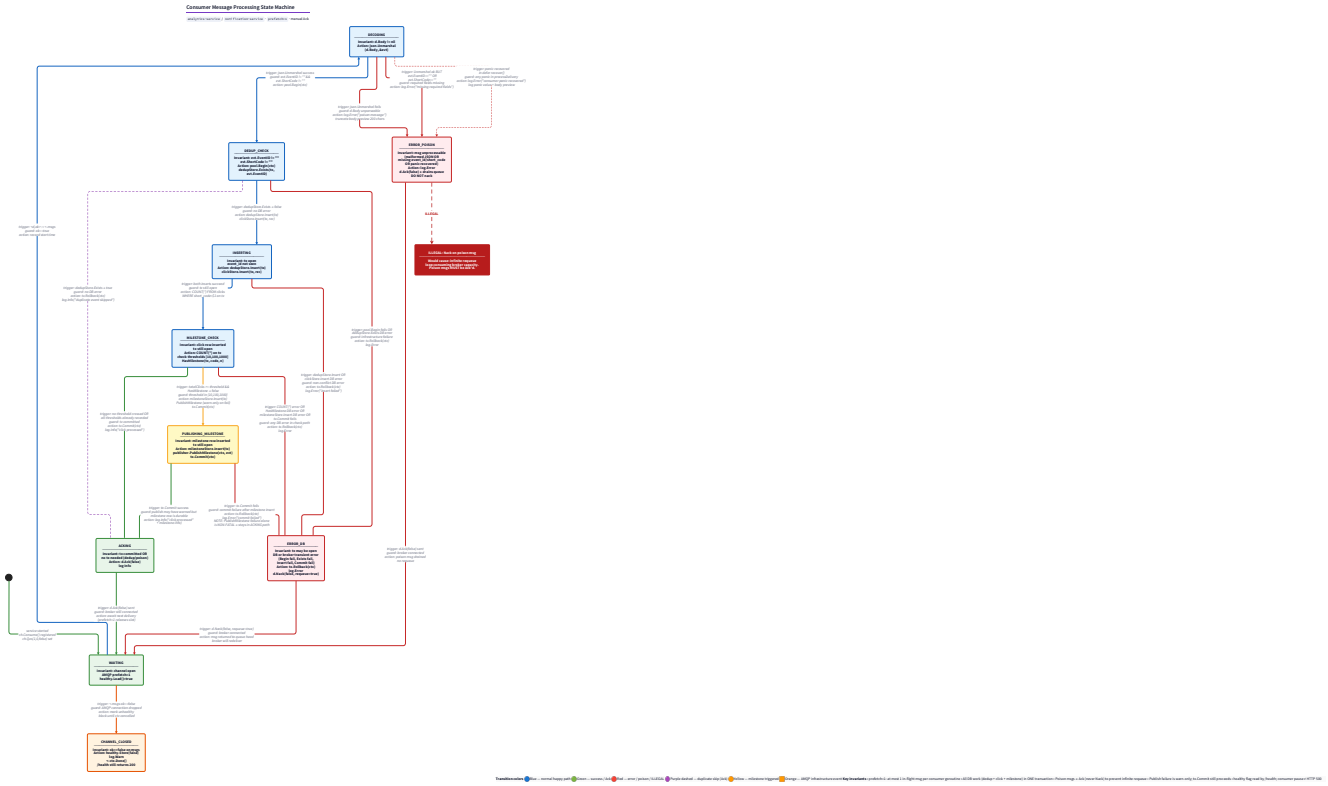
7. Concurrency Specification

Component	Goroutine Model	Shared State	Thread Safety
`ClickConsumer.Run`	1 dedicated goroutine	`*pgxpool.Pool`, `ClickRepository`, `MilestoneRepository`, `DeduplicationRepository`, `*amqp.Channel`	✅ All safe; single goroutine owns AMQP channel — no mutex needed
`StatsHandler.Stats`	per-request (net/http)	`ClickRepository` (pool)	✅ pgxpool goroutine-safe
`StatsHandler.Timeline`	per-request (net/http)	`ClickRepository` (pool)	✅ Same
`errgroup` goroutines in Stats handler	4 sub-goroutines per request	`ClickRepository` (pool)	✅ pgxpool handles concurrent Acquire internally
`amqpAnalyticsPublisher.PublishMilestone`	called only from consumer goroutine	`*amqp.Channel`	✅ Single caller; no mutex needed
`consumer.healthy` atomic	read by health handler, written by consumer	`atomic.Bool`	✅ Lock-free atomic

AMQP prefetch=1 consequence: The consumer goroutine processes exactly one message at a time. The broker will not deliver the next message until `d.Ack()` or `d.Nack()` is called. This provides the sequential processing guarantee without explicit locking in the consumer loop. **No mutex on AMQP channel:** Unlike url-service which has 3 workers sharing one channel, analytics-service has exactly one goroutine (the consumer) using the AMQP channel for both consuming and publishing. No concurrent access occurs. **`atomic.Bool` for health state:** The `healthy` field is written by the consumer goroutine and read by HTTP handler goroutines. Using `sync/atomic.Bool` (Go 1.19+) avoids a mutex on the hot health-check path:

```go
// consumer.go
type ClickConsumer struct {
    // ...
    healthy atomic.Bool
}
// health.go (extended from M1 for consumer health awareness)
func (c *ClickConsumer) IsHealthy() bool {
    return c.healthy.Load()
}
```

Note on `/health` during consumer pause: The spec requires `/health` to return `200` even when RabbitMQ is down. The health handler checks HTTP server liveness only (`{"status":"ok","service":"analytics-service"}`). It does NOT check `consumer.IsHealthy()` — consumer state is operational metadata, not liveness. A Kubernetes readiness probe could check consumer health separately, but that is outside this project's scope.




```
Transaction boundary within processDelivery:

  ┌──────────────────────────────────────────────────────────┐
  │  BEGIN TRANSACTION                                       │
  │                                                          │
  │  dedupStore.Exists(tx, evt.EventID)                      │
  │       │                                                  │
  │   exists=true ──► tx.Rollback() → d.Ack() → return       │
  │       │                                                  │
  │   exists=false                                           │
  │       │                                                  │
  │  dedupStore.Insert(tx, evt.EventID)                      │
  │       │                                                  │
  │  clickStore.Insert(tx, &ClickRecord{...})                │
  │       │                                                  │
  │  checker.CheckAndPublish(ctx, tx, code, ...)             │
  │       │                                                  │
  │       ├── SELECT COUNT(*) FROM clicks WHERE short_code=? (on tx) │
  │       │                                                  │
  │       ├── for each threshold [10, 100, 1000]:           │
  │       │     milestoneStore.HasMilestone(tx, code, threshold)  │
  │       │       │                                          │
  │       │    not recorded yet                              │
  │       │       │                                          │
  │       │     milestoneStore.Insert(tx, code, threshold)   │
  │       │       │                                          │
  │       │     publisher.PublishMilestone(ctx, evt)  ◄── OUTSIDE tx  │
  │       │       │                                          │
  │       │     (publish fail → log + continue; tx not affected)  │
  │       │                                                  │
  │  COMMIT TRANSACTION                                      │
  │                                                          │
  │  d.Ack(false)                                            │
  └──────────────────────────────────────────────────────────┘

  If any step errors before COMMIT: tx.Rollback() + d.Nack(false, true)
  If COMMIT fails: d.Nack(false, true)
  If publish fails after milestone INSERT but before COMMIT:
    publish failure is logged; COMMIT still proceeds (milestone is durable)
```

8. Implementation Sequence with Checkpoints

Phase 1: DB Schema — `clicks`, `milestones`, `processed_events` Tables (1h)

1. Write `migration.sql` as specified in §3.1.
2. Write `runMigrations` function in `main.go` (same pattern as url-service).
3. Extend `config.go` to add `IPHashSalt`.
4. Run against Docker `analytics_db`:

```bash
docker compose up analytics_db -d

DATABASE_URL="postgres://analyticsuser:analyticspass@localhost:5433/analyticsdb" \

RABBITMQ_URL="amqp://guest:guest@localhost:5672/" \

go run ./services/analytics-service/
```

1. Verify schema:

```bash
psql -h localhost -p 5433 -U analyticsuser -d analyticsdb \

  -c "\d clicks; \d milestones; \d processed_events; \di idx_clicks_*; \di idx_milestones_*;"
```

Checkpoint: Tables `clicks`, `milestones`, `processed_events` exist with correct columns and types. All five indexes visible in `\di` output. Service starts and logs `"analytics migrations applied"`. `go build ./services/analytics-service/...` exits 0.

Phase 2: AMQP Consumer Goroutine — Prefetch, Message Loop, Panic Recovery, Idempotency (2–2.5h)

1. Write `errors.go`: `writeError`, `writeJSON`, `truncate`.
2. Write `haship.go`: `hashIP`, `newUUID`.
3. Write `store.go`: `ClickRecord`, `MilestoneRecord`, all three repository interfaces, `pgxClickStore` stub (Insert + CountByCode only), `pgxMilestoneStore` stub (HasMilestone + Insert), `pgxDeduplicationStore` (both methods).
4. Write `publisher.go`: `AnalyticsPublisher` interface, `amqpAnalyticsPublisher`, `PublishMilestone`.
5. Write `milestone.go`: `MilestoneChecker` struct, `NewMilestoneChecker`, `CheckAndPublish` (stub returning nil for now — complete in Phase 3).
6. Write `consumer.go`: `ClickConsumer`, `NewClickConsumer`, `Run`, `processDelivery` (full algorithm from §5.1 and §5.2).
7. Wire consumer in `main.go`: construct all stores, call `go consumer.Run(ctx)`.

```bash
docker compose up analytics_db rabbitmq -d

# Start url-service too so it publishes events on redirect

docker compose up url-service -d

# Start analytics-service

DATABASE_URL=... RABBITMQ_URL=... go run ./services/analytics-service/
```

Manually trigger a URL click (against url-service):

```bash
curl http://localhost:8081/<short_code>
```

Checkpoint: Consumer logs `"click processed"` with `event_id` and `short_code`. `psql -h localhost -p 5433 -U analyticsuser -d analyticsdb -c "SELECT * FROM clicks LIMIT 5;"` shows a row. `processed_events` has the `event_id`. Sending the **same** AMQP message twice (use RabbitMQ management UI to redeliver) logs `"duplicate event skipped"` and `clicks` count stays at 1.

Phase 3: Click Insert, Milestone Check, `MilestoneReachedEvent` Publish — One Transaction (2–2.5h)

1. Complete `pgxClickStore.Insert` with `pgtype.Text` for nullable referer.

2. Complete all remaining `pgxClickStore` methods (`CountByCode` , `CountByCodeSince` , `TopReferers` , `TimelineBuckets`).
3. Complete `MilestoneChecker.CheckAndPublish` (full algorithm from §5.2).
4. Add `pgxMilestoneStore.HasMilestone` and `pgxMilestoneStore.Insert` (full implementations).

```bash
# Send 10 click events for the same short_code

for i in $(seq 1 10); do curl -s http://localhost:8081/<short_code> > /dev/null; done

sleep 3  # wait for outbox poller

psql -h localhost -p 5433 -U analyticsuser -d analyticsdb \

  -c "SELECT short_code, milestone, triggered_at FROM milestones;"

# Expect: one row with milestone=10

# RabbitMQ management UI → notifications.events queue → should show 1 message
```

Checkpoint: After 10 redirects, `milestones` table has one row `(short_code=<code>, milestone=10)` . After 100 total redirects, a second row appears with `milestone=100` . Sending duplicate events does not create additional milestone rows. `MilestoneReachedEvent` appears in `notifications.events` queue (verify via RabbitMQ management UI or by checking queue depth).

Phase 4: `GET /stats/:code` and `GET /stats/:code/timeline` Handlers (1.5–2h)

1. Write `handler.go` : `StatsHandler` , `NewStatsHandler` , `Stats` method, `Timeline` method.
2. Register routes in `main.go` (already shown in §5.8).
3. Test queries with EXPLAIN ANALYZE.

```bash
# After inserting 20+ clicks across different times and referers:

curl http://localhost:8082/stats/<short_code>

# Expect: {"short_code":"...","total_clicks":20,"clicks_last_24h":20,"clicks_last_7d":20,"top_referers":[]}

curl "http://localhost:8082/stats/<short_code>/timeline?interval=day"

# Expect: {"short_code":"...","interval":"day","points":[{"period":"2026-03-02T00:00:00Z","clicks":20}]}

curl "http://localhost:8082/stats/<short_code>/timeline?interval=hour"

# Expect: [{...}] bucketed by hour

curl "http://localhost:8082/stats/<short_code>/timeline?interval=week"

# Expect: 400 {"error":"interval must be \"day\" or \"hour\""}
```

EXPLAIN ANALYZE verification:

```bash
psql -h localhost -p 5433 -U analyticsuser -d analyticsdb -c "

EXPLAIN ANALYZE

SELECT COUNT(*) FROM clicks

WHERE short_code = 'abc1234' AND clicked_at >= now() - interval '24 hours';

"

# Must show: Index Scan using idx_clicks_short_code_time
```

Checkpoint: All four route variants return correct responses. EXPLAIN ANALYZE on `CountByCodeSince` shows `idx_clicks_short_code_time` . EXPLAIN ANALYZE on `TopReferers` shows `idx_clicks_referer` . `GET /health` still returns `200` regardless of RabbitMQ state.

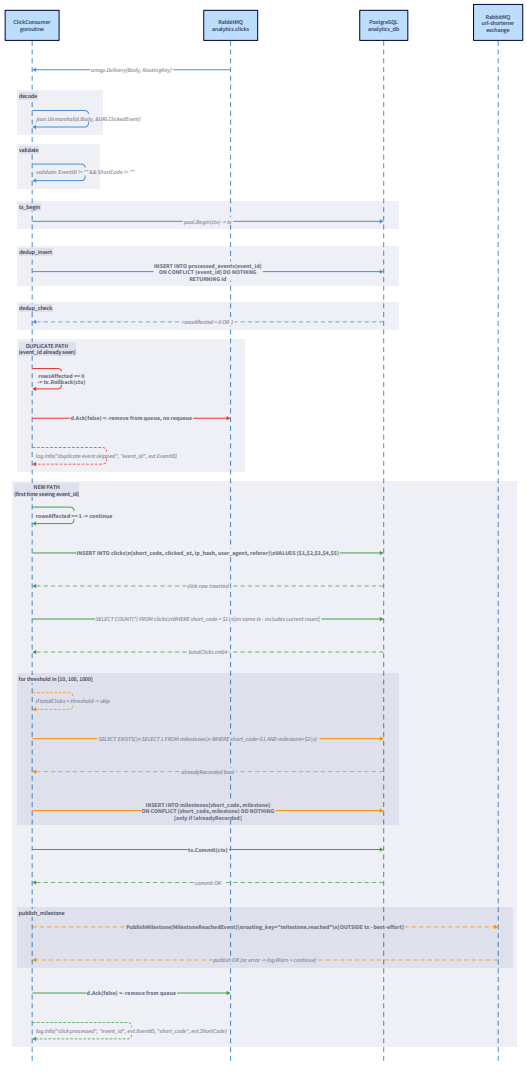
Phase 5: Test — Duplicate `URLClickedEvent` → Click Count = 1 (1h)

1. Write `analytics_test.go` covering all test cases from §9.

2. Run unit tests (mock stores) without Docker.
3. Run integration tests with Docker `analytics_db`.

```bash
# Unit tests (no DB needed)

go test ./services/analytics-service/... -run TestUnit -v

# Integration test (requires running analytics_db)

DATABASE_URL="postgres://analyticsuser:analyticspass@localhost:5433/analyticsdb" \

RABBITMQ_URL="amqp://guest:guest@localhost:5672/" \

go test ./services/analytics-service/... -run TestIntegration -tags integration -v
```

Checkpoint: `TestDuplicateEventIDCountIsOne` passes: after processing the same `URLClickedEvent` twice (same `event_id`), `CountByCode` returns 1. `TestMilestoneTriggeredAtExactly10` passes. `TestPoisonMessageAcked` passes (malformed JSON → consumer continues, no nack). All unit tests green. `go test ./services/analytics-service/...` exits 0.



At-Least-Once + Idempotency Guarantee

Scenario	Outcome
Message delivered once	INSERT processed_events → rowsAffected=1 → click counted
Message redelivered (same event_id)	INSERT → conflict → rowsAffected=0 → ROLLBACK → Ack → no-op
tx.Commit fails after INSERT clicks	d.Nack(requeue=true) → redelivered → dedup fires → no double-count
Milestone publish fails after COMMIT	Milestone row durable; event may be missing; notification deduplicates replays
Prefetch = 1	Broker holds next message until d.Ack or d.Nack - serial processing

```
Test scenario: duplicate URLClickedEvent
  event_id = "evt-abc-123"
  short_code = "xyz7890"
  Message 1 delivered:
    tx.Begin()
    dedupStore.Exists(tx, "evt-abc-123") → false
    dedupStore.Insert(tx, "evt-abc-123")
    clickStore.Insert(tx, ClickRecord{ShortCode:"xyz7890",...})
    checker.CheckAndPublish(tx, "xyz7890",...)
    tx.Commit()
    d.Ack()
  Message 2 delivered (same event_id, redelivered by broker):
    tx.Begin()
    dedupStore.Exists(tx, "evt-abc-123") → true   ← PRIMARY KEY hit
    tx.Rollback()
    d.Ack()   ← NOT Nack; duplicate is permanently consumed
  Result:
    clicks table: 1 row
    processed_events: 1 row
    CountByCode("xyz7890") = 1   √
```

9. Test Specification

9.1 Unit Tests — `MilestoneChecker` (mock stores)

```go
// analytics_test.go

package analytics

import (

    "context"

    "errors"

    "sync/atomic"

    "testing"

    "time"

    "github.com/jackc/pgx/v5"

    "github.com/yourhandle/url-shortener/shared/events"

)

// mockClickStore implements ClickRepository for testing

type mockClickStore struct {

    insertFn           func(ctx context.Context, tx pgx.Tx, rec *ClickRecord) error

    countByCodeFn      func(ctx context.Context, code string) (int64, error)

    countByCodeSinceFn func(ctx context.Context, code string, since time.Time) (int64, error)

    topReferersFn      func(ctx context.Context, code string, n int) ([]RefererCount, error)

    timelineFn         func(ctx context.Context, code, unit string) ([]TimelinePoint, error)

}

// (implement all ClickRepository methods by delegating to Fn fields)

// mockMilestoneStore implements MilestoneRepository for testing

type mockMilestoneStore struct {

    hasMilestoneFn func(ctx context.Context, tx pgx.Tx, code string, m int) (bool, error)

    insertFn       func(ctx context.Context, tx pgx.Tx, code string, m int) error

}

// mockPublisher implements AnalyticsPublisher for testing

type mockPublisher struct {

    publishCount int32

    publishErr   error

    publishFn    func(ctx context.Context, evt *events.MilestoneReachedEvent) error

}

func (m *mockPublisher) PublishMilestone(ctx context.Context, evt *events.MilestoneReachedEvent) error {

    atomic.AddInt32(&m.publishCount, 1)

    if m.publishFn != nil { return m.publishFn(ctx, evt) }
```

```
    return m.publishErr
}

func TestMilestoneChecker_NoMilestoneBelow10(t *testing.T) {
    totalClicks := int64(9)
    clickStore := &mockClickStore{
        // CountByCode on tx returns 9
    }
    checker := NewMilestoneChecker(clickStore, &mockMilestoneStore{}, &mockPublisher{}, slog.Default())
    // pass a mock tx that returns 9 for COUNT(*)
    // verify publishCount == 0
    // verify hasMilestone never called (optimization: skip check when below min threshold)
}
func TestMilestoneChecker_Threshold10Triggered(t *testing.T) {
    publisher := &mockPublisher{}
    milestoneStore := &mockMilestoneStore{
        hasMilestoneFn: func(_ context.Context, _ pgx.Tx, _ string, m int) (bool, error) {
            return false, nil // not yet recorded
        },
        insertFn: func(_ context.Context, _ pgx.Tx, _ string, m int) error {
            return nil
        },
    }
    // totalClicks = 10 (from mock tx COUNT query)
    // Run CheckAndPublish
    // Assert publisher.publishCount == 1
    // Assert published milestone == 10
    if publisher.publishCount != 1 {
        t.Errorf("expected 1 publish, got %d", publisher.publishCount)
    }
}
func TestMilestoneChecker_AlreadyRecorded_NoPublish(t *testing.T) {
    publisher := &mockPublisher{}
    milestoneStore := &mockMilestoneStore{
        hasMilestoneFn: func(_ context.Context, _ pgx.Tx, _ string, m int) (bool, error) {
            return true, nil // already recorded for all thresholds
        },
```

```
    }

    // totalClicks = 100; all thresholds already recorded

    // publishCount must remain 0

    if publisher.publishCount != 0 {

        t.Error("must not publish when milestone already recorded")

    }

}

func TestMilestoneChecker_PublishFailureContinues(t *testing.T) {

    // publisher returns error

    publisher := &mockPublisher{publishErr: errors.New("amqp down")}

    milestoneStore := &mockMilestoneStore{

        hasMilestoneFn: func(_ context.Context, _ pgx.Tx, _ string, m int) (bool, error) {

            return false, nil

        },

        insertFn: func(_ context.Context, _ pgx.Tx, _ string, m int) error { return nil },

    }

    // totalClicks = 10

    // CheckAndPublish must return nil (publish failure is non-fatal)

    // err := checker.CheckAndPublish(ctx, tx, code, "", "", "")

    // if err != nil { t.Errorf("expected nil, got %v", err) }

}

func TestMilestoneChecker_MultipleThresholdsAtOnce(t *testing.T) {

    // totalClicks = 1000 and no milestones recorded yet

    // Expect: 3 publishes (10, 100, 1000) and 3 inserts

    publisher := &mockPublisher{}

    insertCount := 0

    milestoneStore := &mockMilestoneStore{

        hasMilestoneFn: func(_ context.Context, _ pgx.Tx, _ string, m int) (bool, error) {

            return false, nil

        },

        insertFn: func(_ context.Context, _ pgx.Tx, _ string, m int) error {

            insertCount++

            return nil

        },

    }

    // run with totalClicks=1000
```

```
    if publisher.publishCount != 3 {
        t.Errorf("expected 3 publishes, got %d", publisher.publishCount)
    }
    if insertCount != 3 {
        t.Errorf("expected 3 milestone inserts, got %d", insertCount)
    }
}
```

9.2 Unit Tests — Consumer Idempotency and Poison Messages

```go
func TestProcessDelivery_DuplicateEventID_SingleClick(t *testing.T) {
    // Integration-style test using miniredis-equivalent for DB: testcontainers or real analyticsdb
    // 1. Create consumer with real pool (requires analytics_db running)
    // 2. Build a valid URLClickedEvent JSON with event_id = "dedup-test-001"
    // 3. Call consumer.processDelivery twice with the same event_id
    // 4. Assert clickStore.CountByCode returns 1
    // This is the PRIMARY acceptance criterion for M4.
}
func TestProcessDelivery_MalformedJSON_Acked(t *testing.T) {
    // Create a mock amqp.Delivery with Body = []byte("not json")
    // Call processDelivery
    // Assert: delivery was Acked (not Nacked)
    // Assert: no click row inserted
    // Implementation note: wrap amqp.Delivery to allow test inspection of Ack/Nack calls
}
func TestProcessDelivery_MissingEventID_Acked(t *testing.T) {
    // Body = valid JSON but event_id = ""
    // Assert: Acked, no click row
}
func TestProcessDelivery_MissingShortCode_Acked(t *testing.T) {
    // Body = valid JSON but short_code = ""
    // Assert: Acked, no click row
}
func TestProcessDelivery_DBError_Nacked(t *testing.T) {
    // Mock pool.Begin to fail
    // Assert: Nacked with requeue=true
}
```

9.3 Unit Tests — `StatsHandler`

```go
func TestStatsHandler_Returns200WithZeros_UnknownCode(t *testing.T) {
    store := &mockClickStore{
        countByCodeFn:      func(_ context.Context, _ string) (int64, error) { return 0, nil },
        countByCodeSinceFn: func(_ context.Context, _ string, _ time.Time) (int64, error) { return 0, nil },
        topReferersFn:      func(_ context.Context, _ string, _ int) ([]RefererCount, error) { return []RefererCount{}, nil
},
    }
    h := NewStatsHandler(store, slog.Default())
    req := httptest.NewRequest("GET", "/stats/nonexistent", nil)
    req.SetPathValue("code", "nonexistent")
    rec := httptest.NewRecorder()
    h.Stats(rec, req)
    if rec.Code != 200 {
        t.Errorf("expected 200, got %d", rec.Code)
    }
    var resp statsResponse
    json.Unmarshal(rec.Body.Bytes(), &resp)
    if resp.TotalClicks != 0 { t.Error("total_clicks should be 0") }
    if resp.TopReferers == nil { t.Error("top_referers must be [] not null") }
}
func TestStatsHandler_TopReferers_MaxFive(t *testing.T) {
    store := &mockClickStore{
        countByCodeFn:      func(_ context.Context, _ string) (int64, error) { return 100, nil },
        countByCodeSinceFn: func(_ context.Context, _ string, _ time.Time) (int64, error) { return 50, nil },
        topReferersFn: func(_ context.Context, _ string, n int) ([]RefererCount, error) {
            if n != 5 { return nil, fmt.Errorf("expected n=5, got %d", n) }
            return []RefererCount{
                {"https://google.com", 40},
                {"https://twitter.com", 20},
                {"https://reddit.com", 15},
                {"https://hn.algolia.com", 10},
                {"https://lobste.rs", 5},
            }, nil
        },
    }
    h := NewStatsHandler(store, slog.Default())
```

```
    req := httptest.NewRequest("GET", "/stats/abc1234", nil)
    req.SetPathValue("code", "abc1234")
    rec := httptest.NewRecorder()
    h.Stats(rec, req)
    var resp statsResponse
    json.Unmarshal(rec.Body.Bytes(), &resp)
    if len(resp.TopReferers) != 5 {
        t.Errorf("expected 5 top_referers, got %d", len(resp.TopReferers))
    }
    if resp.TopReferers[0].Referer != "https://google.com" {
        t.Errorf("first referer wrong: %q", resp.TopReferers[0].Referer)
    }
}
func TestStatsHandler_DBError_Returns500(t *testing.T) {
    store := &mockClickStore{
        countByCodeFn: func(_ context.Context, _ string) (int64, error) {
            return 0, errors.New("db down")
        },
        countByCodeSinceFn: func(_ context.Context, _ string, _ time.Time) (int64, error) { return 0, nil },
        topRefererersFn:      func(_ context.Context, _ string, _ int) ([]RefererCount, error) { return nil, nil },
    }
    h := NewStatsHandler(store, slog.Default())
    req := httptest.NewRequest("GET", "/stats/abc1234", nil)
    req.SetPathValue("code", "abc1234")
    rec := httptest.NewRecorder()
    h.Stats(rec, req)
    if rec.Code != 500 { t.Errorf("expected 500, got %d", rec.Code) }
}
func TestTimelineHandler_InvalidInterval_Returns400(t *testing.T) {
    h := NewStatsHandler(&mockClickStore{}, slog.Default())
    for _, bad := range []string{"week", "month", "", "DAY", "Hour"} {
        t.Run(bad, func(t *testing.T) {
            req := httptest.NewRequest("GET", "/stats/abc/timeline?interval="+bad, nil)
            req.SetPathValue("code", "abc")
            rec := httptest.NewRecorder()
            h.Timeline(rec, req)
```

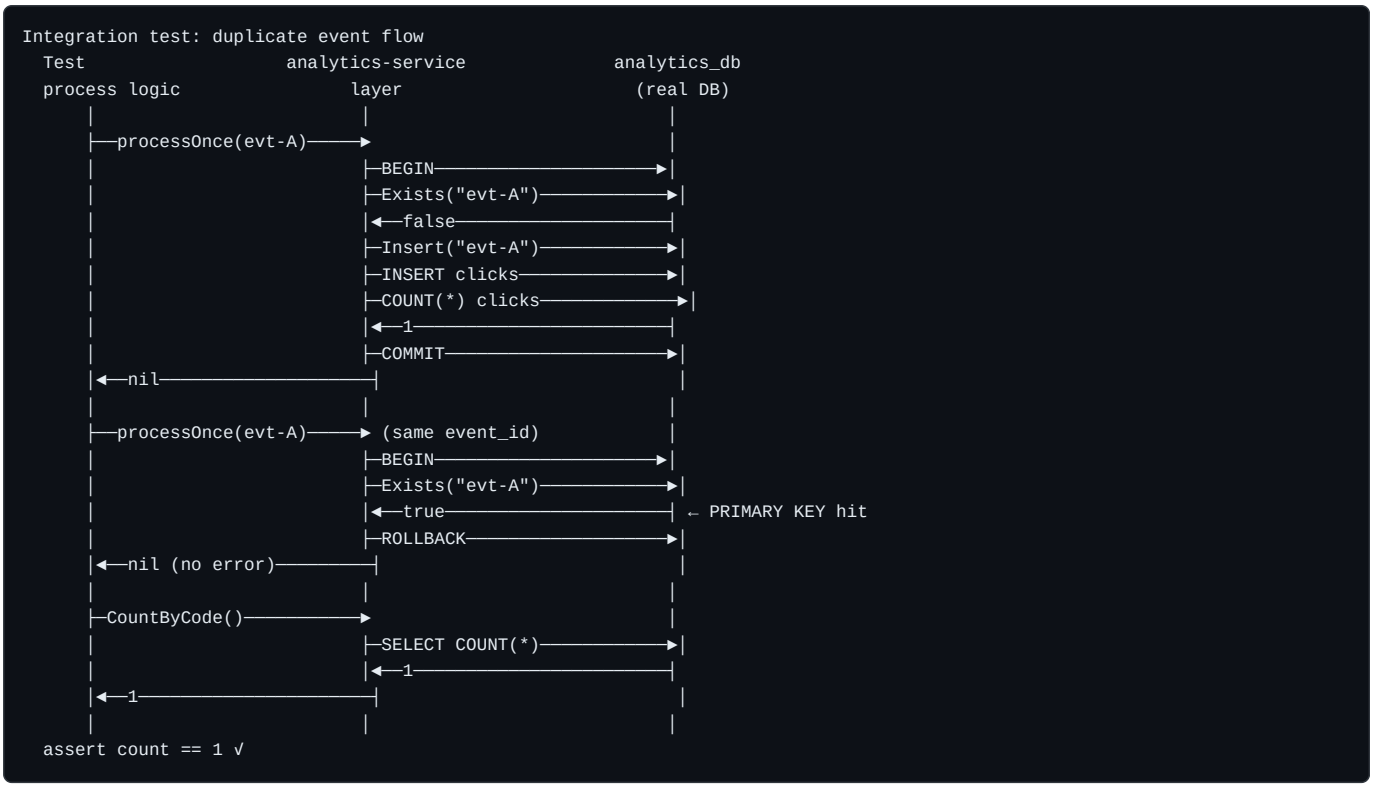
```
            if rec.Code != 400 {
                t.Errorf("interval=%q: expected 400, got %d", bad, rec.Code)
            }
        })
    }
}
func TestTimelineHandler_ValidIntervals_Return200(t *testing.T) {
    store := &mockClickStore{
        timelineFn: func(_ context.Context, _ string, _ string) ([]TimelinePoint, error) {
            return []TimelinePoint{}, nil
        },
    }
    h := NewStatsHandler(store, slog.Default())
    for _, good := range []string{"day", "hour"} {
        t.Run(good, func(t *testing.T) {
            req := httptest.NewRequest("GET", "/stats/abc/timeline?interval="+good, nil)
            req.SetPathValue("code", "abc")
            rec := httptest.NewRecorder()
            h.Timeline(rec, req)
            if rec.Code != 200 {
                t.Errorf("interval=%q: expected 200, got %d", good, rec.Code)
            }
            var resp timelineResponse
            json.Unmarshal(rec.Body.Bytes(), &resp)
            if resp.Points == nil { t.Error("points must be [] not null") }
        })
    }
}
```

9.4 Integration Test — Duplicate `URLClickedEvent` → Click Count = 1 (Primary Acceptance Criterion)

```go
// +build integration

// Run: DATABASE_URL=... RABBITMQ_URL=... go test -tags integration -run TestIntegrationDuplicate ./services/analytics-service/

func TestIntegrationDuplicateEventIDCountIsOne(t *testing.T) {

    cfg, err := loadConfig()

    if err != nil {

        t.Skipf("config not available: %v", err)

    }

    pool, err := NewDBPool(context.Background(), cfg.DatabaseURL, slog.Default())

    if err != nil { t.Fatalf("db: %v", err) }

    t.Cleanup(pool.Close)

    if err := runMigrations(context.Background(), pool, slog.Default()); err != nil {

        t.Fatalf("migrate: %v", err)

    }

    clickStore     := NewClickStore(pool)

    milestoneStore := NewMilestoneStore(pool)

    dedupStore     := NewDeduplicationStore(pool)

    publisher      := &mockPublisher{} // don't need real RabbitMQ for this test

    checker        := NewMilestoneChecker(clickStore, milestoneStore, publisher, slog.Default())

    // Unique short_code and event_id for this test run

    testCode    := fmt.Sprintf("tst%d", time.Now().UnixNano()%100000)

    testEventID := newUUID()

    // Build a valid URLClickedEvent payload

    evt := events.URLClickedEvent{

        BaseEvent: events.BaseEvent{

            EventType:  events.EventTypeURLClicked,

            OccurredAt: time.Now().UTC(),

            EventID:    testEventID,

        },

        ShortCode: testCode,

        IPHash:    "aabbcc",

        UserAgent: "test-agent",

        ClickedAt: time.Now().UTC(),

    }

    body, _ := json.Marshal(evt)

    // Helper: simulate processDelivery logic directly (without real AMQP)
```

```
processOnce := func() error {
    tx, err := pool.Begin(context.Background())
    if err != nil { return err }
    exists, err := dedupStore.Exists(context.Background(), tx, testEventID)
    if err != nil { tx.Rollback(context.Background()); return err }
    if exists {
        tx.Rollback(context.Background())
        return nil // duplicate — expected on second call
    }
    if err := dedupStore.Insert(context.Background(), tx, testEventID); err != nil {
        tx.Rollback(context.Background()); return err
    }
    rec := &ClickRecord{
        ShortCode: evt.ShortCode,
        ClickedAt: evt.ClickedAt,
        IPHash:    evt.IPHash,
        UserAgent: evt.UserAgent,
    }
    if err := clickStore.Insert(context.Background(), tx, rec); err != nil {
        tx.Rollback(context.Background()); return err
    }
    if err := checker.CheckAndPublish(context.Background(), tx, testCode, "", "", ""); err != nil {
        tx.Rollback(context.Background()); return err
    }
    return tx.Commit(context.Background())
}
// Process the same event twice
if err := processOnce(); err != nil { t.Fatalf("first process: %v", err) }
if err := processOnce(); err != nil { t.Fatalf("second process: %v", err) }
// Assert click count is exactly 1
count, err := clickStore.CountByCode(context.Background(), testCode)
if err != nil { t.Fatalf("count: %v", err) }
if count != 1 {
    t.Errorf("click count after 2 identical events: got %d, want 1", count)
}
// Assert processed_events has exactly 1 row for this event_id
```

```
    var dedupCount int
    pool.QueryRow(context.Background(),
        "SELECT COUNT(*) FROM processed_events WHERE event_id = $1",
        testEventID,
    ).Scan(&dedupCount)
    if dedupCount != 1 {
        t.Errorf("processed_events count: got %d, want 1", dedupCount)
    }
    _ = body // suppress unused warning
}
func TestIntegrationMilestoneAt10(t *testing.T) {
    // Insert 10 unique URLClickedEvents for the same short_code
    // Assert milestones table has exactly one row with milestone=10
    // Assert CountByCode = 10
}
func TestIntegrationTopReferers(t *testing.T) {
    // Insert 20 clicks: 10 from "https://google.com", 5 from "https://twitter.com", 5 with nil referer
    // Call TopReferers(code, 5)
    // Assert: 2 entries returned; google.com first with count=10
    // Assert: nil-referer clicks NOT included
}
func TestIntegrationTimelineDay(t *testing.T) {
    // Insert 5 clicks on 2026-03-01 and 3 clicks on 2026-03-02 (via direct INSERT with past clicked_at)
    // Call TimelineBuckets(code, "day")
    // Assert: 2 TimelinePoints returned, ordered ASC
    // Assert: point[0].Clicks == 5, point[1].Clicks == 3
}
```



9.5 `pgxClickStore` Referer NULL Handling Test

```go
func TestClickInsert_NullReferer(t *testing.T) {

    // Insert a ClickRecord with Referer = ""

    // Verify the database stores NULL (not empty string "")

    // SELECT referer FROM clicks WHERE id = inserted_id → should return NULL

    // This ensures the partial index idx_clicks_referer filters it out correctly

}

func TestClickInsert_NonNullReferer(t *testing.T) {

    // Insert a ClickRecord with Referer = "https://example.com"

    // SELECT referer FROM clicks WHERE id = inserted_id → should return "https://example.com"

    // SELECT COUNT(*) FROM clicks WHERE short_code = ? AND referer IS NOT NULL → 1

}
```

10. Performance Targets

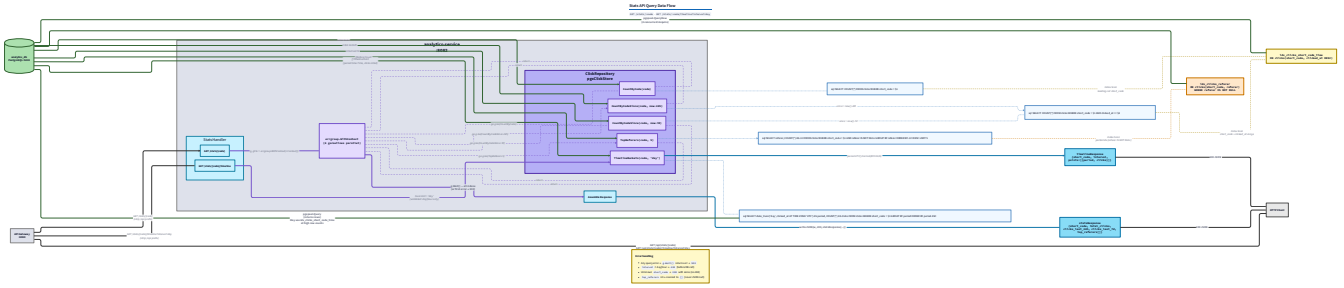
Operation	Target	How to Measure
`URLClickedEvent` consumer processing (dedup + click insert + milestone check + ack)	< 50ms p99	Log `duration_ms` in `processDelivery`; after processing 1000 messages, check log percentiles with `jq`
`GET /stats/:code` (4 concurrent aggregation queries)	< 30ms p99	`wrk -t4 -c20 -d30s http://localhost:8082/stats/<code>` → Latency 99th < 30ms
`GET /stats/:code/timeline?interval=day`	< 50ms p99	`wrk -t4 -c20 -d30s 'http://localhost:8082/stats/<code>/timeline?interval=day'`
`GET /stats/:code/timeline?interval=hour`	< 50ms p99	Same as day
`GET /health`	< 10ms p99	`wrk -t1 -c10 -d10s http://localhost:8082/health`
`CountByCode` query (1M click rows)	< 20ms	`EXPLAIN ANALYZE SELECT COUNT(*) FROM clicks WHERE short_code = '...'` — must show Index Scan
`CountByCodeSince` query (1M rows, 24h window)	< 20ms	Same; must show Index Scan using `idx_clicks_short_code_time`
`TopReferers` query	< 15ms	`EXPLAIN ANALYZE SELECT referer, COUNT(*) ... WHERE short_code = '...' AND referer IS NOT NULL`
Consumer memory steady-state	< 30MB	`docker stats analytics-service-1` after 30 min of load
`processed_events` PRIMARY KEY lookup	< 1ms	`EXPLAIN ANALYZE SELECT EXISTS(SELECT 1 FROM processed_events WHERE event_id = '...')`

EXPLAIN ANALYZE verification commands:

```bash
psql -h localhost -p 5433 -U analyticsuser -d analyticsdb -c "

EXPLAIN ANALYZE

SELECT COUNT(*) FROM clicks

WHERE short_code = 'abc1234' AND clicked_at >= now() - interval '24 hours';

"

# Must show: Index Scan using idx_clicks_short_code_time on clicks

psql -h localhost -p 5433 -U analyticsuser -d analyticsdb -c "

EXPLAIN ANALYZE

SELECT referer, COUNT(*) AS cnt

FROM clicks

WHERE short_code = 'abc1234' AND referer IS NOT NULL

GROUP BY referer ORDER BY cnt DESC LIMIT 5;

"

# Must show: Index Scan using idx_clicks_referer (partial index)

psql -h localhost -p 5433 -U analyticsuser -d analyticsdb -c "

EXPLAIN ANALYZE

SELECT date_trunc('day', clicked_at AT TIME ZONE 'UTC') AS period, COUNT(*) AS clicks

FROM clicks WHERE short_code = 'abc1234'

GROUP BY period ORDER BY period ASC;

"

# May show Seq Scan if row count is low (optimizer preference); with 10k+ rows, Index Scan expected
```

Consumer throughput logging:

```go
// In processDelivery, add timing instrumentation:

start := time.Now()

// ... all processing steps ...

c.log.Info("click processed",

    "event_id",      evt.EventID,

    "short_code",    evt.ShortCode,

    "duration_ms",   time.Since(start).Milliseconds(),

    "correlation_id", corrID,

)
```



```
GET /stats/:code concurrency (errgroup):
  StatsHandler.Stats
        │
        ├─ errgroup.WithContext(r.Context())
        │
        ├─ go CountByCode(gCtx, code) ──────────────────────────►
        │                                                  analytics_db
        ├─ go CountByCodeSince(gCtx, code, now-24h) ──────────►
        │                                                  (4 concurrent
        ├─ go CountByCodeSince(gCtx, code, now-7d) ──────────►  pool acquires)
        │
        ├─ go TopReferers(gCtx, 5) ──────────────────────────►
        │
        └─ g.Wait() ◄── blocks until all 4 return
              │
           if any error:
               500 response
           else:
               200 statsResponse{...}
Performance: 4 queries run concurrently (~5ms each) → ~5ms total (parallelism)
             vs sequential (~20ms) → errgroup gives ~4× speedup for this endpoint.
pgxpool.Acquire is goroutine-safe; 4 concurrent acquires from pool(MaxConns=10)
all succeed without blocking given adequate pool size.
```

10.1 `go.mod` Additions for Analytics Service

```
// services/analytics-service/go.mod
module github.com/yourhandle/url-shortener/analytics-service
go 1.23
require (
    github.com/yourhandle/url-shortener/shared/events v0.0.0
    github.com/yourhandle/url-shortener/shared/logger v0.0.0
    github.com/jackc/pgx/v5 v5.6.0
    github.com/rabbitmq/amqp091-go v1.10.0
    golang.org/x/sync v0.7.0         // for errgroup in stats handler
)
replace (
    github.com/yourhandle/url-shortener/shared/events => ../../shared/events
    github.com/yourhandle/url-shortener/shared/logger => ../../shared/logger
)
```

Note: `golang.org/x/sync` provides `errgroup`. Alternatively, implement the four concurrent stats queries manually with goroutines + channels; `errgroup` is cleaner.

10.2 `docker-compose.yml` Additions

```yaml
analytics-service:
  environment:
    DATABASE_URL: postgres://analyticsuser:analyticspass@analytics_db:5432/analyticsdb
    RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
    IP_HASH_SALT: "change-this-random-salt" # same salt as url-service for consistency
    PORT: "8080"
```

11. Anti-Pattern Guard

The following assertions must hold before this module is considered complete:

```bash
# 1. Analytics service never connects to url_db, user_db, or notification_db
grep -r "url_db\|user_db\|notification_db\|5432\|5434\|5435" \
  services/analytics-service/*.go | grep -v "_test.go" || echo "PASS: no cross-DB access"
# 2. Analytics service never calls url-service HTTP API
grep -r "url-service\|8081\|url_service" \
  services/analytics-service/*.go | grep -v "_test.go" || echo "PASS: no sync call to url-service"
# 3. Analytics service never stores raw IP addresses
grep -r "RemoteAddr\|r\.RemoteAddr" \
  services/analytics-service/*.go | grep -v "haship\|_test" || echo "PASS: no raw IP storage"
# 4. Consumer uses manual Ack (not autoAck)
grep "autoAck.*false\|false.*autoAck" services/analytics-service/consumer.go && echo "PASS" \
  || echo "FAIL: autoAck must be false"
# 5. Poison messages are Acked (not Nacked) — prevent requeue loop
grep -A3 "json.Unmarshal.*err" services/analytics-service/consumer.go | grep "Ack" \
  && echo "PASS: poison msg acked" || echo "FAIL: check Ack vs Nack on parse error"
# 6. No fmt.Println in handlers or consumer (use structured logger)
grep -n "fmt.Println\|fmt.Printf" services/analytics-service/*.go \
  | grep -v "_test.go" || echo "PASS: no fmt.Println"
```

Notification Service + API Gateway + Circuit Breaker

Module ID: `url-shortener-m5`

1. Module Charter

This module implements the two final infrastructure pieces that complete the system: the Notification Service and the API Gateway. The Notification Service owns the `notifications` table in `notification_db`, consumes `url.created`, `url.deleted`, and `milestone.reached` routing keys from the `notifications.events` RabbitMQ queue, persists notification rows with status tracking, logs mock email sends, and exposes `GET /notifications` with JWT verification. The API Gateway is a pure infrastructure component that provides the single client-facing entry point: it routes requests to the appropriate upstream service, verifies JWT tokens on all `/api/*` routes except `/api/auth/*`, enforces per-IP token-bucket rate limiting via Redis INCR+EXPIRE, implements a three-state circuit breaker (CLOSED → OPEN → HALF_OPEN → CLOSED) protecting the url-service proxy path, propagates `X-Correlation-ID` through all forwarded requests, and emits structured JSON logs for every request. This module does **not** implement any business logic in the gateway — the gateway has no knowledge of URL records, users, clicks, or analytics. It does **not** have the notification service call any other service to look up user email addresses — all necessary context (user email, short code) is carried in the event payload itself (established in M1 `shared/events`). It does **not** implement the outbox pattern in either the notification service or the gateway. It does **not** add new domain events — it only consumes events produced by prior milestones. The circuit breaker applies only to the url-service proxy; other upstream proxies use standard error handling. **Upstream dependencies:** `shared/events` (event structs), `shared/auth` (JWT verification), `shared/logger` (structured JSON logger — extended in this module with `correlation_id` default field), RabbitMQ `notifications.events` queue (declared in M1), `notification_db` PostgreSQL, Redis (shared instance from M1, now used by gateway for rate limiting). **Downstream consumers:** End users and API clients communicate exclusively through the gateway. No service calls the notification service. **Invariants that must always hold:**

- The gateway contains zero domain logic — no imports of `shared/events`, no URL record lookups, no analytics queries.
- The circuit breaker state transitions are guarded by a single `sync.Mutex` — no goroutine ever reads `failures` or `lastFailure` without holding the lock.
- Redis errors in the rate limiter cause fail-open behavior (request is allowed through) and are logged at `Warn` — they never return 5xx to the client.
- JWT verification in the gateway is stateless local HMAC verification — the gateway never calls user-service per request.
- The notification service consumer is a single goroutine with `prefetch=1` — no concurrent message processing occurs.
- `notification_db` is the only database the notification service ever writes to.

2. File Structure

Create files in the numbered order below.

```
url-shortener/
│
├── shared/
│   └── logger/
│       └── 1. logger.go          ← extend: WithCorrelationID helper, request logger middleware
│
├── services/
│   └── notification-service/
│       ├── (from M1)
│       │   ├── go.mod            ← add dependencies
│       │   ├── main.go           ← extend: wire consumer, handler, routes
│       │   ├── config.go         ← extend: add JWTSecret field
│       │   ├── db.go             ← unchanged
│       │   ├── rabbitmq.go       ← unchanged (DeclareNotificationQueue from M1)
│       │   └── health.go         ← unchanged
│       │
│       ├── 2.  migration.sql     ← notifications table + index
│       ├── 3.  store.go          ← NotificationRepository interface + pgxNotificationStore
│       ├── 4.  consumer.go       ← NotificationConsumer, Run(), processDelivery(), mockEmail()
│       ├── 5.  handler.go        ← NotificationHandler, ListNotifications method
│       ├── 6.  errors.go         ← writeError, writeJSON helpers
│       └── 7.  notification_test.go ← unit + integration tests
│
└── gateway/
    ├── (from M1 stub)
    │   ├── go.mod                ← add all dependencies
    │   ├── main.go               ← extend: full wiring
    │   ├── config.go             ← extend: full config
    │   ├── health.go             ← unchanged
    │   └── Dockerfile            ← unchanged
    │
    ├── 8.  router.go             ← routing table, route matching, upstream selection
    ├── 9.  proxy.go              ← ReverseProxy per upstream, request forwarding, response copy
    ├── 10. middleware.go         ← CorrelationIDMiddleware, LoggingMiddleware, chain helper
    ├── 11. ratelimit.go          ← RateLimiter interface, redisTokenBucket, 429 response
    ├── 12. circuitbreaker.go     ← CircuitBreaker struct, state machine, Do() method
    ├── 13. jwtmiddleware.go      ← gateway-local JWT middleware (wraps shared/auth)
    └── 14. gateway_test.go       ← circuit breaker, rate limiter, routing, integration tests
```

3. Complete Data Model

3.1 PostgreSQL Schema — Notification Service (`migration.sql`)

```sql
-- ── notifications table ──────────────────────────────────────────────

CREATE TABLE IF NOT EXISTS notifications (

    id         UUID        PRIMARY KEY DEFAULT gen_random_uuid(),

    user_id    UUID        NOT NULL,

    event_type TEXT        NOT NULL,   -- routing key: "url.created" | "url.deleted" | "milestone.reached"

    payload    JSONB       NOT NULL,   -- full event JSON for audit; never queried for filtering

    status     TEXT        NOT NULL DEFAULT 'sent',  -- "pending" | "sent" | "failed"

    created_at TIMESTAMPTZ NOT NULL DEFAULT now(),

    sent_at    TIMESTAMPTZ NULL        -- set when mock email is logged; NULL if pending/failed

);

-- Primary query: caller's notifications newest-first, for GET /notifications pagination.

CREATE INDEX IF NOT EXISTS idx_notifications_user_created

    ON notifications(user_id, created_at DESC);
```

Column rationale:

Column	Type	Constraint	Rationale
`id`	UUID	PK	Non-enumerable row identity
`user_id`	UUID	NOT NULL	Extracted from event payload; used to filter GET /notifications results
`event_type`	TEXT	NOT NULL	Routing key literal; displayed in API response and logs
`payload`	JSONB	NOT NULL	Full event JSON for audit trail; allows replaying notifications without re-querying upstream
`status`	TEXT	NOT NULL DEFAULT 'sent'	Tracks notification delivery state; `sent` immediately after mock email log
`created_at`	TIMESTAMPTZ	NOT NULL	Cursor key for pagination; matches `idx_notifications_user_created`
`sent_at`	TIMESTAMPTZ	NULL	Set when `mockEmail()` completes; NULL for failed or pending rows
Status lifecycle: In this implementation, all notifications immediately transition `pending → sent` in a single handler call (mock email = log line). The `status` column and `sent_at` column exist to support real email integration without schema changes.			

3.2 Go Structs — Notification Service

```go
// store.go

package notification

import "time"

// NotificationRecord maps to one row in the notifications table.
type NotificationRecord struct {
    ID        string     // UUID string
    UserID    string     // UUID string; extracted from event payload
    EventType string     // routing key literal
    Payload   []byte     // raw JSONB bytes (the full event JSON)
    Status    string     // "pending" | "sent" | "failed"
    CreatedAt time.Time
    SentAt    *time.Time // nil if pending/failed
}

// NotificationRepository abstracts all DB operations on the notifications table.
type NotificationRepository interface {
    // Insert creates a new notification row.
    // rec.Status must be "pending" on input; implementation sets it to "sent" and
    // populates SentAt after the mock email log succeeds.
    // Both the insert and the status update occur in a single transaction.
    Insert(ctx context.Context, rec *NotificationRecord) (*NotificationRecord, error)
    // FindByUserID returns paginated notifications for a user, newest-first.
    // afterID is the cursor UUID; empty string = first page.
    // limit is the page size (default 20, max 50).
    // Returns ([]*NotificationRecord, nextCursorID string, error).
    FindByUserID(ctx context.Context, userID, afterID string, limit int) ([]*NotificationRecord, string, error)
}
```

```go
// config.go (extended from M1)

package notification

import (

    "fmt"

    "os"

)

type Config struct {

    DatabaseURL string // required; fatal if empty

    RabbitMQURL string // required; fatal if empty

    Port        string // default "8080"

    ServiceName string // constant "notification-service"

    JWTSecret   string // required; must match JWT_SECRET across all services

}

func loadConfig() (*Config, error) {

    cfg := &Config{

        DatabaseURL: os.Getenv("DATABASE_URL"),

        RabbitMQURL: os.Getenv("RABBITMQ_URL"),

        JWTSecret:   os.Getenv("JWT_SECRET"),

        Port:        envOrDefault("PORT", "8080"),

        ServiceName: "notification-service",

    }

    if cfg.DatabaseURL == "" {

        return nil, fmt.Errorf("DATABASE_URL is required")

    }

    if cfg.RabbitMQURL == "" {

        return nil, fmt.Errorf("RABBITMQ_URL is required")

    }

    if cfg.JWTSecret == "" {

        return nil, fmt.Errorf("JWT_SECRET is required")

    }

    return cfg, nil

}
```

3.3 HTTP Response Schema — Notification Service

```go
// handler.go

package notification

type notificationItem struct {

    ID        string  `json:"id"`

    EventType string  `json:"event_type"`

    Payload   any     `json:"payload"`    // re-decoded from JSONB for JSON embedding

    Status    string  `json:"status"`

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

    SentAt    *string `json:"sent_at,omitempty"` // RFC3339 or absent

}

type notificationListResponse struct {

    Notifications []notificationItem `json:"notifications"` // always array, never null

    NextCursor    *string            `json:"next_cursor"`   // null if no more pages

}
```

3.4 Go Structs — Gateway

```go
// router.go

package gateway

import "net/http"

// Route describes one entry in the routing table.

// The gateway matches incoming requests against all routes and selects the first match.

type Route struct {

    Method        string   // HTTP method, or "" to match any method

    PathPrefix    string   // matched with strings.HasPrefix against r.URL.Path

    Upstream      string   // config key: "url-service" | "analytics-service" | "user-service" | "notification-service"

    StripPrefix   string   // strip this prefix before forwarding (e.g., "/api" stripped, "/shorten" left)

    RequiresAuth  bool     // true = JWT required; false = public

    RateLimitKey  string   // "" = no rate limit; "shorten" | "redirect" = apply named limit

}
```

```go
// circuitbreaker.go

package gateway

import (

    "sync"

    "time"

)

// State represents the circuit breaker's current state.

type State int

const (

    StateClosed   State = iota // normal operation; requests pass through

    StateOpen                  // tripped; requests fail immediately with 503

    StateHalfOpen              // probe mode; one request allowed through to test upstream

)

func (s State) String() string {

    switch s {

    case StateClosed:   return "CLOSED"

    case StateOpen:     return "OPEN"

    case StateHalfOpen: return "HALF_OPEN"

    default:            return "UNKNOWN"

    }

}

// CircuitBreaker implements a three-state circuit breaker protecting one upstream.

// All state is protected by mu. Do not embed State in other structs without this protection.

type CircuitBreaker struct {

    mu              sync.Mutex

    state           State

    failures        int       // consecutive failure count (reset on success in CLOSED)

    lastFailureTime time.Time // time of most recent failure; drives OPEN→HALF_OPEN timeout

    // Configuration (immutable after construction):

    maxFailures     int           // failures to trip: default 5

    openTimeout     time.Duration // how long to stay OPEN before probing: default 30s

    failureWindow   time.Duration // window in which failures are counted: default 10s

    windowStart     time.Time     // when current failure window began

}
```

```go
// ratelimit.go

package gateway

import "context"

// RateLimiter abstracts the rate limiting strategy.

// Implementation uses Redis INCR + EXPIRE for shared state across gateway replicas.

type RateLimiter interface {

    // Allow checks whether the given key has capacity remaining.

    // key is typically "rl:{route_key}:{ip}" e.g. "rl:shorten:192.168.1.1"

    // limit is the maximum requests per window.

    // windowSecs is the window duration in seconds (EXPIRE value).

    //

    // Returns:

    //   allowed bool   - true if request should proceed; false if 429 should be returned

    //   retryAfter int - seconds until the window resets (used in Retry-After header)

    //   err error      - non-nil only on infrastructure failure (Redis down); caller logs + allows

    Allow(ctx context.Context, key string, limit int, windowSecs int) (allowed bool, retryAfter int, err error)

}
```

GO

```go
// config.go (gateway, extended from M1 stub)

package gateway

import (

    "fmt"

    "os"

    "time"

)

// RateLimitConfig holds one named rate limit policy.

type RateLimitConfig struct {

    Limit      int // requests per window

    WindowSecs int // window duration in seconds

}

type Config struct {

    URLServiceURL          string // required

    AnalyticsServiceURL    string // required

    UserServiceURL         string // required

    NotificationServiceURL string // required

    RedisURL               string // required (rate limit store)

    JWTSecret              string // required (local verification)

    Port                   string // default "8080"

    ServiceName            string // constant "gateway"

    // Circuit breaker settings for url-service:

    CBMaxFailures   int           // default 5

    CBOpenTimeout   time.Duration // default 30s

    CBFailureWindow time.Duration // default 10s

    // Rate limit policies (keyed by route identifier):

    RateLimits map[string]RateLimitConfig

    // Populated at loadConfig() as:

    //   "shorten":  {Limit: 10, WindowSecs: 60}

    //   "redirect": {Limit: 300, WindowSecs: 60}

}

func loadConfig() (*Config, error) {

    cfg := &Config{

        URLServiceURL:          os.Getenv("URL_SERVICE_URL"),

        AnalyticsServiceURL:    os.Getenv("ANALYTICS_SERVICE_URL"),

        UserServiceURL:         os.Getenv("USER_SERVICE_URL"),

        NotificationServiceURL: os.Getenv("NOTIFICATION_SERVICE_URL"),
```

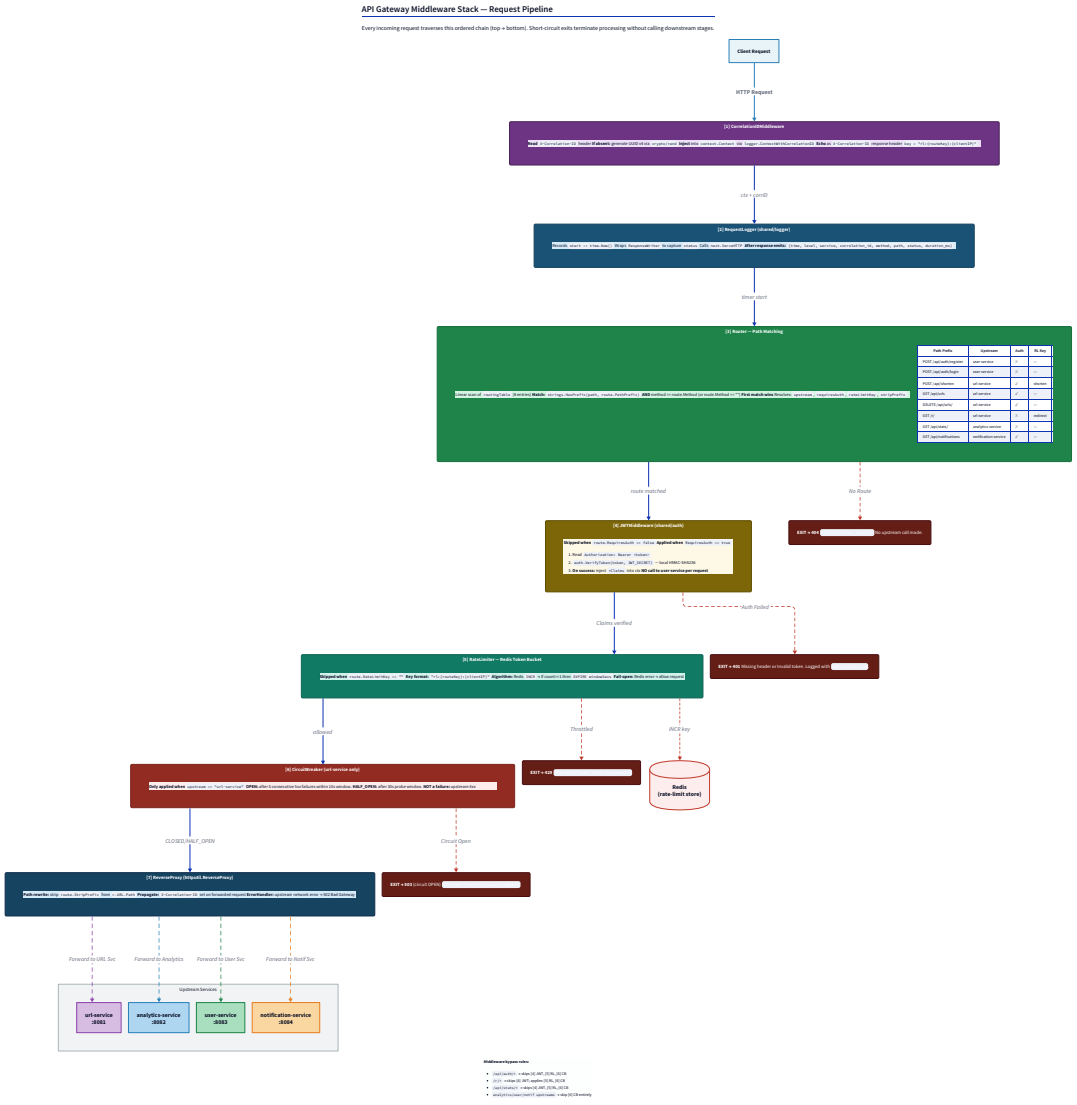
```
        RedisURL:               os.Getenv("REDIS_URL"),
        JWTSecret:              os.Getenv("JWT_SECRET"),
        Port:                   envOrDefault("PORT", "8080"),
        ServiceName:            "gateway",
        CBMaxFailures:          5,
        CBOpenTimeout:          30 * time.Second,
        CBFailureWindow:        10 * time.Second,
        RateLimits: map[string]RateLimitConfig{
            "shorten":  {Limit: 10, WindowSecs: 60},
            "redirect": {Limit: 300, WindowSecs: 60},
        },
    }
    required := map[string]string{
        "URL_SERVICE_URL":          cfg.URLServiceURL,
        "ANALYTICS_SERVICE_URL":    cfg.AnalyticsServiceURL,
        "USER_SERVICE_URL":         cfg.UserServiceURL,
        "NOTIFICATION_SERVICE_URL": cfg.NotificationServiceURL,
        "REDIS_URL":                cfg.RedisURL,
        "JWT_SECRET":               cfg.JWTSecret,
    }
    for k, v := range required {
        if v == "" {
            return nil, fmt.Errorf("%s is required", k)
        }
    }
    return cfg, nil
}
func envOrDefault(key, def string) string {
    if v := os.Getenv(key); v != "" {
        return v
    }
    return def
}
```

3.5 Routing Table (Immutable After Construction)

```go
// router.go — constructed once in main.go and never mutated

var routingTable = []Route{

    // Auth routes — no JWT required

    {Method: "POST", PathPrefix: "/api/auth/register", Upstream: "user-service",         RequiresAuth: false},

    {Method: "POST", PathPrefix: "/api/auth/login",    Upstream: "user-service",         RequiresAuth: false},

    // URL service routes — auth required

    {Method: "POST", PathPrefix: "/api/shorten",       Upstream: "url-service",          RequiresAuth: true,  RateLimitKey:
"shorten",  StripPrefix: "/api"},

    {Method: "GET",  PathPrefix: "/api/urls",          Upstream: "url-service",          RequiresAuth: true,  StripPrefix:
"/api"},

    {Method: "DELETE",PathPrefix:"/api/urls/",         Upstream: "url-service",          RequiresAuth: true,  StripPrefix:
"/api"},

    // Redirect — no auth, rate limited

    {Method: "GET",  PathPrefix: "/r/",               Upstream: "url-service",          RequiresAuth: false, RateLimitKey:
"redirect"},

    // Analytics — no auth

    {Method: "GET",  PathPrefix: "/api/stats/",        Upstream: "analytics-service",    RequiresAuth: false, StripPrefix:
"/api"},

    // Notifications — auth required

    {Method: "GET",  PathPrefix: "/api/notifications", Upstream: "notification-service", RequiresAuth: true,  StripPrefix:
"/api"},

}
```

Path rewriting rule: When `StripPrefix` is non-empty, remove that prefix from `r.URL.Path` before forwarding. Example: `GET /api/shorten` → url-service receives `GET /shorten`. `GET /r/abc1234` → url-service receives `GET /r/abc1234` (no strip). `GET /api/stats/abc1234` → analytics-service receives `GET /stats/abc1234`. **Redirect path rewrite:** url-service internally handles `GET /{code}` (7-char codes). The gateway exposes `GET /r/{code}` and rewrites to `GET /{code}`:

```go
// In proxy forwarding for redirect route: strip "/r" prefix

// /r/abc1234 → /abc1234
```




```
Gateway routing table (prefix matching, first match wins):
  Incoming path                          → Upstream service      → Forwarded path
  ──────────────────────────────────────────────────────────────────────────────
  POST /api/auth/register                → user-service          → /register
  POST /api/auth/login                   → user-service          → /login
  POST /api/shorten          [auth+rl]   → url-service           → /shorten
  GET  /api/urls             [auth]      → url-service           → /urls
  DELETE /api/urls/*         [auth]      → url-service           → /urls/*
  GET  /r/*                  [rl]        → url-service           → /*
  GET  /api/stats/*                      → analytics-service     → /stats/*
  GET  /api/notifications    [auth]      → notification-service  → /notifications
  GET  /health                           → gateway itself        → (no proxy)
  * anything else                        → 404
  [auth] = JWT middleware applied
  [rl]   = rate limiter applied
```

3.6 Extended `shared/logger` Package

```go
// shared/logger/logger.go (extended)

package logger

import (

    "context"

    "log/slog"

    "net/http"

    "os"

    "time"

)

type correlationKey struct{}

// New returns a *slog.Logger writing JSON to stdout with "service" pre-attached.

func New(serviceName string) *slog.Logger {

    h := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{Level: slog.LevelInfo})

    return slog.New(h).With("service", serviceName)

}

// WithCorrelationID returns a derived logger with "correlation_id" attached.

// Used in handlers to bind the correlation ID to all log lines for a request.

func WithCorrelationID(log *slog.Logger, correlationID string) *slog.Logger {

    return log.With("correlation_id", correlationID)

}

// CorrelationIDFromContext retrieves the correlation ID stored by CorrelationIDMiddleware.

// Returns "" if not set.

func CorrelationIDFromContext(ctx context.Context) string {

    if id, ok := ctx.Value(correlationKey{}).(string); ok {

        return id

    }

    return ""

}

// ContextWithCorrelationID stores a correlation ID into a context.

func ContextWithCorrelationID(ctx context.Context, id string) context.Context {

    return context.WithValue(ctx, correlationKey{}, id)

}

// RequestLogger returns middleware that logs each HTTP request as a structured JSON line.

// Emits: time, level, service, correlation_id, method, path, status, duration_ms, msg="request"

// Applied at the outermost layer in the gateway; inner services apply it similarly.
```

```go
func RequestLogger(log *slog.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()
            corrID := CorrelationIDFromContext(r.Context())
            rw := &responseWriter{ResponseWriter: w, status: http.StatusOK}
            next.ServeHTTP(rw, r)
            log.Info("request",
                "correlation_id", corrID,
                "method",         r.Method,
                "path",           r.URL.Path,
                "status",         rw.status,
                "duration_ms",    time.Since(start).Milliseconds(),
            )
        })
    }
}
// responseWriter wraps http.ResponseWriter to capture the status code.
type responseWriter struct {
    http.ResponseWriter
    status int
}
func (rw *responseWriter) WriteHeader(status int) {
    rw.status = status
    rw.ResponseWriter.WriteHeader(status)
}
```

4. Interface Contracts

4.1 `NotificationRepository` — `pgxNotificationStore` Implementation

```go
// store.go

type pgxNotificationStore struct {

    pool *pgxpool.Pool

}

func NewNotificationStore(pool *pgxpool.Pool) NotificationRepository {

    return &pgxNotificationStore{pool: pool}

}
```

`Insert(ctx context.Context, rec *NotificationRecord) (*NotificationRecord, error)`

```
Input:  rec.UserID, rec.EventType, rec.Payload, rec.Status="pending"
        (rec.ID, rec.CreatedAt, rec.SentAt are populated by DB / function)
Algorithm:
  tx, err := s.pool.Begin(ctx)
  if err: return nil, fmt.Errorf("begin notification tx: %w", err)
  Step 1: INSERT with status="pending"
    SQL:
      INSERT INTO notifications (user_id, event_type, payload, status)
      VALUES ($1, $2, $3, 'pending')
      RETURNING id, user_id, event_type, payload, status, created_at, sent_at
    Params: rec.UserID, rec.EventType, rec.Payload
  Step 2: Call mockEmail(rec) — log the mock send
    // mockEmail logs: log.Info("would send email to <user>: <message>")
    // Always succeeds (no real network call)
  Step 3: UPDATE status to "sent", set sent_at = now()
    SQL:
      UPDATE notifications
      SET status = 'sent', sent_at = now()
      WHERE id = $1
      RETURNING sent_at
    Param: inserted.ID
  Step 4: COMMIT
    if err: tx.Rollback(ctx); return nil, fmt.Errorf("commit notification tx: %w", err)
  Step 5: Return populated *NotificationRecord (all fields from RETURNING + updated SentAt)
On any INSERT error:
  tx.Rollback(ctx)
  return nil, fmt.Errorf("insert notification: %w", err)
On UPDATE error:
  tx.Rollback(ctx)
  return nil, fmt.Errorf("update notification status: %w", err)
```

`FindByUserID(ctx context.Context, userID, afterID string, limit int) ([]*NotificationRecord, string, error)`

```
Cursor-based pagination — same pattern as url-service GET /urls.
Default limit: 20. Max enforced by caller.
When afterID == "":
  SQL:
    SELECT id, user_id, event_type, payload, status, created_at, sent_at
    FROM notifications
    WHERE user_id = $1
    ORDER BY created_at DESC, id DESC
    LIMIT $2
  Params: userID, limit+1
When afterID != "":
  SQL:
    SELECT id, user_id, event_type, payload, status, created_at, sent_at
    FROM notifications
    WHERE user_id = $1
      AND (created_at, id) < (
          SELECT created_at, id FROM notifications WHERE id = $2
      )
    ORDER BY created_at DESC, id DESC
    LIMIT $3
  Params: userID, afterID, limit+1
Next-cursor logic:
  Fetch limit+1 rows.
  If len(rows) == limit+1: nextCursor = rows[limit].ID; return rows[:limit]
  Else: nextCursor = ""; return all rows
On success: ([]*NotificationRecord, nextCursor string, nil)
On error:   nil, "", fmt.Errorf("find notifications by user: %w", err)
```

4.2 `NotificationConsumer`

```go
// consumer.go

type NotificationConsumer struct {

    conn  *RabbitMQConn

    pool  *pgxpool.Pool

    store NotificationRepository

    log   *slog.Logger

}

func NewNotificationConsumer(

    conn  *RabbitMQConn,

    pool  *pgxpool.Pool,

    store NotificationRepository,

    log   *slog.Logger,

) *NotificationConsumer

// Run starts the consumer loop. Blocks until ctx is Done.

func (c *NotificationConsumer) Run(ctx context.Context)
```

`Run(ctx context.Context)` — full algorithm:

```
Step 1: Set QoS prefetch
  c.conn.Channel.Qos(1, 0, false)
  On error: log.Error + os.Exit(1)
Step 2: Consume from "notifications.events"
  msgs, err := c.conn.Channel.Consume(
      "notifications.events",
      "",     // auto consumer tag
      false,  // autoAck = false
      false, false, false, nil,
  )
  On error: log.Error + os.Exit(1)
Step 3: Message loop
  for:
    select:
    case <-ctx.Done(): log.Info("notification consumer stopped"); return
    case d, ok := <-msgs:
      if !ok:
        log.Warn("notification amqp channel closed")
        <-ctx.Done()
        return
      c.processDelivery(ctx, d)
```

`processDelivery(ctx context.Context, d amqp.Delivery)` — full algorithm:

```
Step 1: Panic recovery
  defer func() {
    if r := recover(); r != nil:
      log.Error("notification consumer panic", "panic", fmt.Sprintf("%v", r))
      d.Ack(false)
  }()
Step 2: Determine event type from routing key
  eventType := d.RoutingKey
  if eventType is not one of {"url.created","url.deleted","milestone.reached"}:
    log.Warn("unknown routing key", "key", eventType)
    d.Ack(false); return
Step 3: Extract user_id and user_email from payload
  // Route-specific unmarshaling to extract user context without calling other services.
  var userID, userEmail, message string
  switch eventType:
  case "url.created":
    var evt events.URLCreatedEvent
    if err := json.Unmarshal(d.Body, &evt); err != nil:
      log.Error("poison: unmarshal URLCreatedEvent", "error", err,
                "body_preview", truncate(d.Body, 200))
      d.Ack(false); return
    userID    = evt.UserID
    userEmail = evt.UserEmail
    message   = fmt.Sprintf("Your short URL %s has been created for %s", evt.ShortCode, evt.OriginalURL)
    correlationID = evt.CorrelationID
  case "url.deleted":
    var evt events.URLDeletedEvent
    if err := json.Unmarshal(d.Body, &evt); err != nil:
      log.Error("poison: unmarshal URLDeletedEvent", ...)
      d.Ack(false); return
    userID    = evt.UserID
    userEmail = evt.UserEmail
    message   = fmt.Sprintf("Your short URL %s has been deactivated", evt.ShortCode)
    correlationID = evt.CorrelationID
  case "milestone.reached":
    var evt events.MilestoneReachedEvent
    if err := json.Unmarshal(d.Body, &evt); err != nil:
      log.Error("poison: unmarshal MilestoneReachedEvent", ...)
      d.Ack(false); return
    userID    = evt.UserID
    userEmail = evt.UserEmail
    message   = fmt.Sprintf("Your short URL %s has reached %d clicks!", evt.ShortCode, evt.MilestoneN)
    correlationID = evt.CorrelationID
Step 4: Validate required fields
  if userID == "":
    log.Error("notification: missing user_id in event", "event_type", eventType)
    d.Ack(false); return
  // userEmail may be "" for milestone events from incomplete event payloads; proceed
Step 5: Build and insert notification record
  rec := &NotificationRecord{
    UserID:    userID,
    EventType: eventType,
    Payload:   d.Body,
    Status:    "pending",
  }
  saved, err := c.store.Insert(ctx, rec)
  if err != nil:
    log.Error("notification insert failed", "error", err, "event_type", eventType)
    d.Nack(false, true)  // requeue for retry
    return
Step 6: Mock email log
  // mockEmail is called inside store.Insert before commit.
  // Log here as confirmation with correlation_id for trace:
  log.Info("would send email to user",
    "user_id",        userID,
    "user_email",     userEmail,
    "message",        message,
    "notification_id",saved.ID,
    "correlation_id", correlationID,
  )
Step 7: Acknowledge
  d.Ack(false)
  log.Info("notification processed",
```

```
    "notification_id", saved.ID,
    "event_type",      eventType,
    "correlation_id",  correlationID,
  )
```

4.3 `NotificationHandler`

```go
// handler.go

type NotificationHandler struct {

    store NotificationRepository

    log   *slog.Logger

}

func NewNotificationHandler(store NotificationRepository, log *slog.Logger) *NotificationHandler

// ListNotifications handles GET /notifications.

// Requires JWT. Returns caller's notifications, newest-first, paginated.

// Query params: after=<uuid> (cursor), limit=<int> (default 20, max 50)

func (h *NotificationHandler) ListNotifications(w http.ResponseWriter, r *http.Request)
```

`ListNotifications` Algorithm:

```
Step 1: Extract claims
  claims, ok := auth.ClaimsFromContext(r.Context())
  if !ok: writeError(w, 401, "unauthorized"); return
Step 2: Parse query params
  afterID := r.URL.Query().Get("after")
  limit   := 20
  if limitStr := r.URL.Query().Get("limit"); limitStr != "":
    n, err := strconv.Atoi(limitStr)
    if err != nil || n < 1 || n > 50:
      writeError(w, 400, "limit must be integer 1-50"); return
    limit = n
Step 3: Fetch notifications
  recs, nextCursor, err := h.store.FindByUserID(r.Context(), claims.Sub, afterID, limit)
  if err != nil:
    h.log.Error("find notifications failed", "error", err, "user_id", claims.Sub)
    writeError(w, 500, "internal server error"); return
Step 4: Build response items
  items := make([]notificationItem, 0, len(recs))
  for _, rec := range recs:
    var rawPayload any
    json.Unmarshal(rec.Payload, &rawPayload)  // re-hydrate JSONB for embedding
    item := notificationItem{
      ID:        rec.ID,
      EventType: rec.EventType,
      Payload:   rawPayload,
      Status:    rec.Status,
      CreatedAt: rec.CreatedAt.UTC().Format(time.RFC3339),
    }
    if rec.SentAt != nil:
      s := rec.SentAt.UTC().Format(time.RFC3339)
      item.SentAt = &s
    items = append(items, item)
Step 5: Build next_cursor pointer
  var nextCursorPtr *string
  if nextCursor != "":
    nextCursorPtr = &nextCursor
Step 6: Write response
  writeJSON(w, 200, notificationListResponse{
    Notifications: items,
    NextCursor:    nextCursorPtr,
  })
```

4.4 `CircuitBreaker`

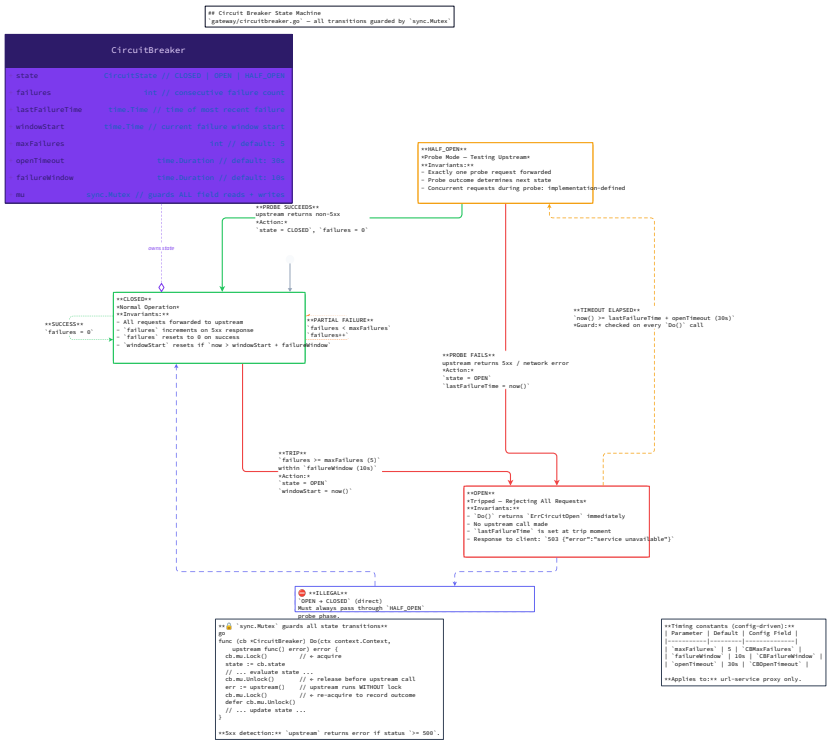
```go
// circuitbreaker.go

// NewCircuitBreaker constructs a CircuitBreaker with explicit configuration.
func NewCircuitBreaker(maxFailures int, openTimeout, failureWindow time.Duration) *CircuitBreaker {
    return &CircuitBreaker{
        state:          StateClosed,
        maxFailures:    maxFailures,
        openTimeout:    openTimeout,
        failureWindow:  failureWindow,
        windowStart:    time.Now(),
    }
}
```

`Do(ctx context.Context, upstream func() error) error` This is the primary entry point. The gateway calls `cb.Do(ctx, func() error { return proxy.forward(req, resp) })`.

```
func (cb *CircuitBreaker) Do(ctx context.Context, upstream func() error) error:
Step 1: Acquire lock and evaluate state
  cb.mu.Lock()
  switch cb.state:
  case StateOpen:
    // Check if openTimeout has elapsed since last failure
    if time.Since(cb.lastFailureTime) >= cb.openTimeout:
      cb.state = StateHalfOpen
      cb.mu.Unlock()
      // fall through to execute probe request (StateHalfOpen case)
    else:
      cb.mu.Unlock()
      return ErrCircuitOpen  // 503 immediately, no upstream call
  case StateHalfOpen:
    // Allow exactly one probe request through.
    // Do NOT release any other concurrent requests — they would see StateHalfOpen
    // and spin-wait or fail. With net/http single-request model, this is safe:
    // set state to something that blocks others.
    // Simple approach: transition to StateClosed optimistically; rollback on failure.
    // This is acceptable for single-gateway, low concurrency probe scenarios.
    cb.mu.Unlock()
    // execute probe below
  case StateClosed:
    // Reset failure window if window has expired
    if time.Since(cb.windowStart) > cb.failureWindow:
      cb.failures    = 0
      cb.windowStart = time.Now()
    cb.mu.Unlock()
    // execute request below
Step 2: Execute upstream function
  err := upstream()
Step 3: Record outcome
  cb.mu.Lock()
  defer cb.mu.Unlock()
  if err == nil:
    // Success
    if cb.state == StateHalfOpen:
      cb.state    = StateClosed
      cb.failures = 0
      cb.log.Info("circuit breaker CLOSED (probe succeeded)")
    else if cb.state == StateClosed:
      cb.failures = 0  // reset consecutive failures on success
    return nil
  // Failure
  cb.lastFailureTime = time.Now()
  if cb.state == StateHalfOpen:
    cb.state = StateOpen
    cb.log.Warn("circuit breaker OPEN (probe failed)")
    return err
  // StateClosed failure
  cb.failures++
  if cb.failures >= cb.maxFailures:
    cb.state       = StateOpen
    cb.windowStart = time.Now()  // reset window for next open period
    cb.log.Warn("circuit breaker OPEN",
      "failures", cb.failures,
      "max_failures", cb.maxFailures,
    )
  return err
// ErrCircuitOpen is returned when the breaker is open and rejects the request.
var ErrCircuitOpen = errors.New("circuit breaker open")
```

What counts as a failure: The `upstream` function passed to `Do` must return a non-nil error on any of the following conditions from the proxied response:

- HTTP 5xx response from upstream (500, 502, 503, 504)
- Network timeout or connection refused
- Context deadline exceeded HTTP 4xx responses (400, 401, 403, 404, 422, 429) are **not** failures — they indicate the upstream is healthy and responding correctly.




```
Circuit breaker state machine:

  ┌──────────────────────────────────────────────────────────┐
  │                                                          │
  │    ┌──────────┐                                          │
  │    │  CLOSED  │◄──────────────────────────────┐          │
  │    │ (normal) │                               │          │
  │    └──────┬───┘                               │          │
  │           │                                   │          │
  │   failures >= maxFailures                probe success   │
  │   within failureWindow (10s)                  │          │
  │           │                                   │          │
  │           ▼                                   │          │
  │    ┌──────────┐    openTimeout (30s) elapsed  ┌┴─────────┐│
  │    │   OPEN   │──────────────────────────────►│HALF_OPEN ││
  │    │  (trip)  │                               │ (probe)  ││
  │    └──────────┘                               └──────────┘│
  │           ▲                                   │          │
  │           └──────────────── probe failure ────┘          │
  │                                                          │
  │  CLOSED:    requests pass through; failures counted      │
  │  OPEN:      requests rejected immediately → 503          │
  │  HALF_OPEN: one probe request allowed; success→CLOSED,   │
  │             failure→OPEN                                 │
  └──────────────────────────────────────────────────────────┘

  Configuration: maxFailures=5, failureWindow=10s, openTimeout=30s
  All state transitions guarded by sync.Mutex.
  ILLEGAL transitions: CLOSED→HALF_OPEN, OPEN→CLOSED (must go through HALF_OPEN)
```

4.5 `RateLimiter` — `redisTokenBucket` Implementation

```go
// ratelimit.go

type redisTokenBucket struct {

    client *redis.Client

    log    *slog.Logger

}

func NewRateLimiter(client *redis.Client, log *slog.Logger) RateLimiter {

    return &redisTokenBucket{client: client, log: log}

}
```

`Allow(ctx context.Context, key string, limit int, windowSecs int) (bool, int, error)`

```
Algorithm — uses two-command Redis pipeline (INCR + EXPIRE):
Step 1: Build Redis key
  redisKey := "rl:" + key   // e.g. "rl:shorten:192.168.1.1"
Step 2: INCR the counter (creates key with value 1 if absent)
  count, err := c.client.Incr(ctx, redisKey).Result()
  if err != nil:
    c.log.Warn("rate limit redis incr failed", "key", redisKey, "error", err)
    return true, 0, err  // fail-open: allow the request
Step 3: Set expiry only on the first increment (count == 1)
  // Setting EXPIRE on every request would reset the window on each hit.
  // Setting only on count==1 means the window starts with the first request
  // and expires exactly windowSecs later. Subsequent requests in the window
  // increment but do not reset the expiry.
  if count == 1:
    expireErr := c.client.Expire(ctx, redisKey, time.Duration(windowSecs)*time.Second).Err()
    if expireErr != nil:
      c.log.Warn("rate limit redis expire failed", "key", redisKey, "error", expireErr)
      // Non-fatal: key will exist without TTL. INCR will keep counting but never expire.
      // Accept this edge case: the counter resets on Redis restart anyway.
Step 4: Calculate retryAfter
  // Approximate: TTL of the current window.
  // For simplicity, use windowSecs as retryAfter (conservative upper bound).
  retryAfter := windowSecs
Step 5: Check limit
  if count > int64(limit):
    return false, retryAfter, nil  // 429
  return true, 0, nil              // allowed
Note on atomicity: INCR is atomic in Redis. The race condition where two concurrent
requests both see count==1 and both call EXPIRE is harmless — both EXPIRE calls
set the same TTL on the same key. The window boundary is accurate.
Note on key structure:
  "shorten" route:  "rl:shorten:<client_ip>"
  "redirect" route: "rl:redirect:<client_ip>"
  IP extraction:    net.SplitHostPort(r.RemoteAddr) or r.Header.Get("X-Forwarded-For")
                    (for this project: use RemoteAddr; no load balancer in front)
```

4.6 Gateway `ReverseProxy`

```go
// proxy.go

package gateway

import (

    "net/http"

    "net/http/httputil"

    "net/url"

)

// UpstreamProxy wraps httputil.ReverseProxy for one upstream service.

type UpstreamProxy struct {

    proxy    *httputil.ReverseProxy

    upstream string  // service name for logging

    log      *slog.Logger

}

// NewUpstreamProxy constructs a reverse proxy targeting baseURL.

func NewUpstreamProxy(baseURL string, log *slog.Logger, upstream string) (*UpstreamProxy, error) {

    target, err := url.Parse(baseURL)

    if err != nil {

        return nil, fmt.Errorf("parse upstream URL %q: %w", baseURL, err)

    }

    rp := httputil.NewSingleHostReverseProxy(target)

    // Custom error handler: capture upstream 5xx for circuit breaker

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

        log.Warn("upstream error", "upstream", upstream, "error", err,

            "correlation_id", logger.CorrelationIDFromContext(r.Context()))

        http.Error(w, `{"error":"upstream error"}`, http.StatusBadGateway)

    }

    return &UpstreamProxy{proxy: rp, upstream: upstream, log: log}, nil

}

// ServeHTTP forwards the request to the upstream. Strips the given prefix before forwarding.

func (p *UpstreamProxy) ServeHTTP(w http.ResponseWriter, r *http.Request, stripPrefix, addPath string) {

    // Rewrite path: strip gateway-level prefix, optionally rewrite

    if stripPrefix != "" {

        r.URL.Path = strings.TrimPrefix(r.URL.Path, stripPrefix)

    }

    if addPath != "" {
```

```
        r.URL.Path = addPath + r.URL.Path
    }
    if r.URL.Path == "" {
        r.URL.Path = "/"
    }
    p.proxy.ServeHTTP(w, r)
}
```

How the circuit breaker wraps the proxy:

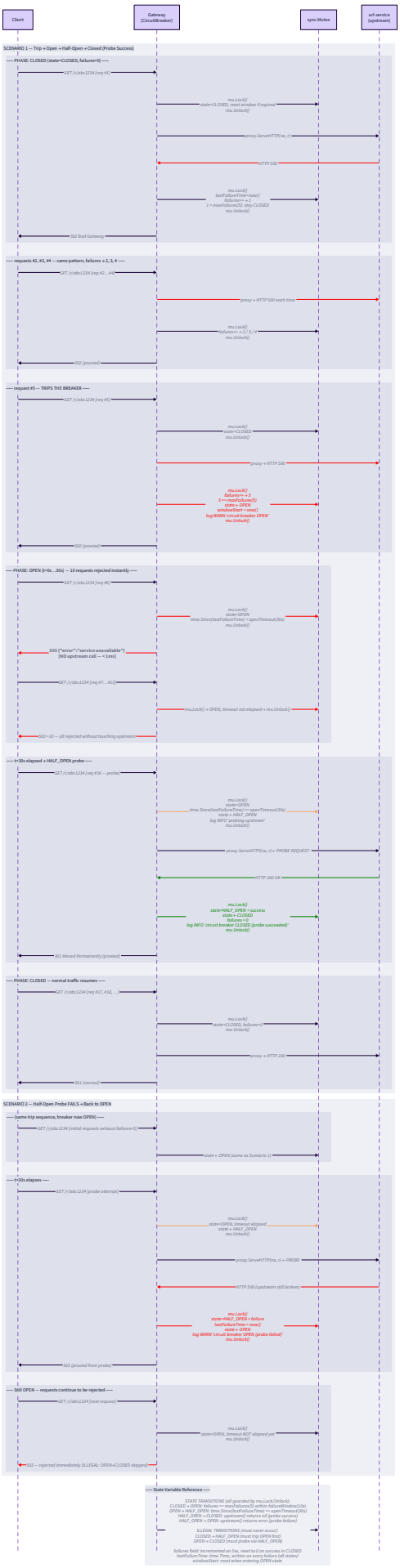
```go
// In the gateway handler — url-service only gets the circuit breaker:
func (g *Gateway) handleURLService(w http.ResponseWriter, r *http.Request, route Route) {
    rw := &responseWriter{ResponseWriter: w, status: 200}
    err := g.urlServiceCB.Do(r.Context(), func() error {
        g.urlProxy.ServeHTTP(rw, r, route.StripPrefix, "")
        // Count 5xx as circuit breaker failures:
        if rw.status >= 500 {
            return fmt.Errorf("upstream 5xx: %d", rw.status)
        }
        return nil
    })
    if errors.Is(err, ErrCircuitOpen):
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusServiceUnavailable)
        w.Write([]byte(`{"error":"service unavailable"}`))
        return
    // Other errors are already written by ErrorHandler
}
```

5. Algorithm Specification

5.1 Gateway Request Processing Pipeline

Every incoming request passes through this ordered middleware chain:

```
Incoming HTTP Request
        |
        ▼
1. CorrelationIDMiddleware
   - Read X-Correlation-ID header
   - If absent: generate UUID v4
   - Store in request context
   - Echo as X-Correlation-ID response header
        |
        ▼
2. RequestLogger (shared/logger.RequestLogger)
   - Record start time
   - Wrap ResponseWriter to capture status code
   - Call next handler
   - After response: log {time, level, service, correlation_id, method, path, status, duration_ms}
        |
        ▼
3. Router.ServeHTTP
   - Match request against routingTable (first match wins)
   - If no match: 404 {"error":"not found"}
   - Determine: upstream, requiresAuth, rateLimitKey, stripPrefix
        |
        ▼
4. [If requiresAuth] JWTMiddleware
   - Read Authorization: Bearer <token>
   - Verify HMAC-SHA256 locally (shared secret)
   - On failure: 401 {"error":"unauthorized"} — stop processing
   - On success: inject Claims into context
        |
        ▼
5. [If rateLimitKey != ""] RateLimiter.Allow
   - key = "rl:{rateLimitKey}:{clientIP}"
   - Redis INCR + conditional EXPIRE
   - If count > limit: 429 {"error":"rate limit exceeded"} + Retry-After header — stop
   - On Redis error: log Warn + allow (fail-open)
        |
        ▼
6. Forward X-Correlation-ID to upstream
   - r.Header.Set("X-Correlation-ID", correlationIDFromContext(r.Context()))
        |
        ▼
7. [If upstream == "url-service"] CircuitBreaker.Do(upstream func)
   - If OPEN: 503 {"error":"service unavailable"} — stop
   - If CLOSED/HALF_OPEN: execute proxy forward
   - Record success or failure based on response status
        |
        ▼
8. [All other upstreams] Direct proxy forward
   - httputil.ReverseProxy.ServeHTTP
   - Path rewrite per route.StripPrefix
        |
        ▼
   Upstream response → client
```


Client
Gateway (CircuitBreaker)
sync.Mutex
url-service (upstream)
SCENARIO 1 — Trip → Open → Half-Open → Closed (Probe Success)
── PHASE: CLOSED (state=CLOSED, failures=0) ──
GET /r/abc1234 [req #1]
mu.Lock()
state=CLOSED, reset window if expired
mu.Unlock()
proxy.ServeHTTP(rw, r)
HTTP 502
mu.Lock()
lastFailureTime=now()
failures++ → 1
1 < maxFailures(5): stay CLOSED
mu.Unlock()
502 Bad Gateway
── requests #2, #3, #4 — same pattern, failures → 2, 3, 4 ──
GET /r/abc1234 [req #2…#4]
proxy → HTTP 500 each time
mu.Lock()
failures++ → 2 / 3 / 4
mu.Unlock()
502 (proxied)
── request #5 — TRIPS THE BREAKER ──
GET /r/abc1234 [req #5]
mu.Lock()
state=CLOSED
mu.Unlock()
proxy → HTTP 502
mu.Lock()
failures++ → 5
5 >= maxFailures(5)
state = OPEN
windowStart = now()
log.WARN 'circuit breaker OPEN'
mu.Unlock()
502 (proxied)
── PHASE: OPEN (t=0s…30s) — 10 requests rejected instantly ──
GET /r/abc1234 [req #6]
mu.Lock()
state=OPEN
time.Since(lastFailureTime) < openTimeout(30s)
mu.Unlock()
503 {"error":"service unavailable"}
[NO upstream call — ~1ms]
GET /r/abc1234 [req #7…#15]
mu.Lock() + OPEN, timeout not elapsed + mu.Unlock()
503 ×10 — all rejected without touching upstream
── t=30s elapsed + HALF_OPEN probe ──
GET /r/abc1234 [req #16 — probe]
mu.Lock()
state=OPEN
time.Since(lastFailureTime) >= openTimeout(30s)
state = HALF_OPEN
log.INFO 'probing upstream'
mu.Unlock()
proxy.ServeHTTP(rw, r) ← PROBE REQUEST
HTTP 200 OK
mu.Lock()
state=HALF_OPEN + success
state = CLOSED
failures = 0
log.INFO 'circuit breaker CLOSED (probe succeeded)'
mu.Unlock()
301 Moved Permanently (proxied)
── PHASE: CLOSED — normal traffic resumes ──
GET /r/abc1234 [req #17, #18, …]
mu.Lock()
state=CLOSED, failures=0
mu.Unlock()
proxy → HTTP 200
301 (normal)
SCENARIO 2 — Half-Open Probe FAILS → Back to OPEN
── (same trip sequence, breaker now OPEN) ──
GET /r/abc1234 [initial requests exhaust failures=5]
state = OPEN (same as Scenario 1)
── t=30s elapses ──
GET /r/abc1234 [probe attempt]
mu.Lock()
state=OPEN, timeout elapsed
state = HALF_OPEN
mu.Unlock()
proxy.ServeHTTP(rw, r) ← PROBE
HTTP 530 (upstream still broken)
mu.Lock()
state=HALF_OPEN + failure
lastFailureTime = now()
state = OPEN
log.WARN 'circuit breaker OPEN (probe failed)'
mu.Unlock()
502 (proxied from probe)
── Still OPEN — requests continue to be rejected ──
GET /r/abc1234 [next request]
mu.Lock()
state=OPEN, timeout NOT elapsed yet
mu.Unlock()
503 — rejected immediately (ILLEGAL: OPEN→CLOSED skipped)
── State Variable Reference ──
STATE TRANSITIONS (all guarded by mu.Lock/Unlock):
CLOSED → OPEN: failures >= maxFailures(5) within failureWindow(30s)
OPEN → HALF_OPEN: time.Since(lastFailureTime) >= openTimeout(30s)
HALF_OPEN → CLOSED: upstream(r) returns non-error (probe success)
HALF_OPEN → OPEN: upstream(r) returns error (probe failure)
ILLEGAL TRANSITIONS (must never occur):
CLOSED → HALF_OPEN (must trip OPEN first)
OPEN → CLOSED (must probe via HALF_OPEN)
failures field: incremented on 5xx, reset to 0 on success in CLOSED
lastFailureTime: time.Time, written on every failure (all states)
windowStart: reset when entering OPEN state

```
Gateway middleware chain (layered decorators):
  ┌──────────────────────────────────────────────────────────┐
  │  net/http server                                         │
  │                                                          │
  │  handler := chain(                                       │
  │    CorrelationIDMiddleware,      ← outermost (always runs)   │
  │    logger.RequestLogger(log),   ← second (captures timing)  │
  │    router.ServeHTTP,            ← matches route          │
  │  )                                                       │
  │                                                          │
  │  Per-route (inside router):                              │
  │    if requiresAuth:  JWTMiddleware → 401 on fail         │
  │    if rateLimitKey:  RateLimiter → 429 on exceeded       │
  │    add correlation header                                │
  │    if upstream=url-service: CircuitBreaker.Do(proxy)     │
  │    else: direct proxy                                    │
  └──────────────────────────────────────────────────────────┘
```

5.2 `CorrelationIDMiddleware` Implementation

GO

```go
// middleware.go

package gateway

import (

    "net/http"

    "github.com/yourhandle/url-shortener/shared/logger"

)

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

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

        id := r.Header.Get("X-Correlation-ID")

        if id == "" {

            id = newUUID()  // crypto/rand UUID v4 (same helper as other services)

        }

        ctx := logger.ContextWithCorrelationID(r.Context(), id)

        w.Header().Set("X-Correlation-ID", id)

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

    })

}

// chain applies middlewares right-to-left (outermost first in call order).

func chain(h http.Handler, middlewares ...func(http.Handler) http.Handler) http.Handler {

    for i := len(middlewares) - 1; i >= 0; i-- {

        h = middlewares[i](h)

    }

    return h

}
```

5.3 Gateway JWT Middleware

```go
// jwtmiddleware.go

package gateway

import (

    "encoding/json"

    "net/http"

    "strings"

    "github.com/yourhandle/url-shortener/shared/auth"

    "github.com/yourhandle/url-shortener/shared/logger"

)

// GatewayJWTMiddleware verifies JWT on routes with requiresAuth=true.

// On success: injects Claims into context for downstream use.

// On failure: writes 401 and does NOT forward to upstream.

// This wraps shared/auth.JWTMiddleware to add gateway-specific logging.

func GatewayJWTMiddleware(secret string, log *slog.Logger) func(http.Handler) http.Handler {

    inner := auth.JWTMiddleware(secret)

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

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

            // shared/auth.JWTMiddleware writes 401 if token invalid.

            // We wrap it to log the rejection with correlation ID.

            rw := &responseWriter{ResponseWriter: w, status: 200}

            inner(next).ServeHTTP(rw, r)

            if rw.status == 401 {

                log.Info("JWT rejected",

                    "correlation_id", logger.CorrelationIDFromContext(r.Context()),

                    "path", r.URL.Path,

                )

            }

        })

    }

}
```

5.4 Rate Limiter — Client IP Extraction

```go
// ratelimit.go

// clientIP extracts the real client IP from a request.

// For this project (no load balancer): uses RemoteAddr directly.

// In production with a reverse proxy, check X-Forwarded-For first.

func clientIP(r *http.Request) string {

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

    if err != nil {

        return r.RemoteAddr

    }

    return ip

}

// rateLimitKey builds the Redis key for a route+IP combination.

// Format: "rl:{routeKey}:{ip}"

func buildRateLimitKey(routeKey, ip string) string {

    return "rl:" + routeKey + ":" + ip

}
```

5.5 Gateway `ServeHTTP` — Full Router Logic

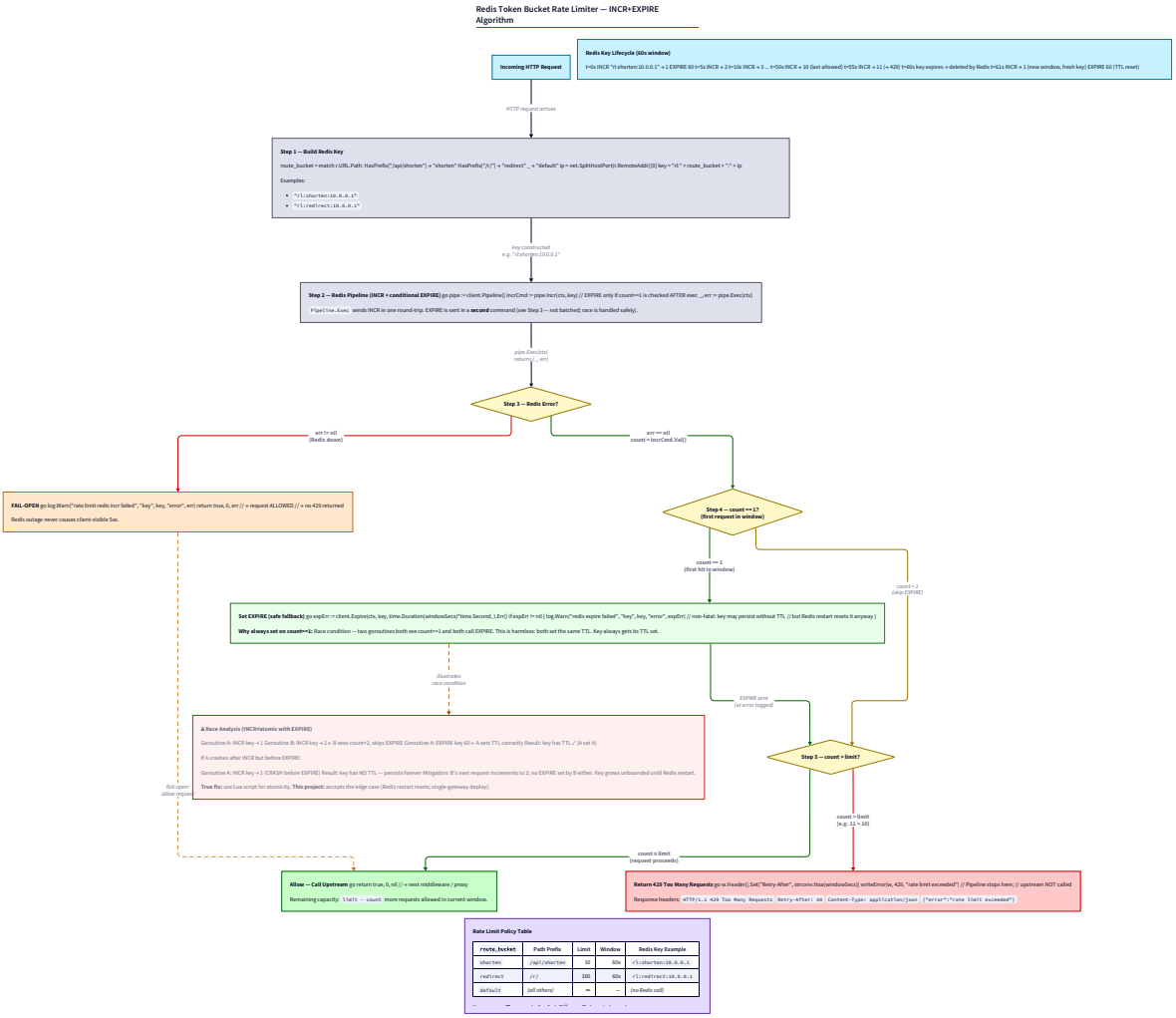
```go
// router.go

type Gateway struct {

    routes         []Route

    proxies        map[string]*UpstreamProxy  // keyed by upstream service name

    urlServiceCB   *CircuitBreaker

    rateLimiter    RateLimiter

    jwtSecret      string

    rateLimits     map[string]RateLimitConfig

    log            *slog.Logger

}

func (g *Gateway) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    // Find matching route

    var matched *Route

    for i := range g.routes {

        rt := &g.routes[i]

        if strings.HasPrefix(r.URL.Path, rt.PathPrefix) {

            if rt.Method == "" || rt.Method == r.Method {

                matched = rt

                break

            }

        }

    }

    if matched == nil {

        writeError(w, 404, "not found")

        return

    }

    // JWT authentication

    if matched.RequiresAuth {

        claims, ok := g.verifyJWT(w, r)

        if !ok {

            return  // 401 already written

        }

        _ = claims  // inject into context for upstream if needed

    }

    // Rate limiting
```

```
    if matched.RateLimitKey != "" {
        rlCfg, exists := g.rateLimits[matched.RateLimitKey]
        if exists {
            ip := clientIP(r)
            key := buildRateLimitKey(matched.RateLimitKey, ip)
            allowed, retryAfter, _ := g.rateLimiter.Allow(r.Context(), key, rlCfg.Limit, rlCfg.WindowSecs)
            if !allowed {
                w.Header().Set("Retry-After", strconv.Itoa(retryAfter))
                writeError(w, 429, "rate limit exceeded")
                return
            }
        }
    }
    // Forward X-Correlation-ID header to upstream
    corrID := logger.CorrelationIDFromContext(r.Context())
    r.Header.Set("X-Correlation-ID", corrID)
    // Route to upstream
    proxy, ok := g.proxies[matched.Upstream]
    if !ok {
        g.log.Error("no proxy for upstream", "upstream", matched.Upstream)
        writeError(w, 500, "internal server error")
        return
    }
    // Apply path rewriting for redirect route
    outPath := r.URL.Path
    if matched.StripPrefix != "" {
        outPath = strings.TrimPrefix(outPath, matched.StripPrefix)
    }
    // Special: /r/{code} → /{code}
    if strings.HasPrefix(r.URL.Path, "/r/") {
        outPath = strings.TrimPrefix(outPath, "/r")
    }
    r2 := r.Clone(r.Context())
    r2.URL.Path = outPath
    if r2.URL.Path == "" { r2.URL.Path = "/" }
    // Circuit breaker for url-service only
```

```
    if matched.Upstream == "url-service" {
        rw := &responseWriter{ResponseWriter: w, status: 200}
        err := g.urlServiceCB.Do(r.Context(), func() error {
            proxy.proxy.ServeHTTP(rw, r2)
            if rw.status >= 500 {
                return fmt.Errorf("upstream 5xx: %d", rw.status)
            }
            return nil
        })
        if errors.Is(err, ErrCircuitOpen) {
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusServiceUnavailable)
            w.Write([]byte(`{"error":"service unavailable"}`))
        }
        return
    }
    // Direct proxy for other upstreams
    proxy.proxy.ServeHTTP(w, r2)
}
```

5.6 Gateway Startup Sequence

```
main.go startup — gateway:
  1. loadConfig()
     → if any required env var empty: fmt.Fprintf(stderr) + os.Exit(1)
  2. log := logger.New(cfg.ServiceName)
  3. Build Redis client for rate limiting:
     opts, err := redis.ParseURL(cfg.RedisURL)
     client := redis.NewClient(opts)
     // Ping with 3s timeout; warn on failure (fail-open)
     pingCtx, cancel := context.WithTimeout(ctx, 3*time.Second)
     if err := client.Ping(pingCtx).Err(); err != nil:
       log.Warn("gateway redis unreachable, rate limiting will fail-open", "error", err)
     cancel()
  4. Build UpstreamProxy for each service:
     proxies := map[string]*UpstreamProxy{
       "url-service":          NewUpstreamProxy(cfg.URLServiceURL, log, "url-service"),
       "analytics-service":    NewUpstreamProxy(cfg.AnalyticsServiceURL, log, "analytics-service"),
       "user-service":         NewUpstreamProxy(cfg.UserServiceURL, log, "user-service"),
       "notification-service": NewUpstreamProxy(cfg.NotificationServiceURL, log, "notification-service"),
     }
  5. Build CircuitBreaker:
     cb := NewCircuitBreaker(cfg.CBMaxFailures, cfg.CBOpenTimeout, cfg.CBFailureWindow)
  6. Build RateLimiter:
     rl := NewRateLimiter(client, log)
  7. Build Gateway:
     gw := &Gateway{
       routes: routingTable,
       proxies: proxies,
       urlServiceCB: cb,
       rateLimiter: rl,
       jwtSecret: cfg.JWTSecret,
       rateLimits: cfg.RateLimits,
       log: log,
     }
  8. Build mux:
     mux := http.NewServeMux()
     mux.HandleFunc("GET /health", NewHealthHandler(cfg.ServiceName))
     mux.Handle("/", chain(
       gw,
       logger.RequestLogger(log),
       CorrelationIDMiddleware,
     ))
  9. Health-check all four downstream services on startup:
     go func():
       for _, upstream := range []struct{name, url string}{
         {"url-service", cfg.URLServiceURL},
         {"analytics-service", cfg.AnalyticsServiceURL},
         {"user-service", cfg.UserServiceURL},
         {"notification-service", cfg.NotificationServiceURL},
       }:
         resp, err := http.Get(upstream.url + "/health")
         if err != nil || resp.StatusCode != 200:
           log.Warn("upstream not healthy at startup",
             "upstream", upstream.name,
             "error", err,
           )
         else:
           log.Info("upstream healthy", "upstream", upstream.name)
     ()
  10. srv.ListenAndServe()
```




```
Gateway wiring diagram:

  Client ──HTTP──▶  ┌──────────────────────────────────┐
                    │           API Gateway            │
                    │                                  │
                    │  CorrelationIDMiddleware         │
                    │  RequestLogger                   │
                    │  Router (routingTable)           │
                    │    │                             │
                    │    ├─[auth]   JWTMiddleware      │  ← shared/auth.VerifyToken
                    │    ├─[rl]     redisTokenBucket   │  ← Redis INCR+EXPIRE
                    │    └─[proxy]                     │
                    │         │                        │
                    │    url-service ──▶ CircuitBreaker│
                    │         │                        │
                    │    analytics-service ──▶ direct  │
                    │    user-service ──▶ direct       │
                    │    notification-service ──▶direct│
                    │                                  │
                    │  Redis ◀──── rate limit keys     │
                    └──────────────────────────────────┘
                                    │
                                (proxied to)
                                    ▼
                    ┌──────────────────────────────────┐
                    │  url-service       :8081         │
                    │  analytics-service :8082         │
                    │  user-service      :8083         │
                    │  notification-svc  :8084         │
                    └──────────────────────────────────┘
```

5.7 Notification Service Startup Sequence

```
main.go startup — notification-service:
  1. loadConfig() → fatal if DATABASE_URL | RABBITMQ_URL | JWT_SECRET missing
  2. log := logger.New(cfg.ServiceName)
  3. ctx, cancel := context.WithCancel(context.Background())
  4. NewDBPool(ctx, cfg.DatabaseURL, log) → fatal on error
  5. runMigrations(ctx, pool, log) → fatal on error
  6. NewRabbitMQConn(ctx, cfg.RabbitMQURL, log, 10) → fatal after 10 attempts
  7. DeclareNotificationQueue(mq.Channel) → fatal on error (from M1)
  8. store := NewNotificationStore(pool)
  9. consumer := NewNotificationConsumer(mq, pool, store, log)
  10. handler := NewNotificationHandler(store, log)
  11. mux := http.NewServeMux()
      mux.HandleFunc("GET /health", NewHealthHandler(cfg.ServiceName))
      mux.Handle("GET /notifications",
        auth.JWTMiddleware(cfg.JWTSecret)(http.HandlerFunc(handler.ListNotifications)))
  12. go consumer.Run(ctx)
  13. srv.ListenAndServe()
  14. On SIGTERM: cancel(); srv.Shutdown(...)
```

5.8 Schema Migration — Notification Service

GO

```go
// main.go

const notificationSchema = `

CREATE TABLE IF NOT EXISTS notifications (

    id         UUID        PRIMARY KEY DEFAULT gen_random_uuid(),

    user_id    UUID        NOT NULL,

    event_type TEXT        NOT NULL,

    payload    JSONB       NOT NULL,

    status     TEXT        NOT NULL DEFAULT 'sent',

    created_at TIMESTAMPTZ NOT NULL DEFAULT now(),

    sent_at    TIMESTAMPTZ NULL

);

CREATE INDEX IF NOT EXISTS idx_notifications_user_created

    ON notifications(user_id, created_at DESC);

`

func runMigrations(ctx context.Context, pool *pgxpool.Pool, log *slog.Logger) error {

    _, err := pool.Exec(ctx, notificationSchema)

    if err != nil {

        return fmt.Errorf("notification migrations: %w", err)

    }

    log.Info("notification migrations applied")

    return nil

}
```

5.9 `docker-compose.yml` Additions (M5)

```yaml
# Extend existing service environment blocks:

notification-service:

  environment:

    DATABASE_URL: postgres://notificationuser:notificationpass@notification_db:5432/notificationdb

    RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/

    JWT_SECRET: "change-this-in-production-minimum-32-chars"

    PORT: "8080"

gateway:

  environment:

    URL_SERVICE_URL: http://url-service:8080

    ANALYTICS_SERVICE_URL: http://analytics-service:8080

    USER_SERVICE_URL: http://user-service:8080

    NOTIFICATION_SERVICE_URL: http://notification-service:8080

    REDIS_URL: redis://redis:6379/0

    JWT_SECRET: "change-this-in-production-minimum-32-chars"

    PORT: "8080"
```

`JWT_SECRET` must be identical across: user-service, url-service, notification-service, gateway. Any mismatch causes all JWT verifications to fail with 401

6. Error Handling Matrix

6.1 Notification Service

Error Scenario	Detected By	Recovery	HTTP Status / AMQP	Log Level
Missing `DATABASE_URL`	`loadConfig()`	`os.Exit(1)`	— (pre-HTTP)	Error (stderr)
Missing `RABBITMQ_URL`	`loadConfig()`	`os.Exit(1)`	— (pre-HTTP)	Error (stderr)
Missing `JWT_SECRET`	`loadConfig()`	`os.Exit(1)`	— (pre-HTTP)	Error (stderr)
DB unreachable at startup	`NewDBPool` ping	`os.Exit(1)`	—	Error
Migration SQL fails	`pool.Exec`	`os.Exit(1)`	—	Error
RabbitMQ unreachable (all 10 attempts)	`NewRabbitMQConn`	`os.Exit(1)`	—	Error
Queue declare fails	`DeclareNotificationQueue`	`os.Exit(1)`	—	Error
Consumer QoS fails	`ch.Qos`	`os.Exit(1)`	—	Error
Consumer `Consume` fails	`ch.Consume`	`os.Exit(1)`	—	Error
AMQP channel closed mid-run	`ok == false` on msgs	Block on `<-ctx.Done()`	—	Warn
Consumer panic	`defer recover()`	`d.Ack(false)`	—	Error
Unknown routing key	key not in {created,deleted,milestone}	`d.Ack(false)`	—	Warn
Malformed JSON body (any event type)	`json.Unmarshal`	`d.Ack(false)`	—	Error
Missing `user_id` in event	field check	`d.Ack(false)`	—	Error
DB `Insert` fails	`store.Insert`	`d.Nack(false, true)`	—	Error
Missing Authorization header	`JWTMiddleware`	401	401	none
Invalid/expired JWT on GET /notifications	`JWTMiddleware`	401	401	none
DB `FindByUserID` fails	`store.FindByUserID`	500	500	Error
Invalid `limit` query param	`strconv.Atoi`	400	400	none
Invalid `after` cursor (non-UUID)	subquery on PK	DB returns 0 rows gracefully	—	none

6.2 API Gateway

Error Scenario	Detected By	Recovery	HTTP Status	Log Level
Missing required env var	`loadConfig()`	`os.Exit(1)`	—	Error (stderr)
Redis parse URL fails	`redis.ParseURL`	Warn + continue (fail-open)	—	Warn
Redis unreachable at startup	`client.Ping`	Warn + continue (fail-open)	—	Warn
Invalid upstream URL in config	`url.Parse`	`os.Exit(1)`	—	Error
No matching route	router loop exhausted	404 `{"error":"not found"}`	404	none
Missing Authorization header	`JWTMiddleware`	401 `{"error":"authorization header required"}`	401	Info
Invalid/expired JWT	`auth.VerifyToken`	401 `{"error":"unauthorized"}`	401	Info
Rate limit exceeded	`redisTokenBucket.Allow`	429 + `Retry-After: 60` header	429	none
Redis INCR error (rate limiter)	`client.Incr`	Warn + allow (fail-open)	—	Warn
Redis Expire error (rate limiter)	`client.Expire`	Warn + continue (non-fatal)	—	Warn
Circuit OPEN, request rejected	`cb.Do` returns `ErrCircuitOpen`	503 `{"error":"service unavailable"}`	503	none (state transition already logged)
Upstream returns 5xx	proxy response status	Increment circuit breaker failure counter	5xx (proxied)	Warn
Upstream network error / timeout	`proxy.ErrorHandler`	502 `{"error":"upstream error"}`	502	Warn
Circuit transitions CLOSED→OPEN	`cb.Do` failure count	Log state transition	—	Warn
Circuit transitions OPEN→HALF_OPEN	timeout elapsed	Log state transition	—	Info
Probe succeeds (HALF_OPEN→CLOSED)	`cb.Do` success	Log state transition	—	Info
Probe fails (HALF_OPEN→OPEN)	`cb.Do` failure	Log state transition	—	Warn
Upstream health check fails at startup	`http.Get` on `/health`	Log warning; gateway starts anyway	—	Warn
Response writer write error	`w.Write` return	Ignored (client disconnected)	—	none

`writeError` and `writeJSON` in gateway (`router.go` or `errors.go`):

```go
// errors.go in gateway package

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

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

    w.WriteHeader(status)

    json.NewEncoder(w).Encode(struct {

        Error string `json:"error"`

    }{Error: message})

}

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

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

    w.WriteHeader(status)

    json.NewEncoder(w).Encode(v)

}
```

7. Implementation Sequence with Checkpoints

Phase 1: Notification Service — DB Schema, Consumer, Handler (2–2.5h)

1. Write `migration.sql` as specified in §3.1.
2. Write `errors.go`: `writeError`, `writeJSON`, `truncate`.
3. Write `store.go`: `NotificationRecord`, `NotificationRepository` interface, `pgxNotificationStore` with `Insert` (two-step tx: INSERT then UPDATE status) and `FindByUserID` (cursor pagination).
4. Write `consumer.go`: `NotificationConsumer`, `Run()`, `processDelivery()` with full event-type switch and `mockEmail` log.
5. Write `handler.go`: `NotificationHandler`, `ListNotifications` method.
6. Extend `config.go` with `JWTSecret`.
7. Extend `main.go`: `runMigrations`, wire all components, register routes, start consumer goroutine.

```bash
docker compose up notification_db rabbitmq -d

DATABASE_URL="postgres://notificationuser:notificationpass@localhost:5435/notificationdb" \

RABBITMQ_URL="amqp://guest:guest@localhost:5672/" \

JWT_SECRET="test-secret-minimum-32-chars-long" \

go run ./services/notification-service/
```

Verify schema:

```bash
psql -h localhost -p 5435 -U notificationuser -d notificationdb \

  -c "\d notifications; \di idx_notifications_*;"
```

Manually publish a test message to `notifications.events` queue via RabbitMQ management UI (routing key `url.created`, body matching `URLCreatedEvent` JSON with `user_id` and `user_email` populated). **Checkpoint:** Consumer logs `"would send email to user"` with `user_id` and `message`. `SELECT * FROM notifications;` shows one row with `status='sent'` and non-null `sent_at`. `GET /notifications` with a valid JWT returns `200 [{"id":"...","event_type":"url.created",...}]`. `go build ./services/notification-service/...` exits 0.

Phase 2: Extended `shared/logger` (0.5–1h)

1. Extend `shared/logger/logger.go` with `WithCorrelationID`, `CorrelationIDFromContext`, `ContextWithCorrelationID`, `RequestLogger`, and the `responseWriter` capture struct.
2. Update `go.work` if not already including `shared/logger`.
3. Verify all existing services still compile: `go build ./...` from monorepo root. **Checkpoint:** `go build github.com/yourhandle/url-shortener/shared/logger` exits 0. A test that calls `logger.RequestLogger` as middleware and checks that the log line contains `"correlation_id"` passes.

Phase 3: Gateway Routing Table and Reverse Proxy (1–1.5h)

1. Extend `gateway/config.go` to full spec (§3.4 Config struct with all fields).
2. Write `gateway/router.go`: `Route` struct, `routingTable` slice, `Gateway` struct, stub `ServeHTTP` that routes + proxies without auth/rate limit yet.
3. Write `gateway/proxy.go`: `UpstreamProxy`, `NewUpstreamProxy`, `ServeHTTP` with path rewriting.
4. Write `gateway/middleware.go`: `CorrelationIDMiddleware`, `chain` helper.
5. Extend `gateway/main.go`: construct proxies, gateway, wire mux, start server.

```bash
docker compose up url-service analytics-service user-service notification-service -d

REDIS_URL="redis://localhost:6379/0" \

URL_SERVICE_URL="http://localhost:8081" \

ANALYTICS_SERVICE_URL="http://localhost:8082" \

USER_SERVICE_URL="http://localhost:8083" \

NOTIFICATION_SERVICE_URL="http://localhost:8084" \

JWT_SECRET="test-secret-minimum-32-chars-long" \

go run ./gateway/

# Test proxy routing (no auth yet on protected routes in this phase):

curl http://localhost:8080/api/auth/register -X POST \

  -d '{"email":"e2e@example.com","password":"password123"}' \

  -H "Content-Type: application/json"

# Expect: 201 {user_id, email} (proxied to user-service)
```

Checkpoint: `GET http://localhost:8080/health` returns `{"status":"ok","service":"gateway"}`. `POST /api/auth/register` proxied correctly to user-service. `GET /api/stats/<code>` proxied to analytics-service. Unknown path `/xyz` returns 404. `X-Correlation-ID` response header present on all responses.

Phase 4: Gateway JWT Middleware and X-Correlation-ID Propagation (1h)

1. Write `gateway/jwtmiddleware.go`: `GatewayJWTMiddleware` wrapping `shared/auth.JWTMiddleware`.
2. Integrate into `Gateway.ServeHTTP`: apply JWT check when `route.RequiresAuth == true`.
3. Verify `X-Correlation-ID` is forwarded to upstream (set on `r.Header` before proxy).

```bash
# Without token on protected route:
curl http://localhost:8080/api/shorten -X POST \
  -d '{"url":"https://example.com"}' -H "Content-Type: application/json"
# Expect: 401 {"error":"authorization header required"}
# With invalid token:
curl http://localhost:8080/api/urls \
  -H "Authorization: Bearer invalid.token.here"
# Expect: 401
# With valid token (get one from login first):
curl http://localhost:8080/api/urls \
  -H "Authorization: Bearer $VALID_TOKEN"
# Expect: 200 {"urls":[],"next_cursor":null} (proxied to url-service)
```

Check that `X-Correlation-ID` appears in url-service logs:

```bash
docker compose logs url-service | grep correlation_id
```

Checkpoint: Auth-protected routes return 401 without valid JWT. Valid JWT routes through to upstream. `X-Correlation-ID` appears in forwarded request headers and in downstream service logs. Public routes (`/api/auth/*` , `/r/*` , `/api/stats/*`) pass through without JWT check.

Phase 5: Redis Token-Bucket Rate Limiter with 429 + Retry-After (1.5–2h)

1. Write `gateway/ratelimit.go` : `RateLimiter` interface, `redisTokenBucket` , `Allow` implementation, `clientIP` , `buildRateLimitKey` .
2. Integrate into `Gateway.ServeHTTP` : apply rate limit when `route.RateLimitKey != ""` .
3. Wire `redis.Client` construction in `gateway/main.go` .

```bash
# Rate limit test for POST /api/shorten (limit=10/min):
TOKEN=$(curl -s -X POST http://localhost:8080/api/auth/login \
  -d '{"email":"e2e@example.com","password":"password123"}' \
  -H "Content-Type: application/json" | python3 -c "import sys,json; print(json.load(sys.stdin)['token'])")
for i in $(seq 1 11); do
  RESP=$(curl -s -o /dev/null -w "%{http_code}" -X POST http://localhost:8080/api/shorten \
    -d '{"url":"https://example.com/'$i'"}' \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $TOKEN")
  echo "Request $i: HTTP $RESP"
done
# Expected output:
# Request 1-10: HTTP 201 (or whatever url-service returns)
# Request 11:   HTTP 429
```

Verify `Retry-After` header on 429:

```bash
curl -v -X POST http://localhost:8080/api/shorten ... 2>&1 | grep -i retry-after

# Expect: < Retry-After: 60
```

Verify Redis keys:

```bash
redis-cli -p 6379 KEYS "rl:*"

# Expect: ["rl:shorten:127.0.0.1"]

redis-cli -p 6379 GET "rl:shorten:127.0.0.1"

# Expect: "11" (or current count)

redis-cli -p 6379 TTL "rl:shorten:127.0.0.1"

# Expect: positive integer ≤ 60
```

Checkpoint: 11th `POST /api/shorten` from same IP returns `429` with `Retry-After: 60` header. Redis key `rl:shorten:<ip>` has a TTL ≤ 60s. `GET /r/<code>` allows 300 requests before 429. Killing Redis and making a request: gateway logs `"rate limit redis incr failed"` at Warn, request is allowed (fail-open), no 500 returned.

Phase 6: Circuit Breaker State Machine (2–2.5h)

1. Write `gateway/circuitbreaker.go`: `State`, `CircuitBreaker`, `NewCircuitBreaker`, `Do()` method (full algorithm from §4.4).
2. Integrate `CircuitBreaker` into `Gateway.ServeHTTP` for url-service only.
3. Wire `NewCircuitBreaker(cfg.CBMaxFailures, cfg.CBOpenTimeout, cfg.CBFailureWindow)` in `main.go`.

```bash
# Simulate url-service failure by stopping it:

docker compose stop url-service

# Make 5 requests (will fail; circuit counts them):

for i in $(seq 1 6); do

  RESP=$(curl -s -o /dev/null -w "%{http_code}" http://localhost:8080/r/abc1234)

  echo "Request $i: HTTP $RESP"

done

# Requests 1-5: HTTP 502 (upstream error, circuit counting)

# Request 6+:   HTTP 503 (circuit OPEN, rejected immediately)

# Wait 30s then try again (half-open probe):

sleep 31

RESP=$(curl -s -o /dev/null -w "%{http_code}" http://localhost:8080/r/abc1234)

echo "Probe: HTTP $RESP"

# If url-service still stopped: 502 (probe fails → back to OPEN)

# Start url-service again:

docker compose start url-service

sleep 31  # wait for next half-open

RESP=$(curl -s -o /dev/null -w "%{http_code}" http://localhost:8080/r/abc1234)

echo "Probe after restart: HTTP $RESP"

# Expect: 301 (probe succeeds → CLOSED)
```

Check gateway logs for state transitions:

```bash
docker compose logs gateway | grep "circuit breaker"

# Expected lines:

# {"level":"WARN","service":"gateway","msg":"circuit breaker OPEN","failures":5,...}

# {"level":"INFO","service":"gateway","msg":"circuit breaker CLOSED (probe succeeded)",...}
```

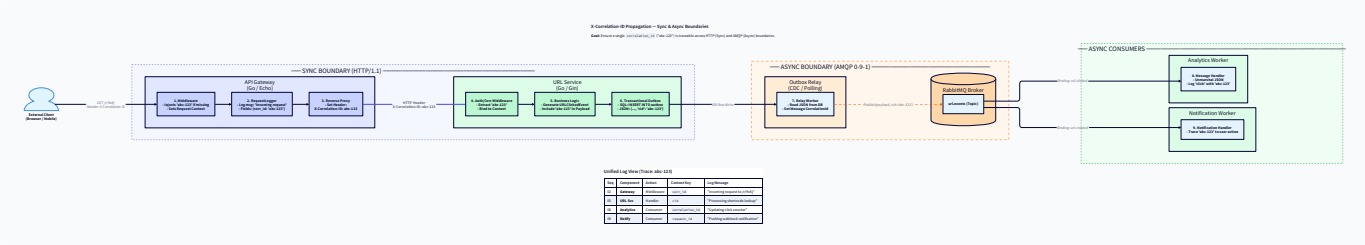
Checkpoint: Circuit opens after 5 consecutive 5xx responses within 10s. `GET /r/<code>` returns `503 {"error":"service unavailable"}` immediately (< 1ms, no upstream call) while circuit is OPEN. After 30s, one probe request is forwarded. On success, circuit closes. Other upstream proxies (analytics, user, notification) are NOT affected by url-service circuit breaker.

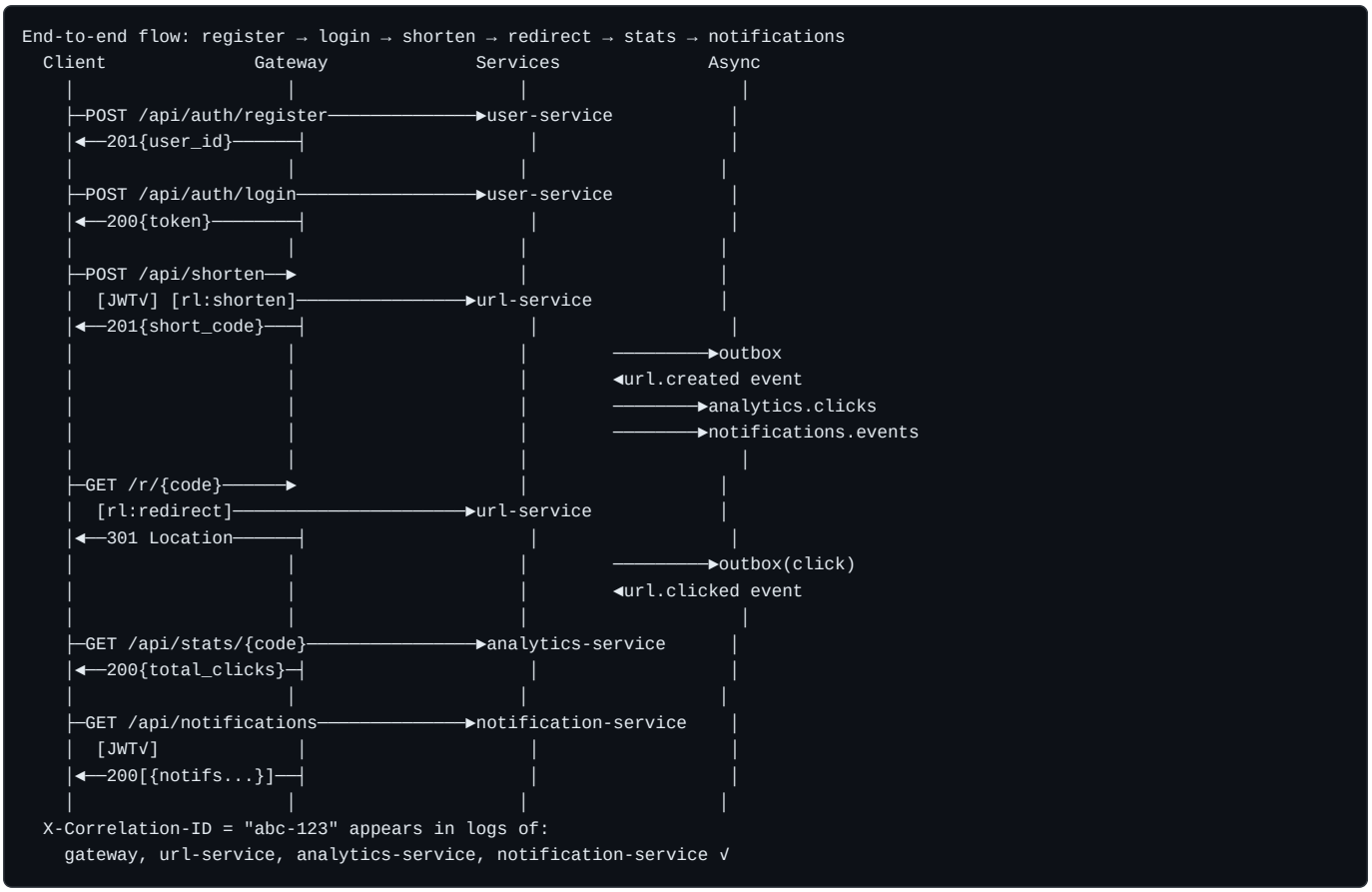
Phase 7: End-to-End Integration Test (1.5–2h)

1. Write `gateway/gateway_test.go` (§8 test cases).
2. Write integration test script `scripts/e2e_test.sh`.
3. Run full stack via `docker compose up --build -d` and execute end-to-end test.

```bash
docker compose up --build -d

sleep 60  # wait for all containers healthy

bash scripts/e2e_test.sh
```

Checkpoint: End-to-end test script passes all assertions. Rate limit test: 11th `POST /api/shorten` returns 429. Circuit breaker test: 6th request after simulated failure returns 503. `X-Correlation-ID` appears in logs of gateway, url-service, analytics-service, notification-service for the same request UUID. `docker compose down -v` teardown exits 0.




End-to-end flow: register → login → shorten → redirect → stats → notifications
Client
Gateway
Services
Async
POST /api/auth/register
user-service
201{user_id}
POST /api/auth/login
user-service
200{token}
POST /api/shorten
[JWT√] [rl:shorten]
url-service
201{short_code}
outbox
url.created event
analytics.clicks
notifications.events
GET /r/{code}
[rl:redirect]
url-service
301 Location
outbox(click)
url.clicked event
GET /api/stats/{code}
analytics-service
200{total_clicks}
GET /api/notifications
[JWT√]
notification-service
200[{notifs...}]
X-Correlation-ID = "abc-123" appears in logs of:
gateway, url-service, analytics-service, notification-service √

8. Test Specification

8.1 Circuit Breaker Unit Tests (`gateway_test.go`)

```go
package gateway

import (

    "errors"

    "sync"

    "testing"

    "time"

)

func TestCircuitBreaker_InitiallyClosed(t *testing.T) {

    cb := NewCircuitBreaker(5, 30*time.Second, 10*time.Second)

    if cb.state != StateClosed {

        t.Errorf("initial state: got %v want CLOSED", cb.state)

    }

}

func TestCircuitBreaker_ClosedPassesSuccess(t *testing.T) {

    cb := NewCircuitBreaker(5, 30*time.Second, 10*time.Second)

    err := cb.Do(context.Background(), func() error { return nil })

    if err != nil {

        t.Errorf("success in CLOSED: got error %v", err)

    }

    cb.mu.Lock()

    if cb.state != StateClosed {

        t.Errorf("state after success: got %v", cb.state)

    }

    cb.mu.Unlock()

}

func TestCircuitBreaker_OpensAfterMaxFailures(t *testing.T) {

    cb := NewCircuitBreaker(3, 30*time.Second, 10*time.Second)

    fail := func() error { return errors.New("upstream error") }

    for i := 0; i < 3; i++ {

        cb.Do(context.Background(), fail)

    }

    cb.mu.Lock()

    state := cb.state

    cb.mu.Unlock()
```

```
    if state != StateOpen {
        t.Errorf("state after max failures: got %v want OPEN", state)
    }
}
func TestCircuitBreaker_OpenRejectsWithoutCallingUpstream(t *testing.T) {
    cb := NewCircuitBreaker(1, 30*time.Second, 10*time.Second)
    cb.Do(context.Background(), func() error { return errors.New("fail") }) // trip
    called := false
    err := cb.Do(context.Background(), func() error {
        called = true
        return nil
    })
    if called {
        t.Error("upstream should NOT be called when circuit is OPEN")
    }
    if !errors.Is(err, ErrCircuitOpen) {
        t.Errorf("expected ErrCircuitOpen, got %v", err)
    }
}
func TestCircuitBreaker_TransitionsToHalfOpenAfterTimeout(t *testing.T) {
    cb := NewCircuitBreaker(1, 100*time.Millisecond, 10*time.Second)
    cb.Do(context.Background(), func() error { return errors.New("fail") }) // trip
    time.Sleep(150 * time.Millisecond)
    // Next Do should transition to HALF_OPEN and allow through
    probeCalled := false
    cb.Do(context.Background(), func() error {
        probeCalled = true
        return nil
    })
    if !probeCalled {
        t.Error("probe request should be forwarded in HALF_OPEN")
    }
    cb.mu.Lock()
    state := cb.state
    cb.mu.Unlock()
    if state != StateClosed {
```

```
        t.Errorf("state after successful probe: got %v want CLOSED", state)
    }
}
func TestCircuitBreaker_HalfOpenFailureReopens(t *testing.T) {
    cb := NewCircuitBreaker(1, 100*time.Millisecond, 10*time.Second)
    cb.Do(context.Background(), func() error { return errors.New("fail") })
    time.Sleep(150 * time.Millisecond)
    cb.Do(context.Background(), func() error { return errors.New("probe fail") })
    cb.mu.Lock()
    state := cb.state
    cb.mu.Unlock()
    if state != StateOpen {
        t.Errorf("state after failed probe: got %v want OPEN", state)
    }
}
func TestCircuitBreaker_SuccessResetFailureCount(t *testing.T) {
    cb := NewCircuitBreaker(3, 30*time.Second, 10*time.Second)
    fail := func() error { return errors.New("fail") }
    cb.Do(context.Background(), fail)
    cb.Do(context.Background(), fail) // 2 failures
    cb.Do(context.Background(), func() error { return nil }) // success — reset
    cb.mu.Lock()
    failures := cb.failures
    cb.mu.Unlock()
    if failures != 0 {
        t.Errorf("failures after success: got %d want 0", failures)
    }
}
func TestCircuitBreaker_ConcurrentSafe(t *testing.T) {
    cb := NewCircuitBreaker(100, 30*time.Second, 10*time.Second)
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            cb.Do(context.Background(), func() error { return nil })
```

```
        }()
    }
    wg.Wait() // must not race-detect or panic
}
func TestCircuitBreaker_FailureWindowReset(t *testing.T) {
    // Failures in window 1 do not carry over to window 2
    cb := NewCircuitBreaker(3, 30*time.Second, 100*time.Millisecond)
    fail := func() error { return errors.New("fail") }
    cb.Do(context.Background(), fail)
    cb.Do(context.Background(), fail) // 2 failures in window 1
    time.Sleep(150 * time.Millisecond) // window expires
    cb.Do(context.Background(), fail) // 1 failure in window 2
    cb.mu.Lock()
    state := cb.state
    failures := cb.failures
    cb.mu.Unlock()
    if state != StateClosed {
        t.Errorf("should still be CLOSED (only 1 failure in new window): got %v", state)
    }
    if failures != 1 {
        t.Errorf("failures in new window: got %d want 1", failures)
    }
}
```

8.2 Rate Limiter Unit Tests

```go
func TestRateLimiter_AllowsUnderLimit(t *testing.T) {
    mr := miniredis.RunT(t)
    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})
    rl := NewRateLimiter(client, slog.Default())
    for i := 0; i < 10; i++ {
        allowed, _, err := rl.Allow(context.Background(), "test:127.0.0.1", 10, 60)
        if err != nil { t.Fatalf("Allow error: %v", err) }
        if !allowed { t.Errorf("request %d should be allowed", i+1) }
    }
}
func TestRateLimiter_BlocksAtLimit(t *testing.T) {
    mr := miniredis.RunT(t)
    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})
    rl := NewRateLimiter(client, slog.Default())
    for i := 0; i < 10; i++ {
        rl.Allow(context.Background(), "shorten:127.0.0.1", 10, 60)
    }
    allowed, retryAfter, err := rl.Allow(context.Background(), "shorten:127.0.0.1", 10, 60)
    if err != nil { t.Fatalf("Allow error: %v", err) }
    if allowed { t.Error("11th request should be blocked") }
    if retryAfter != 60 { t.Errorf("retryAfter: got %d want 60", retryAfter) }
}
func TestRateLimiter_KeyHasTTL(t *testing.T) {
    mr := miniredis.RunT(t)
    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})
    rl := NewRateLimiter(client, slog.Default())
    rl.Allow(context.Background(), "shorten:10.0.0.1", 10, 60)
    ttl := mr.TTL("rl:shorten:10.0.0.1")
    if ttl <= 0 { t.Error("rate limit key must have a positive TTL") }
    if ttl > 60*time.Second { t.Errorf("TTL too long: %v", ttl) }
}
func TestRateLimiter_WindowResets(t *testing.T) {
    mr := miniredis.RunT(t)
    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})
    rl := NewRateLimiter(client, slog.Default())
```

```
    for i := 0; i < 10; i++ {
        rl.Allow(context.Background(), "shorten:1.1.1.1", 10, 1)
    }
    mr.FastForward(2 * time.Second)
    allowed, _, _ := rl.Allow(context.Background(), "shorten:1.1.1.1", 10, 1)
    if !allowed { t.Error("request after window reset should be allowed") }
}
func TestRateLimiter_FailOpenOnRedisError(t *testing.T) {
    // Unreachable Redis → Allow returns true (fail-open)
    client := redis.NewClient(&redis.Options{Addr: "localhost:1"})
    rl := NewRateLimiter(client, slog.Default())
    allowed, _, err := rl.Allow(context.Background(), "test:1.2.3.4", 10, 60)
    if !allowed { t.Error("should fail-open when Redis unreachable") }
    if err == nil { t.Error("err should be non-nil for infrastructure failure") }
}
func TestRateLimiter_DifferentIPsIndependent(t *testing.T) {
    mr := miniredis.RunT(t)
    client := redis.NewClient(&redis.Options{Addr: mr.Addr()})
    rl := NewRateLimiter(client, slog.Default())
    // Exhaust limit for IP1
    for i := 0; i < 10; i++ {
        rl.Allow(context.Background(), "shorten:192.168.1.1", 10, 60)
    }
    // IP2 should still be allowed
    allowed, _, _ := rl.Allow(context.Background(), "shorten:192.168.1.2", 10, 60)
    if !allowed { t.Error("different IP should not be rate limited") }
}
```

8.3 Router Unit Tests

```go
func TestRouter_MatchesCorrectRoute(t *testing.T) {
    cases := []struct {
        method   string
        path     string
        wantUpstream string
        wantAuth bool
    }{
        {"POST", "/api/auth/register", "user-service", false},
        {"POST", "/api/auth/login",    "user-service", false},
        {"POST", "/api/shorten",       "url-service",  true},
        {"GET",  "/api/urls",          "url-service",  true},
        {"GET",  "/r/abc1234",         "url-service",  false},
        {"GET",  "/api/stats/abc1234", "analytics-service", false},
        {"GET",  "/api/notifications", "notification-service", true},
    }
    gw := &Gateway{routes: routingTable, log: slog.Default()}
    for _, tc := range cases {
        t.Run(tc.method+" "+tc.path, func(t *testing.T) {
            matched := gw.matchRoute(tc.method, tc.path)
            if matched == nil {
                t.Fatal("no route matched")
            }
            if matched.Upstream != tc.wantUpstream {
                t.Errorf("upstream: got %q want %q", matched.Upstream, tc.wantUpstream)
            }
            if matched.RequiresAuth != tc.wantAuth {
                t.Errorf("auth: got %v want %v", matched.RequiresAuth, tc.wantAuth)
            }
        })
    }
}
func TestRouter_NoMatchReturns404(t *testing.T) {
    req := httptest.NewRequest("GET", "/nonexistent/path", nil)
    rec := httptest.NewRecorder()
    gw := buildTestGateway()
```

```
    gw.ServeHTTP(rec, req)
    if rec.Code != 404 {
        t.Errorf("unknown path: got %d want 404", rec.Code)
    }
}
func TestRouter_JWTRequired_Returns401WithoutToken(t *testing.T) {
    req := httptest.NewRequest("GET", "/api/urls", nil)
    rec := httptest.NewRecorder()
    gw := buildTestGateway()
    gw.ServeHTTP(rec, req)
    if rec.Code != 401 {
        t.Errorf("protected route without JWT: got %d want 401", rec.Code)
    }
}
```

8.4 Notification Consumer Unit Tests

```go
func TestNotificationConsumer_URLCreatedEvent(t *testing.T) {
    // Build URLCreatedEvent JSON
    evt := events.URLCreatedEvent{
        BaseEvent: events.BaseEvent{
            EventType:     "url.created",
            OccurredAt:    time.Now().UTC(),
            EventID:       "test-evt-001",
            CorrelationID: "corr-123",
        },
        ShortCode:   "abc1234",
        OriginalURL: "https://example.com",
        UserID:      "user-uuid-001",
        UserEmail:   "user@example.com",
    }
    body, _ := json.Marshal(evt)
    inserted := make([]*NotificationRecord, 0)
    store := &mockNotificationStore{
        insertFn: func(ctx context.Context, rec *NotificationRecord) (*NotificationRecord, error) {
            inserted = append(inserted, rec)
            rec.ID = "notif-uuid-001"
            return rec, nil
        },
    }
    consumer := &NotificationConsumer{store: store, log: slog.Default()}
    d := amqp.Delivery{RoutingKey: "url.created", Body: body}
    consumer.processDelivery(context.Background(), d)  // NOTE: test version that doesn't call d.Ack
    if len(inserted) != 1 {
        t.Fatalf("expected 1 insert, got %d", len(inserted))
    }
    if inserted[0].UserID != "user-uuid-001" {
        t.Errorf("user_id: got %q", inserted[0].UserID)
    }
    if inserted[0].EventType != "url.created" {
        t.Errorf("event_type: got %q", inserted[0].EventType)
    }
```

```
}

func TestNotificationConsumer_MalformedJSON_DoesNotInsert(t *testing.T) {

    insertCalled := false

    store := &mockNotificationStore{

        insertFn: func(_ context.Context, _ *NotificationRecord) (*NotificationRecord, error) {

            insertCalled = true

            return nil, nil

        },

    }

    consumer := &NotificationConsumer{store: store, log: slog.Default()}

    d := amqp.Delivery{RoutingKey: "url.created", Body: []byte("not json")}

    consumer.processDelivery(context.Background(), d)

    if insertCalled {

        t.Error("insert should NOT be called for malformed JSON")

    }

}

func TestNotificationConsumer_MissingUserID_DoesNotInsert(t *testing.T) {

    // URLCreatedEvent with UserID = ""

    evt := events.URLCreatedEvent{

        BaseEvent: events.BaseEvent{EventType: "url.created", EventID: "x"},

        UserID:    "", // missing

    }

    body, _ := json.Marshal(evt)

    insertCalled := false

    store := &mockNotificationStore{

        insertFn: func(_ context.Context, _ *NotificationRecord) (*NotificationRecord, error) {

            insertCalled = true; return nil, nil

        },

    }

    consumer := &NotificationConsumer{store: store, log: slog.Default()}

    consumer.processDelivery(context.Background(), amqp.Delivery{RoutingKey: "url.created", Body: body})

    if insertCalled { t.Error("insert must not be called when user_id is empty") }

}

func TestNotificationConsumer_UnknownRoutingKey_Acked(t *testing.T) {

    // Routing key not in {url.created, url.deleted, milestone.reached}

    // Must Ack (not Nack) and not insert
```

```
    insertCalled := false
    store := &mockNotificationStore{
        insertFn: func(_ context.Context, _ *NotificationRecord) (*NotificationRecord, error) {
            insertCalled = true; return nil, nil
        },
    }
    consumer := &NotificationConsumer{store: store, log: slog.Default()}
    consumer.processDelivery(context.Background(), amqp.Delivery{
        RoutingKey: "unknown.key", Body: []byte(`{}`),
    })
    if insertCalled { t.Error("unknown routing key must not trigger insert") }
}
```

8.5 `NotificationHandler` Unit Tests

```go
func TestListNotifications_EmptyList(t *testing.T) {
    store := &mockNotificationStore{
        findByUserIDFn: func(_ context.Context, _, _ string, _ int) ([]*NotificationRecord, string, error) {
            return []*NotificationRecord{}, "", nil
        },
    }
    h := NewNotificationHandler(store, slog.Default())
    claims := &auth.Claims{Sub: "user-uuid", Email: "u@e.com", Iss: "url-shortener"}
    ctx := context.WithValue(context.Background(), auth.TestClaimsKey{}, claims)
    req := httptest.NewRequest("GET", "/notifications", nil).WithContext(ctx)
    rec := httptest.NewRecorder()
    h.ListNotifications(rec, req)
    if rec.Code != 200 { t.Errorf("got %d", rec.Code) }
    var resp notificationListResponse
    json.Unmarshal(rec.Body.Bytes(), &resp)
    if resp.Notifications == nil { t.Error("notifications must be [] not null") }
    if len(resp.Notifications) != 0 { t.Errorf("expected 0, got %d", len(resp.Notifications)) }
    if resp.NextCursor != nil { t.Error("next_cursor should be null") }
}
func TestListNotifications_WithResults(t *testing.T) {
    createdAt := time.Now().UTC()
    store := &mockNotificationStore{
        findByUserIDFn: func(_ context.Context, _, _ string, _ int) ([]*NotificationRecord, string, error) {
            return []*NotificationRecord{
                {
                    ID:        "notif-001",
                    UserID:    "user-uuid",
                    EventType: "url.created",
                    Payload:   []byte(`{"short_code":"abc"}`),
                    Status:    "sent",
                    CreatedAt: createdAt,
                },
            }, "", nil
        },
    }
```

```
    h := NewNotificationHandler(store, slog.Default())
    claims := &auth.Claims{Sub: "user-uuid"}
    ctx := context.WithValue(context.Background(), auth.TestClaimsKey{}, claims)
    req := httptest.NewRequest("GET", "/notifications", nil).WithContext(ctx)
    rec := httptest.NewRecorder()
    h.ListNotifications(rec, req)
    var resp notificationListResponse
    json.Unmarshal(rec.Body.Bytes(), &resp)
    if len(resp.Notifications) != 1 { t.Fatalf("expected 1 notification") }
    if resp.Notifications[0].ID != "notif-001" { t.Error("id mismatch") }
    if resp.Notifications[0].EventType != "url.created" { t.Error("event_type mismatch") }
}
func TestListNotifications_InvalidLimitParam(t *testing.T) {
    for _, bad := range []string{"abc", "0", "51", "-1"} {
        t.Run("limit="+bad, func(t *testing.T) {
            h := NewNotificationHandler(&mockNotificationStore{}, slog.Default())
            claims := &auth.Claims{Sub: "uid"}
            ctx := context.WithValue(context.Background(), auth.TestClaimsKey{}, claims)
            req := httptest.NewRequest("GET", "/notifications?limit="+bad, nil).WithContext(ctx)
            rec := httptest.NewRecorder()
            h.ListNotifications(rec, req)
            if rec.Code != 400 { t.Errorf("limit=%s: got %d want 400", bad, rec.Code) }
        })
    }
}
```

8.6 End-to-End Integration Test Script (`scripts/e2e_test.sh`)

BASH

```bash
#!/usr/bin/env bash

# scripts/e2e_test.sh — full end-to-end test through gateway

set -e

BASE="http://localhost:8080"

EMAIL="e2e+$(date +%s)@example.com"

PASSWORD="e2epassword123"

echo "=== 1. Register ==="

REG=$(curl -sf -X POST "$BASE/api/auth/register" \

  -H "Content-Type: application/json" \

  -d "{\"email\":\"$EMAIL\",\"password\":\"$PASSWORD\"}")

USER_ID=$(echo "$REG" | python3 -c "import sys,json; print(json.load(sys.stdin)['user_id'])")

echo "PASS: registered user_id=$USER_ID"

echo "=== 2. Login ==="

LOGIN=$(curl -sf -X POST "$BASE/api/auth/login" \

  -H "Content-Type: application/json" \

  -d "{\"email\":\"$EMAIL\",\"password\":\"$PASSWORD\"}")

TOKEN=$(echo "$LOGIN" | python3 -c "import sys,json; print(json.load(sys.stdin)['token'])")

echo "PASS: token obtained"

echo "=== 3. Shorten URL ==="

SHORTEN=$(curl -sf -X POST "$BASE/api/shorten" \

  -H "Content-Type: application/json" \

  -H "Authorization: Bearer $TOKEN" \

  -d '{"url":"https://example.com/very/long/path"}')

SHORT_CODE=$(echo "$SHORTEN" | python3 -c "import sys,json; print(json.load(sys.stdin)['short_code'])")

echo "PASS: short_code=$SHORT_CODE"

echo "=== 4. Redirect ==="

REDIRECT_STATUS=$(curl -so /dev/null -w "%{http_code}" "http://localhost:8080/r/$SHORT_CODE")

if [ "$REDIRECT_STATUS" != "301" ]; then

  echo "FAIL: redirect returned $REDIRECT_STATUS, want 301"; exit 1

fi

echo "PASS: redirect returned 301"

echo "=== 5. Check stats (wait 3s for outbox) ==="

sleep 3

STATS=$(curl -sf "$BASE/api/stats/$SHORT_CODE")

TOTAL=$(echo "$STATS" | python3 -c "import sys,json; print(json.load(sys.stdin)['total_clicks'])")
```

```
if [ "$TOTAL" -lt "1" ]; then
  echo "FAIL: total_clicks=$TOTAL, expected >= 1"; exit 1
fi
echo "PASS: total_clicks=$TOTAL"
echo "=== 6. Check notifications (wait 2s) ==="
sleep 2
NOTIFS=$(curl -sf "$BASE/api/notifications" -H "Authorization: Bearer $TOKEN")
COUNT=$(echo "$NOTIFS" | python3 -c "import sys,json; print(len(json.load(sys.stdin)['notifications']))")
if [ "$COUNT" -lt "1" ]; then
  echo "FAIL: notifications count=$COUNT, expected >= 1"; exit 1
fi
echo "PASS: notifications count=$COUNT"
echo "=== 7. Rate limit test (11 POST /api/shorten) ==="
for i in $(seq 1 10); do
  curl -s -X POST "$BASE/api/shorten" \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $TOKEN" \
    -d "{\"url\":\"https://example.com/$i\"}" > /dev/null
done
ELEVENTH=$(curl -so /dev/null -w "%{http_code}" -X POST "$BASE/api/shorten" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"url":"https://example.com/overflow"}')
if [ "$ELEVENTH" != "429" ]; then
  echo "FAIL: 11th shorten request returned $ELEVENTH, want 429"; exit 1
fi
echo "PASS: 11th request returned 429"
echo "=== 8. Correlation ID propagation ==="
CORR_ID="test-corr-$(date +%s)"
curl -sf "$BASE/api/stats/$SHORT_CODE" -H "X-Correlation-ID: $CORR_ID" > /dev/null
sleep 1
GW_LOG=$(docker compose logs gateway 2>/dev/null | grep "$CORR_ID" | wc -l)
ANALYTICS_LOG=$(docker compose logs analytics-service 2>/dev/null | grep "$CORR_ID" | wc -l)
if [ "$GW_LOG" -eq "0" ]; then
  echo "FAIL: correlation_id $CORR_ID not found in gateway logs"; exit 1
fi
```

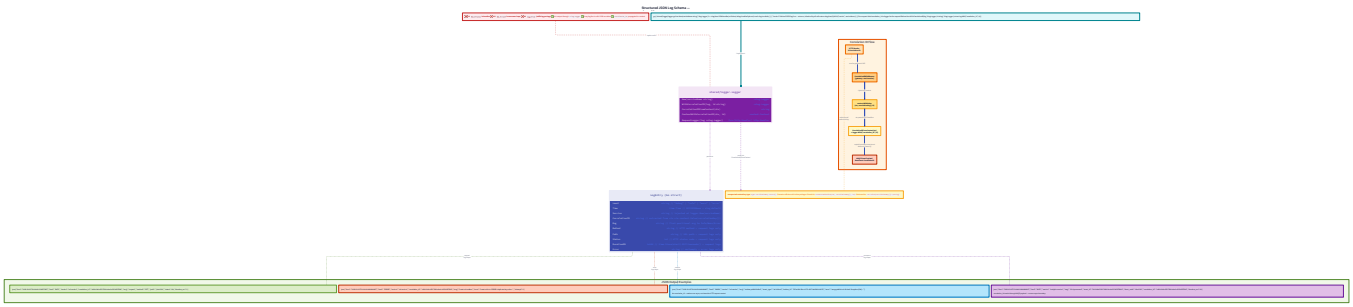
```
if [ "$ANALYTICS_LOG" -eq "0" ]; then
  echo "FAIL: correlation_id $CORR_ID not found in analytics-service logs"; exit 1
fi
echo "PASS: correlation_id propagated to gateway and analytics-service logs"
echo ""
echo "All end-to-end tests PASSED."
```

9. Performance Targets

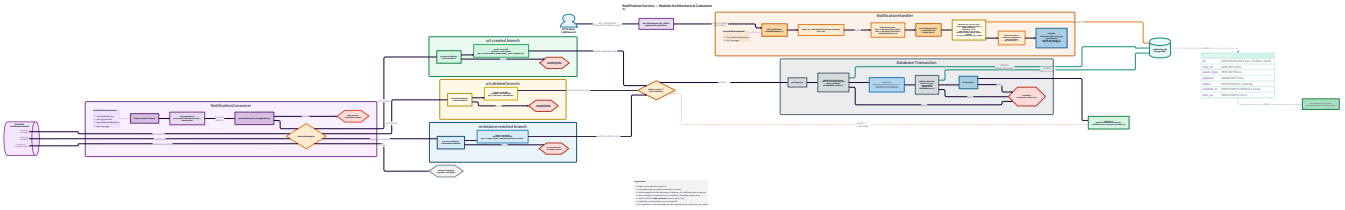
Operation	Target	How to Measure
Gateway overhead (JWT verify + rate limit Redis check + proxy)	< 5ms added latency p99	Compare `wrk` p99 direct to url-service vs through gateway on `GET /r/<code>`
Circuit breaker state read (CLOSED → allow)	< 1µs	`go test -bench=BenchmarkCBClosed -benchtime=10s ./gateway/`
Circuit breaker reject (OPEN → 503)	< 100µs	`go test -bench=BenchmarkCBOpen -benchtime=10s ./gateway/`
Rate limit check (`INCR` + conditional `EXPIRE`)	< 2ms p99	`wrk -t1 -c1 -d10s` against gateway shorten route; compare Redis latency via `redis-cli --latency`
`GET /notifications` (DB query, 20 rows)	< 20ms p99	`wrk -t4 -c20 -d30s` with valid Bearer token via Lua script
End-to-end register→shorten→redirect→stats through gateway	< 100ms p99	Shell script with `curl --trace-time` measuring each step
11th rate-limited request (429, no upstream call)	< 2ms	`curl -o /dev/null -w "%{time_total}"` on the 11th request
503 circuit-open response (no upstream call)	< 1ms	Same measurement approach
Notification consumer message processing	< 30ms p99	Log `duration_ms` in `processDelivery`, check after 1000 messages
Gateway startup to HTTP ready	< 5s after all upstreams healthy	Docker healthcheck pass time in `docker compose ps`

Circuit breaker benchmarks:

```go
func BenchmarkCircuitBreakerClosed(b *testing.B) {
    cb := NewCircuitBreaker(5, 30*time.Second, 10*time.Second)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        cb.Do(context.Background(), func() error { return nil })
    }
    // Expected: ~200-500 ns/op (mutex acquire + single function call)
}
func BenchmarkCircuitBreakerOpen(b *testing.B) {
    cb := NewCircuitBreaker(1, 30*time.Second, 10*time.Second)
    cb.Do(context.Background(), func() error { return errors.New("trip") })
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        cb.Do(context.Background(), func() error { return nil })
    }
    // Expected: ~100-200 ns/op (mutex acquire + state check + return ErrCircuitOpen)
}
```



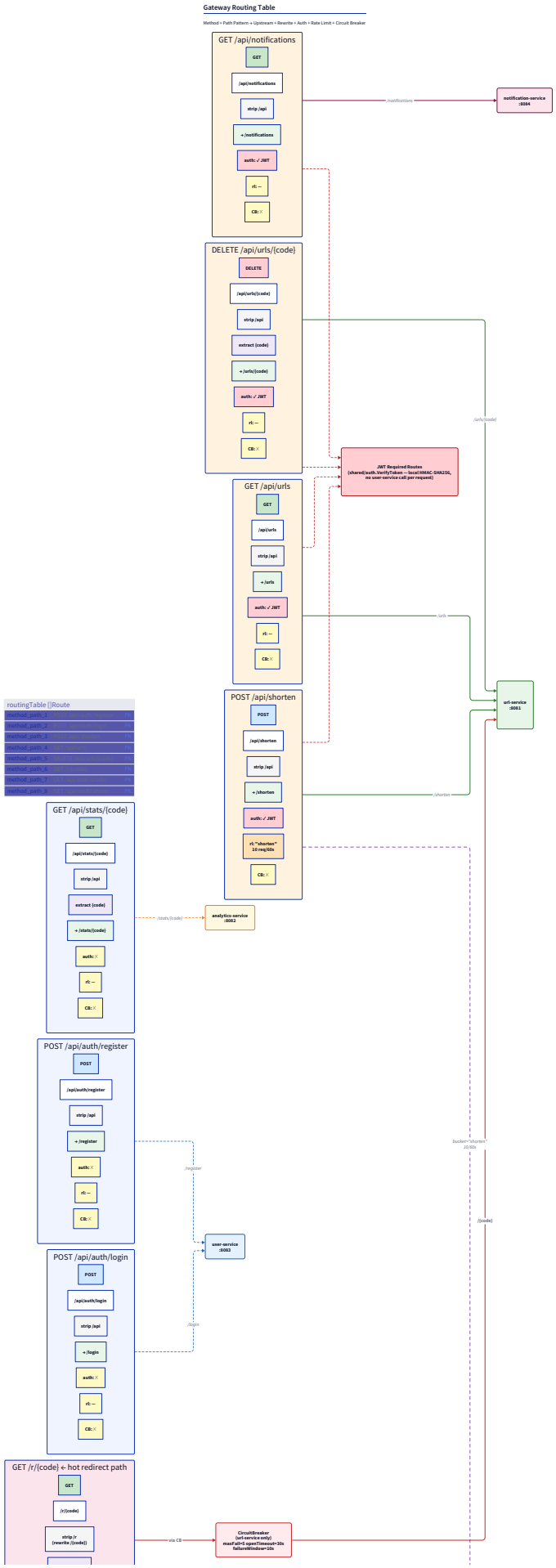
```
Latency budget for GET /r/{code} through gateway (cache hit path):
  ┌──────────────────────────────────────────────────────────────────┐
  | Network recv + HTTP parse:                  ~0.1ms               |
  | CorrelationIDMiddleware (UUID gen if absent):< 0.1ms             |
  | RequestLogger start:                         < 0.01ms            |
  | Router.matchRoute (linear scan, 8 entries):  < 0.01ms            |
  | [No auth required for redirect route]                            |
  | RateLimiter.Allow (Redis INCR):              ~0.5-1ms            | ← network to Redis
  | Set X-Correlation-ID on forwarded request:   < 0.01ms            |
  | CircuitBreaker.Do (mutex acquire, CLOSED):   < 0.001ms           |
  | httputil.ReverseProxy.ServeHTTP:             ~1-3ms              | ← url-service cache hit
  | ResponseWriter capture status:               < 0.01ms            |
  | RequestLogger log write:                     < 0.1ms             |
  | Network send:                                ~0.1ms               |
  | TOTAL gateway overhead (above direct call):  ~1-2ms typical      |
  |                                              < 5ms p99 ✓         |
  └──────────────────────────────────────────────────────────────────┘
```



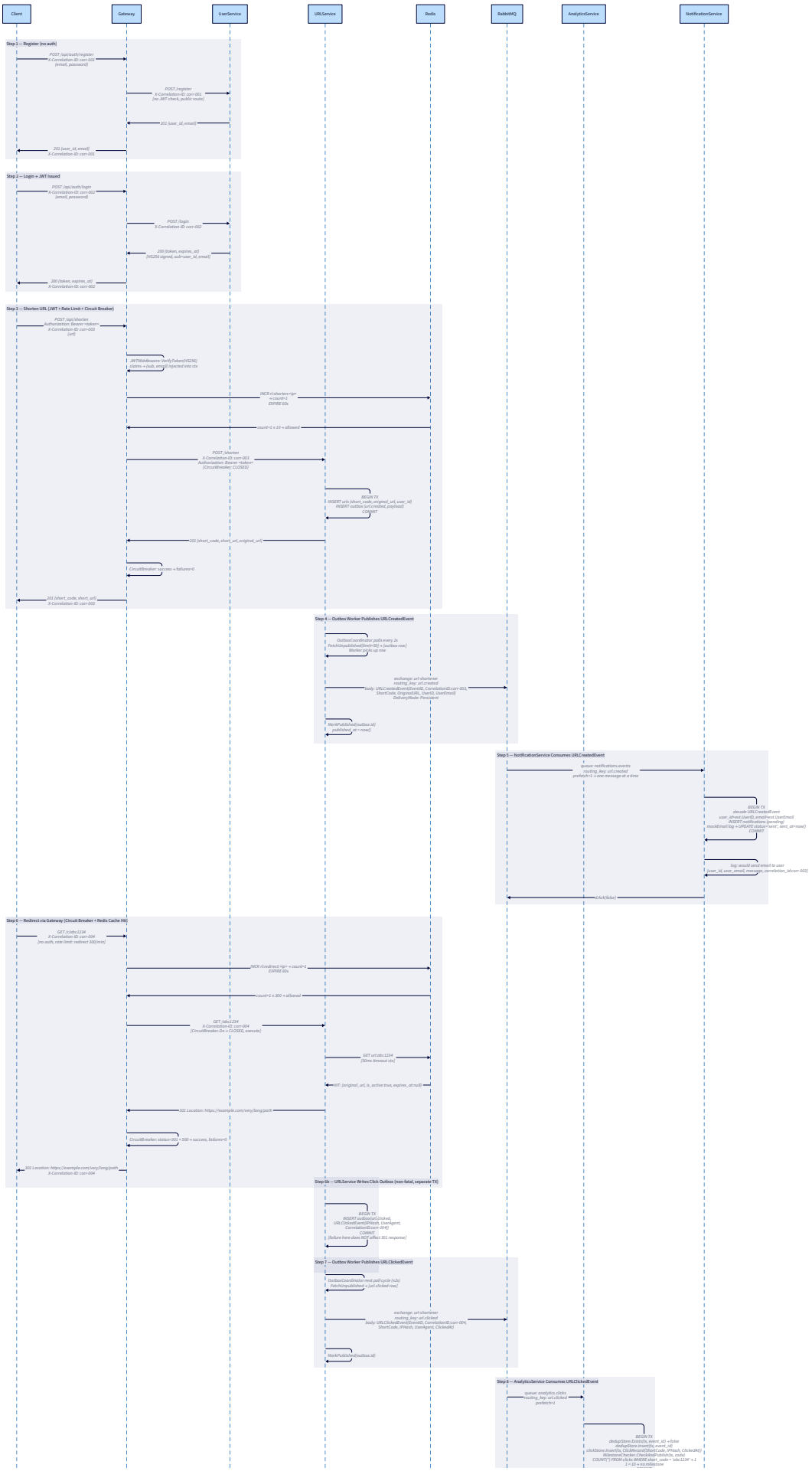
```
Rate limiter Redis key lifecycle:
  Request 1 (t=0):
    INCR "rl:shorten:192.168.1.1" → 1
    EXPIRE "rl:shorten:192.168.1.1" 60
    allowed=true
  Requests 2-10 (t=1s to t=10s):
    INCR → 2, 3, ..., 10
    (EXPIRE not called; count==1 is false)
    allowed=true
  Request 11 (t=11s):
    INCR → 11
    11 > 10 → allowed=false, retryAfter=60
    Response: 429 + "Retry-After: 60"
  t=60s: Redis key expires automatically
  Request 12 (t=61s):
    INCR "rl:shorten:192.168.1.1" → 1 (key re-created)
    EXPIRE "rl:shorten:192.168.1.1" 60
    allowed=true (new window)
```

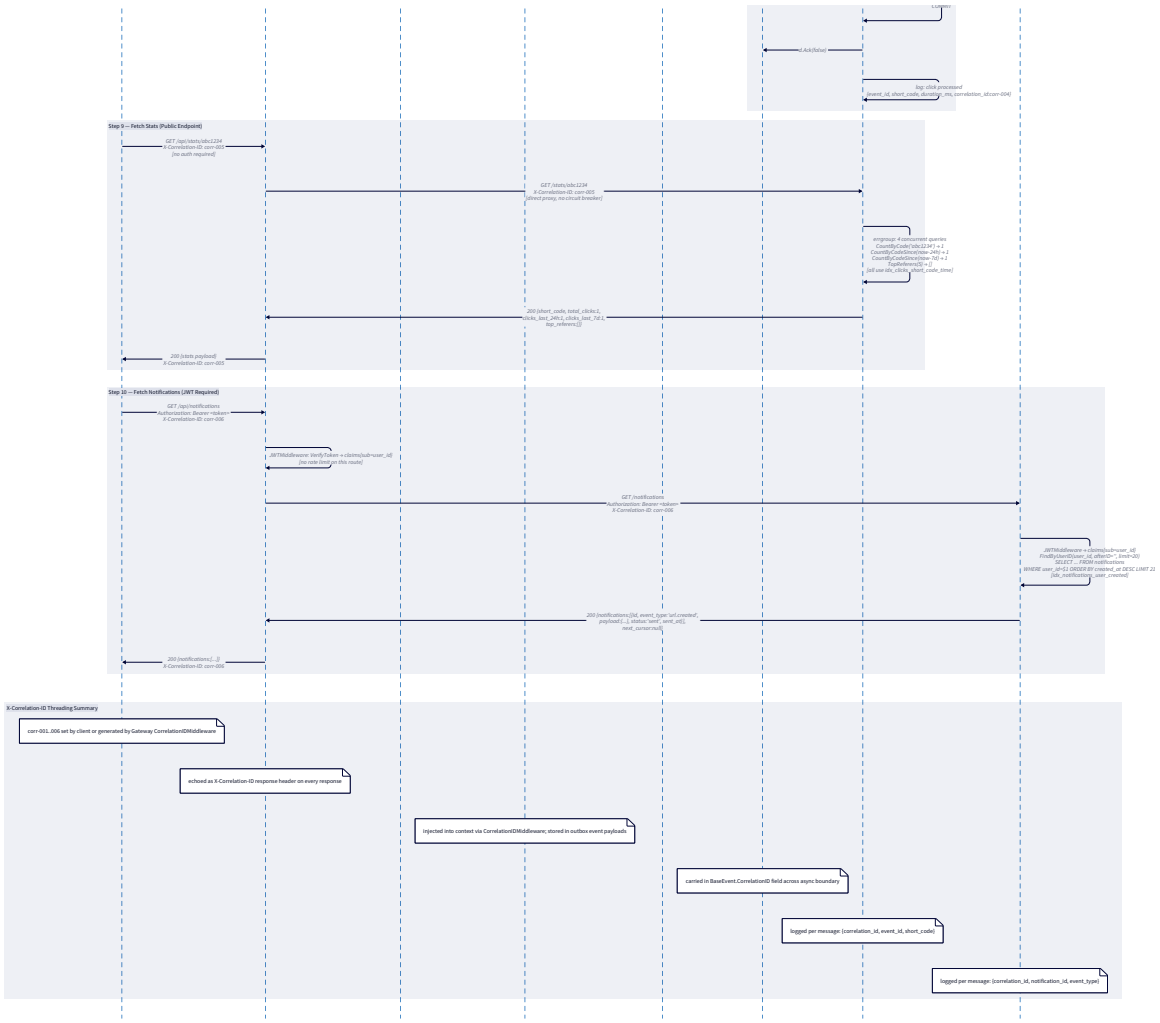
9.1 `go.mod` Additions

```
// services/notification-service/go.mod
module github.com/yourhandle/url-shortener/notification-service
go 1.23
require (
    github.com/yourhandle/url-shortener/shared/events v0.0.0
    github.com/yourhandle/url-shortener/shared/logger v0.0.0
    github.com/yourhandle/url-shortener/shared/auth   v0.0.0
    github.com/jackc/pgx/v5 v5.6.0
    github.com/rabbitmq/amqp091-go v1.10.0
    github.com/golang-jwt/jwt/v5 v5.2.1
)
replace (
    github.com/yourhandle/url-shortener/shared/events => ../../shared/events
    github.com/yourhandle/url-shortener/shared/logger => ../../shared/logger
    github.com/yourhandle/url-shortener/shared/auth   => ../../shared/auth
)
// gateway/go.mod
module github.com/yourhandle/url-shortener/gateway
go 1.23
require (
    github.com/yourhandle/url-shortener/shared/logger v0.0.0
    github.com/yourhandle/url-shortener/shared/auth   v0.0.0
    github.com/golang-jwt/jwt/v5 v5.2.1
    github.com/redis/go-redis/v9 v9.6.0
    github.com/alicebob/miniredis/v2 v2.32.0  // test only
)
replace (
    github.com/yourhandle/url-shortener/shared/logger => ../shared/logger
    github.com/yourhandle/url-shortener/shared/auth   => ../shared/auth
)
```


Gateway Routing Table
Method + Path Pattern → Upstream + Rewrite + Auth + Rate Limit + Circuit Breaker
GET /api/notifications
GET
/api/notifications
strip /api
→ /notifications
auth: ✓ JWT
rl: —
CB: X
/notifications
notification-service :8084
DELETE /api/urls/{code}
DELETE
/api/urls/{code}
strip /api
extract {code}
→ /urls/{code}
auth: ✓ JWT
rl: —
CB: X
/urls/{code}
JWT Required Routes (shared/auth.VerifyToken — local HMAC-SHA256, no user-service call per request)
GET /api/urls
GET
/api/urls
strip /api
→ /urls
auth: ✓ JWT
rl: —
CB: X
/urls
POST /api/shorten
POST
/api/shorten
strip /api
→ /shorten
auth: ✓ JWT
rl: "shorten" 10 req/60s
CB: X
/shorten
url-service :8081
routingTable []Route
method_path_1
method_path_2
method_path_3
method_path_4
method_path_5
method_path_6
method_path_7
method_path_8
GET /api/stats/{code}
GET
/api/stats/{code}
strip /api
extract {code}
→ /stats/{code}
auth: X
rl: —
CB: X
/stats/{code}
analytics-service :8082
POST /api/auth/register
POST
/api/auth/register
strip /api
→ /register
auth: X
rl: —
CB: X
/register
user-service :8083
POST /api/auth/login
POST
/api/auth/login
strip /api
→ /login
auth: X
rl: —
CB: X
/login
bucket="shorten" 10/60s
/{code}
GET /r/{code} ← hot redirect path
GET
/r/{code}
strip /r
(rewrite /{code})
via CB
CircuitBreaker (url-service only) maxFail=5 openTimeout=30s failureWindow=10s


```
Correlation ID propagation across all five processes:
  Client               Gateway             url-service        analytics-service
    |                     |                     |                     |
    ├─X-Corr-ID: abc-123──▶                    |                     |
    |                     ├─CorrelationIDMiddleware: stores "abc-123"   |
    |                     ├─RequestLogger: logs {correlation_id:"abc-123",...}
    |                     ├─X-Corr-ID: abc-123──▶                     |
    |                     |               ├─CorrelationIDMiddleware    |
    |                     |               ├─handler logs {correlation_id:"abc-123"}
    |                     |               ├─INSERT outbox(correlation_id:"abc-123")
    |                     |               |                     |
    |                     |         [outbox poll → AMQP publish]|
    |                     |           URLClickedEvent.CorrelationID = "abc-123"
    |                     |                                 ├─consumer.processDelivery
    |                     |                                 ├─log {correlation_id:"abc-123"}
    |                     |                                 |
  async boundary:         |
    |                     |
  URLCreatedEvent.CorrelationID = "abc-123" → notifications.events
    |                                             notification-service
    |                                             ├─processDelivery
    |                                             ├─log {correlation_id:"abc-123"}
```


Client
Gateway
UserService
URLService
Redis
RabbitMQ
AnalyticsService
NotificationService
Step 1 — Register (no auth)
POST /api/auth/register
X-Correlation-ID: corr-001
{email, password}
POST /register
X-Correlation-ID: corr-001
[no JWT check, public route]
201 {user_id, email}
201 {user_id, email}
X-Correlation-ID: corr-001
Step 2 — Login → JWT issued
POST /api/auth/login
X-Correlation-ID: corr-002
{email, password}
POST /login
X-Correlation-ID: corr-002
200 {token, expires_at}
[HS256 signed, sub=user_id, email]
200 {token, expires_at}
X-Correlation-ID: corr-002
Step 3 — Shorten URL (JWT + Rate Limit + Circuit Breaker)
POST /api/shorten
Authorization: Bearer <token>
X-Correlation-ID: corr-003
{url}
JWTMiddleware: VerifyToken(HS256)
claims = {sub, email} injected into ctx
INCR rl:shorten:<ip>
= count+1
EXPIRE 60s
count+1 ≤ 10 = allowed
POST /shorten
X-Correlation-ID: corr-003
Authorization: Bearer <token>
[CircuitBreaker: CLOSED]
BEGIN TX
INSERT urls (short_code, original_url, user_id)
INSERT outbox (url.created, payload)
COMMIT
201 {short_code, short_url, original_url}
CircuitBreaker: success = failures=0
201 {short_code, short_url, original_url}
X-Correlation-ID: corr-003
Step 4 — Outbox Worker Publishes URLCreatedEvent
OutboxCoordinator polls every 2s
FetchUnpublished(limit=50) → [outbox row]
Worker picks up row
exchange: url.shortener
routing_key: url.created
body: URLCreatedEvent{EventID, CorrelationID:corr-003,
ShortCode, OriginalURL, UserID, UserEmail}
DeliveryMode: Persistent
MarkPublished(outbox.id)
published_at = now()
Step 5 — NotificationService Consumes URLCreatedEvent
queue: notifications.events
routing_key: url.created
prefetch=1 → one message at a time
BEGIN TX
decode URLCreatedEvent
user_id=evt.UserID, email=evt.UserEmail
INSERT notifications (pending)
mockEmail log → UPDATE status='sent', sent_at=now()
COMMIT
log: would send email to user
{user_id, user_email, message, correlation_id:corr-003}
d.Ack(false)
Step 6 — Redirect via Gateway (Circuit Breaker + Redis Cache Hit)
GET /r/abc1234
X-Correlation-ID: corr-004
[no auth, rate limit: redirect 300/min]
INCR rl:redirect:<ip> → count+1
EXPIRE 60s
count+1 ≤ 300 = allowed
GET /abc1234
X-Correlation-ID: corr-004
[CircuitBreaker.Do = CLOSED, execute]
GET url:abc1234
[50ms timeout ctx]
HIT: {original_url, is_active:true, expires_at:null}
301 Location: https://example.com/very/long/path
CircuitBreaker: state=302 = 500 = success, failures=0
301 Location: https://example.com/very/long/path
X-Correlation-ID: corr-004
Step 6b — URLService Writes Click Outbox (non-fatal, separate TX)
BEGIN TX
INSERT outbox(url.clicked,
URLClickedEvent{IPHash, UserAgent,
CorrelationID:corr-004})
COMMIT
[failure here does NOT affect 301 response]
Step 7 — Outbox Worker Publishes URLClickedEvent
OutboxCoordinator next poll cycle (≤2s)
FetchUnpublished → url.clicked row]
exchange: url.shortener
routing_key: url.clicked
body: URLClickedEvent{EventID, CorrelationID:corr-004,
ShortCode, IPHash, UserAgent, ClickedAt}
MarkPublished(outbox.id)
Step 8 — AnalyticsService Consumes URLClickedEvent
queue: analytics.clicks
routing_key: url.clicked
prefetch=1
BEGIN TX
dedupStore.Exists(ctx, event_id) = false
dedupStore.Insert(ctx, event_id)
clickStore.Insert(ctx, ClickRecord{ShortCode, IPHash, ClickedAt})
MilestoneChecker.CheckAndPublish(ctx, code)
COUNT(*) FROM clicks WHERE short_code = 'abc1234' = 1
1 = 10 = no milestone




```
Anti-pattern guard — gateway contains ZERO domain logic:
  FORBIDDEN imports in gateway/:
    github.com/yourhandle/url-shortener/shared/events  ← domain events
    Any reference to "short_code", "URLRecord", "clicks"
  ALLOWED imports in gateway/:
    github.com/yourhandle/url-shortener/shared/auth   ← JWT verification
    github.com/yourhandle/url-shortener/shared/logger ← structured logging
    github.com/redis/go-redis/v9                       ← rate limiting
    net/http, net/http/httputil                        ← proxy
    sync, time, strings, context                       ← stdlib
  Verification command:
    grep -r "shared/events\|URLRecord\|short_code\|ClickRecord" gateway/ \
      | grep -v "_test.go" && echo "FAIL: domain logic in gateway" \
      || echo "PASS: gateway is domain-free"
  Additional anti-pattern: notification service MUST NOT call other services
    grep -r "url-service\|analytics-service\|user-service\|8081\|8082\|8083" \
      services/notification-service/*.go | grep -v "_test.go" \
      && echo "FAIL: notification-service makes cross-service call" \
      || echo "PASS: notification-service self-contained"
```

Project Structure: URL Shortener (Microservices)

Directory Tree

```
url-shortener/                                  # Monorepo root
│
├── go.work                                     # Go workspace (all modules) — M1
│
├── shared/
│   ├── events/
│   │   ├── go.mod                              # module: shared/events — M1
│   │   ├── events.go                           # All domain event structs + routing key constants — M1
│   │   └── events_test.go                      # JSON round-trip + constant tests — M1
│   │
│   ├── logger/
│   │   ├── go.mod                              # module: shared/logger — M1
│   │   └── logger.go                           # JSON logger + WithCorrelationID + RequestLogger middleware — M1, M5
│   │
│   └── auth/
│       ├── go.mod                              # module: shared/auth — M2
│       ├── auth.go                             # Claims struct, TokenIssuer interface, VerifyToken, ErrTokenInvalid
— M2
│       └── middleware.go                       # JWTMiddleware, ClaimsFromContext, claimsKey — M2
│
├── services/
│   │
│   ├── url-service/
│   │   ├── go.mod                              # module: url-service; requires shared/events, auth, logger, pgx,
redis, amqp, jwt — M1–M3
│   │   ├── main.go                             # Startup sequence, route wiring, outbox coordinator start, graceful
shutdown — M1–M3
│   │   ├── config.go                           # Config struct: DB/Redis/AMQP/JWT/ShortURLBase/Outbox fields,
loadConfig() — M1–M3
│   │   ├── db.go                               # NewDBPool (pgxpool, 10s ping, MaxConns=10) — M1
│   │   ├── redis.go                            # NewRedisClient (non-fatal ping, always returns client) — M1
│   │   ├── rabbitmq.go                         # NewRabbitMQConn (retry+backoff), declareExchange — M1
│   │   ├── health.go                           # NewHealthHandler (pre-encoded JSON, no DB check) — M1
│   │   ├── migration.sql                       # urls + outbox tables, all 4 indexes — M3
│   │   ├── store.go                            # URLRecord, URLRepository interface, pgxURLStore
(Insert/FindByCode/FindByUserID/Deactivate) — M3
│   │   ├── outbox_store.go                     # OutboxRecord, OutboxRepository interface, pgxOutboxStore
(InsertWithURL/InsertEvent/FetchUnpublished/MarkPublished) — M3
│   │   ├── base62.go                           # base62Alphabet const, Encode(*big.Int), Decode() — M3
│   │   ├── codegen.go                          # ShortCodeGenerator, Generate() using crypto/rand — M3
│   │   ├── cache.go                            # CachedURL, RedisCache (Get/Set/Del, error-isolated, TTL logic) — M3
│   │   ├── publisher.go                        # RabbitMQPublisher interface, amqpPublisher with sync.Mutex — M3
│   │   ├── outbox.go                           # OutboxCoordinator, 3-worker pool, Run(), poll(), runWorker(),
graceful drain — M3
│   │   ├── handler.go                          # Handler struct; Shorten/Redirect/ListURLs/DeleteURL;
CorrelationIDMiddleware; hashIP; newUUID — M3
│   │   ├── validate.go                         # validateURL (scheme + host check) — M3
│   │   ├── errors.go                           # ErrShortCodeConflict, ErrURLNotFound, ErrNotOwner; writeError,
writeJSON, isPgUniqueViolation — M3
│   │   ├── url_test.go                         # Base62 round-trip, codegen, validateURL, handler unit tests (mock
stores), RedisCache tests (miniredis), outbox worker tests — M3
│   │   ├── bench_test.go                       # BenchmarkRedirectCacheHit, BenchmarkShortCodeGenerate — M3
│   │   └── Dockerfile                          # Multi-stage Go build, scratch final image, EXPOSE 8080 — M1
│   │
│   ├── analytics-service/
│   │   ├── go.mod                              # module: analytics-service; requires shared/events, logger, pgx,
amqp, golang.org/x/sync — M1, M4
│   │   ├── main.go                             # Startup: DB→migrations→AMQP→queues→stores→consumer goroutine→HTTP —
M1, M4
│   │   ├── config.go                           # Config: DATABASE_URL, RABBITMQ_URL, IPHashSalt, AMQPPrefetchCount=1
— M1, M4
│   │   ├── db.go                               # NewDBPool reused from M1 pattern — M1
│   │   ├── rabbitmq.go                         # NewRabbitMQConn + DeclareAnalyticsQueue ("analytics.clicks" +
"url.clicked" binding) — M1
│   │   ├── health.go                           # NewHealthHandler (always 200, no consumer state) — M1
│   │   ├── migration.sql                       # clicks + milestones + processed_events tables, 5 indexes — M4
│   │   ├── store.go                            # ClickRecord, MilestoneRecord, StatsResult; ClickRepository,
MilestoneRepository, DeduplicationRepository interfaces + pgx impls — M4
│   │   ├── consumer.go                         # ClickConsumer, Run() (prefetch=1, manual ack), processDelivery()
(dedup→click→milestone in one tx), panic recovery, newUUID — M4
```

```
|   |   ├── milestone.go                          # MilestoneChecker, CheckAndPublish() (thresholds 10/100/1000, tx-scoped COUNT, ON CONFLICT DO NOTHING) — M4
|   |   ├── publisher.go                          # AnalyticsPublisher interface, amqpAnalyticsPublisher.PublishMilestone() — M4
|   |   ├── handler.go                            # StatsHandler; Stats() (errgroup 4 concurrent queries); Timeline() (day/hour validation) — M4
|   |   ├── errors.go                             # writeError, writeJSON, truncate helpers — M4
|   |   ├── haship.go                             # hashIP(remoteAddr, salt) → SHA-256 hex; newUUID — M4
|   |   ├── analytics_test.go                     # MilestoneChecker unit tests (mock stores), consumer idempotency test, StatsHandler tests (mock), integration tests — M4
|   |   └── Dockerfile                            # Multi-stage Go build, scratch final image — M1
|   |
|   ├── user-service/
|   |   ├── go.mod                                # module: user-service; requires shared/auth, logger, pgx, bcrypt, jwt — M1, M2
|   |   ├── main.go                               # Startup: DB→migrations→stores→handler→routes (register/login/me) — M1, M2
|   |   ├── config.go                             # Config: DATABASE_URL, JWTSecret, BCryptCost=12, TokenTTL=24h — M1, M2
|   |   ├── db.go                                 # NewDBPool — M1
|   |   ├── health.go                             # NewHealthHandler — M1
|   |   ├── migration.sql                         # users table (id UUID PK, email UNIQUE, password_hash, created_at) + idx_users_email — M2
|   |   ├── store.go                              # User struct, UserRepository interface, pgxUserStore (Insert/FindByEmail), isPgUniqueViolation — M2
|   |   ├── password.go                           # PasswordHasher interface, bcryptHasher (Hash/Verify, ErrPasswordMismatch) — M2
|   |   ├── token.go                              # jwtTokenIssuer (Issue/Verify), dummy-hash constant for timing safety — M2
|   |   ├── handler.go                            # Handler struct; Register/Login (timing-safe)/Me methods; request/response types — M2
|   |   ├── validate.go                           # validateEmail (regex), validatePassword (min 8 chars) — M2
|   |   ├── errors.go                             # ErrDuplicateEmail, ErrUserNotFound, ErrPasswordMismatch, ErrTokenInvalid; writeError — M2
|   |   ├── user_test.go                          # Validator tests, bcryptHasher tests, jwtTokenIssuer tests, handler unit tests (mock store), integration round-trip test — M2
|   |   └── Dockerfile                            # Multi-stage Go build, scratch final image — M1
|   |
|   └── notification-service/
|       ├── go.mod                                # module: notification-service; requires shared/events, auth, logger, pgx, amqp, jwt — M1, M5
|       ├── main.go                               # Startup: DB→migrations→AMQP→queues→store→consumer goroutine→HTTP (JWTMiddleware on /notifications) — M1, M5
|       ├── config.go                             # Config: DATABASE_URL, RABBITMQ_URL, JWTSecret — M1, M5
|       ├── db.go                                 # NewDBPool — M1
|       ├── rabbitmq.go                           # NewRabbitMQConn + DeclareNotificationQueue ("notifications.events" + 3 bindings) — M1
|       ├── health.go                             # NewHealthHandler — M1
|       ├── migration.sql                         # notifications table (id, user_id, event_type, payload JSONB, status, created_at, sent_at) + idx_notifications_user_created — M5
|       ├── store.go                              # NotificationRecord, NotificationRepository interface, pgxNotificationStore (Insert 2-step tx / FindByUserID cursor pagination) — M5
|       ├── consumer.go                           # NotificationConsumer, Run() (prefetch=1), processDelivery() (routing-key switch, mockEmail log, insert, ack/nack), panic recovery — M5
|       ├── handler.go                            # NotificationHandler, ListNotifications() (JWT claims→FindByUserID→paginated response) — M5
|       ├── errors.go                             # writeError, writeJSON, truncate — M5
|       ├── notification_test.go                  # Consumer unit tests (mock store, all event types, malformed JSON, unknown key), handler unit tests, integration test — M5
|       └── Dockerfile                            # Multi-stage Go build, scratch final image — M1
|
├── gateway/
|   ├── go.mod                                    # module: gateway; requires shared/auth, logger, go-redis, jwt, miniredis (test) — M1, M5
|   ├── main.go                                   # Startup: config→Redis→proxies→CircuitBreaker→RateLimiter→Gateway→mux→upstream health checks→serve — M1, M5
|   ├── config.go                                 # Config: all upstream URLs, RedisURL, JWTSecret, CB settings, RateLimits map — M1, M5
|   ├── health.go                                 # NewHealthHandler — M1
|   ├── router.go                                 # Route struct, routingTable (8 routes), Gateway struct, ServeHTTP (match→auth→rl→proxy), matchRoute() — M5
|   ├── proxy.go                                  # UpstreamProxy (httputil.ReverseProxy wrapper), NewUpstreamProxy, path rewriting, ErrorHandler — M5
```

```
│   ├── middleware.go                          # CorrelationIDMiddleware, chain() helper — M5
│   ├── ratelimit.go                           # RateLimiter interface, redisTokenBucket (INCR+EXPIRE, fail-open),
clientIP, buildRateLimitKey — M5
│   ├── circuitbreaker.go                      # State enum, CircuitBreaker (sync.Mutex, 3-state machine),
NewCircuitBreaker, Do(), ErrCircuitOpen — M5
│   ├── jwtmiddleware.go                       # GatewayJWTMiddleware (wraps shared/auth.JWTMiddleware, adds
correlation log) — M5
│   ├── errors.go                              # writeError, writeJSON helpers — M5
│   ├── gateway_test.go                        # CircuitBreaker unit tests (all transitions, concurrency),
RateLimiter tests (miniredis), router matching tests, notification handler tests — M5
│   └── Dockerfile                             # Multi-stage Go build, scratch final image — M1
│
├── scripts/
│   ├── smoke_test.sh                          # Health-check all 5 /health endpoints — M1
│   └── e2e_test.sh                            # Full register→login→shorten→redirect→stats→notifications→rate-
limit→correlation-ID test — M5
│
├── docker-compose.yml                         # Full container mesh: 4 PostgreSQL, RabbitMQ, Redis, 4 services,
gateway — M1 (extended M2–M5)
└── README.md                                  # Prerequisites, startup commands, port table, RabbitMQ credentials —
M1
```

Creation Order

1. Project Scaffold (30 min)

- Create all directories: `shared/events/`, `shared/logger/`, `shared/auth/`, `services/url-service/`, `services/analytics-service/`, `services/user-service/`, `services/notification-service/`, `gateway/`, `scripts/`
- `go.work` (workspace root)
- `shared/events/go.mod`
- `shared/logger/go.mod`
- All five service `go.mod` files with empty `require` blocks (fill in as needed per phase)
- Stub `main.go` in each service/gateway: `package main; func main() {}`

2. Shared Packages — Events and Logger (30 min)

- `shared/events/events.go` — BaseEvent, URLCreatedEvent, URLClickedEvent, URLDeletedEvent, MilestoneReachedEvent, EventType constants
- `shared/events/events_test.go` — JSON round-trip tests, constant value tests
- `shared/logger/logger.go` — `New()` (JSON handler, service field) — M5 extensions added later

3. Infrastructure Config — docker-compose.yml (1–1.5h)

- `docker-compose.yml` — all 4 PostgreSQL instances (ports 5432–5435), RabbitMQ (5672, 15672), Redis (6379), all 5 app containers with healthchecks and depends_on chains
- Verify: `docker compose config` → exit 0; `docker compose up` infra containers → all healthy

4. Per-Service Foundations: db.go, redis.go, rabbitmq.go, health.go, config.go, main.go, Dockerfile (1.5–2h)

- **url-service**: `config.go`, `db.go`, `redis.go`, `rabbitmq.go`, `health.go`, `main.go` (health route only), `Dockerfile`
- **analytics-service**: `config.go`, `db.go`, `rabbitmq.go` (+ `DeclareAnalyticsQueue`), `health.go`, `main.go`, `Dockerfile`
- **user-service**: `config.go`, `db.go`, `health.go`, `main.go`, `Dockerfile`
- **notification-service**: `config.go`, `db.go`, `rabbitmq.go` (+ `DeclareNotificationQueue`), `health.go`, `main.go`, `Dockerfile`
- **gateway**: `config.go` (stub), `health.go`, `main.go` (health only), `Dockerfile`
- `scripts/smoke_test.sh`
- `README.md`
- **Checkpoint M1**: `docker compose up --build -d` → all 10 containers healthy; all `/health` return `200 {"status":"ok"}`

5. Shared Auth Package (1h)

- `shared/auth/go.mod`
- `shared/auth/auth.go` — Claims, TokenIssuer interface, VerifyToken, ErrTokenInvalid, TestClaimsKey
- `shared/auth/middleware.go` — JWTMiddleware, ClaimsFromContext, claimsKey
- Update `go.work` to include `./shared/auth`

6. User Service — Full Implementation (3–4h)

- `services/user-service/migration.sql`
- `services/user-service/errors.go`
- `services/user-service/store.go`
- `services/user-service/password.go`
- `services/user-service/token.go`
- `services/user-service/validate.go`
- `services/user-service/handler.go`
- Extend `services/user-service/config.go` (JWTSecret, BCryptCost, TokenTTL)
- Extend `services/user-service/main.go` (runMigrations, full route wiring)
- `services/user-service/user_test.go`
- **Checkpoint M2**: register→login→GET /me round-trip succeeds; duplicate email → 409; wrong password → 401 (same body as unknown email)

7. URL Service — Full Implementation (6–8h)

- `services/url-service/migration.sql`
- `services/url-service/errors.go`
- `services/url-service/validate.go`
- `services/url-service/base62.go`
- `services/url-service/codegen.go`
- `services/url-service/store.go`
- `services/url-service/outbox_store.go`
- `services/url-service/cache.go`
- `services/url-service/publisher.go`
- `services/url-service/outbox.go`
- `services/url-service/handler.go`
- Extend `services/url-service/config.go` (JWTSecret, ShortURLBase, OutboxPollInterval)
- Extend `services/url-service/main.go` (runMigrations, all routes, outbox coordinator start)
- `services/url-service/url_test.go`
- `services/url-service/bench_test.go`
- **Checkpoint M3**: POST /shorten → 201; GET /:code → 301 (DB then Redis); DELETE → 204 + 410; outbox rows transition to `published_at != NULL` within 5s; `EXPLAIN ANALYZE` shows index scans

8. Analytics Service — Full Implementation (4–5h)

- `services/analytics-service/migration.sql`
- `services/analytics-service/errors.go`
- `services/analytics-service/haship.go`
- `services/analytics-service/store.go`
- `services/analytics-service/publisher.go`
- `services/analytics-service/milestone.go`
- `services/analytics-service/consumer.go`
- `services/analytics-service/handler.go`
- Extend `services/analytics-service/config.go` (IPHashSalt)

- Extend `services/analytics-service/main.go` (runMigrations, consumer goroutine, /stats routes)
- `services/analytics-service/analytics_test.go`
- **Checkpoint M4**: same `event_id` processed twice → click count = 1; 10 redirects → `milestones` row milestone=10; GET /stats → 200 with correct counts; `EXPLAIN ANALYZE` shows index scans

9. Extended shared/logger (30 min)

- Extend `shared/logger/logger.go` — add `WithCorrelationID`, `CorrelationIDFromContext`, `ContextWithCorrelationID`, `RequestLogger`, `responseWriter`
- Verify `go build ./...` from root still exits 0

10. Notification Service — Full Implementation (2–3h)

- `services/notification-service/migration.sql`
- `services/notification-service/errors.go`
- `services/notification-service/store.go`
- `services/notification-service/consumer.go`
- `services/notification-service/handler.go`
- Extend `services/notification-service/config.go` (JWTSecret)
- Extend `services/notification-service/main.go` (runMigrations, consumer goroutine, /notifications route)
- `services/notification-service/notification_test.go`
- **Checkpoint M5-a**: URLCreatedEvent consumed → notifications row inserted with status='sent'; GET /notifications with valid JWT → 200 with notification item

11. API Gateway — Full Implementation (4–5h)

- Extend `gateway/config.go` (all fields: upstream URLs, Redis, JWT, CB settings, RateLimits map)
- `gateway/errors.go`
- `gateway/middleware.go`
- `gateway/proxy.go`
- `gateway/router.go`
- `gateway/jwtmiddleware.go`
- `gateway/ratelimit.go`
- `gateway/circuitbreaker.go`
- Extend `gateway/main.go` (full wiring: Redis client, proxies, CB, RL, Gateway, mux, upstream health checks)
- `gateway/gateway_test.go`
- `scripts/e2e_test.sh`
- **Checkpoint M5-b**: `bash scripts/e2e_test.sh` → all assertions pass; 11th POST /api/shorten → 429; 6th request after url-service stop → 503; circuit reopens after 30s probe; `X-Correlation-ID` in logs of all services for same request

---

File Count Summary

Category	Count
`shared/` package files	8
`url-service/` files	18
`analytics-service/` files	16
`user-service/` files	15
`notification-service/` files	14
`gateway/` files	13
Root / scripts / infra	3
Total files	**87**
Directories	**14**
Estimated lines of code	**~6,500–8,000**