Engineering Specification: Timetravel v2 Service

🏗️ Insurance Platform Reliability & Audit System

Author: Navita Dandotiya (navita.dandotiya@gmail.com)
Date: 2026-02-16
Scope: v2 Policyholder Records, Audit, and Observability

🎯 System Purpose

This platform stores, audits, and reconstructs policy risk data used during underwriting. It enables:

- historical reconstruction (time travel queries)
- compliance & regulatory audits
- risk recalculation & premium adjustments
- safe global rollout with version & feature controls

📐 Proposal

This system is designed as a **reliable, auditable record-keeping service** for an insurance platform. It stores underwriting data while preserving a complete history of how each record evolves over time.

The architecture exposes two API versions to maintain backward compatibility while introducing advanced capabilities.

Version 1 endpoints provide simple create, update, and retrieval of records. These behave exactly like the original system to ensure no disruption to existing clients.

Version 2 introduces **record versioning and time-travel queries**. Every change creates a new immutable version, allowing us to answer critical compliance questions such as *what we knew*, *when we knew it*, and *when the change actually occurred*.

Data is persisted in **SQLite**, ensuring durability across restarts while keeping the deployment lightweight.

To support auditability and regulatory compliance, each version stores:

- effective timestamps,
- change timestamps,
- version numbers,
- and optional metadata for traceability.

The service is structured using a layered design:

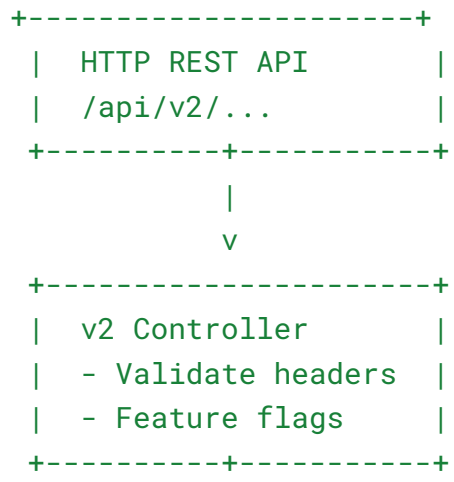
- handlers manage HTTP requests,
- services enforce business rules,
- repositories handle persistence,
- and feature flags control rollout between v1 and v2 behavior.

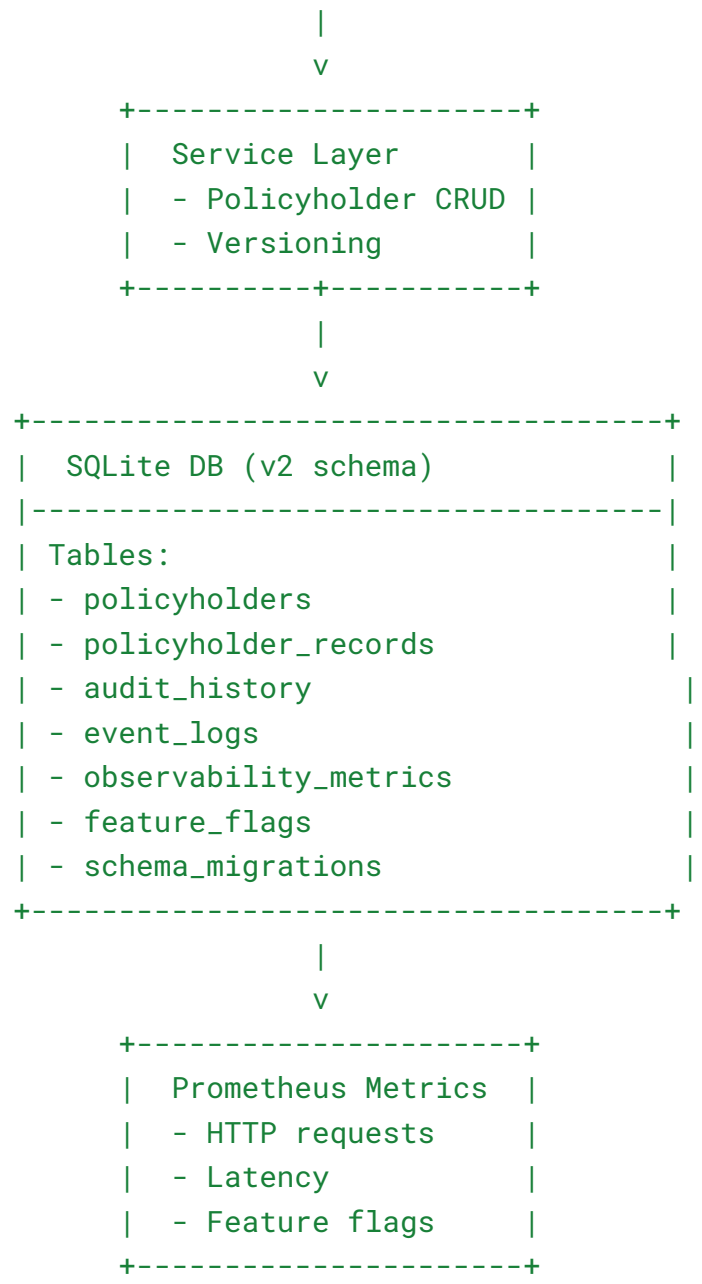
This design ensures:

- backward compatibility,
- full audit history,
- compliance readiness,
- and scalability for future global expansion.

🧩 Core Entities (ERD)

Entities & Relationships





System Overview

The v2 service extends v1 to provide:

- **Policyholder records with versioning**
- **Audit trail for every record change**
- **Event logging**
- **Feature-flag controlled rollout**

- **Observability metrics with Prometheus**
- **SQLite persistence layer**

The service exposes a REST API (`/api/v1/…` and `/api/v2/…`) with consistent ID types (`int64`) and versioned records.

🧩 Entities

Table	Description
`policyholders`	Stores base info for each policyholder
`policyholder_records`	Versioned records per policyholder
`audit_history`	Tracks changes with versioning and timestamps
`event_logs`	Action logs for integration & audit
`observability_metrics`	Stores Prometheus metrics for persistence
`feature_flags`	Controls feature rollout and observability
`schema_migrations`	Tracks schema versions for migration safety

4️⃣ Feature Flags

Flag	Default	Purpose
`enable_v2_api`	true	Expose `/api/v2/...` endpoints
`enable_audit_logging`	true	Enable audit_history table logging

`enable_metrics`	true	Enable Prometheus observability metrics

Feature Flag Strategy:

- Boolean flags control incremental rollout
- Flags can be toggled per region

Rollout percentage can be set dynamically.

Feature flags allow safe rollout:

Feature	v1	v2	Rollout
New Risk Engine	❌	✔	10% → 100%
Realtime Dashboard	❌	✔	region-based
Event Streaming	❌	✔	beta users
Advanced Premium Model	✔	✔	staged

🚀 API Versioning Strategy (v1 & v2)

Base path: `/api/v{version}/...`

- **v1**: Legacy endpoints (`/api/v1/...`)
- **v2**: Enhanced endpoints with:
    - `int64` IDs for policyholders and records
    - Versioned records

    - Audit logging
    - Feature flags support

Endpoints Examples:

Method	Path	Feature
GET	/api/v2/records/{id}	Fetch latest record
POST	/api/v2/records/{id}	Create/update record
GET	/api/v2/records/{id}/versions	Fetch record versions
GET	/api/v2/records/{id}/versions/{ver}	Fetch specific version
GET	/metrics	Prometheus metrics

Deprecation Policy:

- v1 will remain active until v2 adoption ≥80%
- Logs and metrics will track usage by API version

URL Versioning

```
/api/v1/policies
```

```
/api/v2/policies
```

Version Differences

V1

✔ basic CRUD
✔ audit logging
✔ risk profile updates

V2

✔ time travel queries
✔ event streaming
✔ enhanced risk scoring
✔ realtime dashboard support

🌍 Global Rollout Strategy

Rollout

1. Enable `enable_v2_api` feature flag in **staging**
2. Smoke test APIs: `/health`, `/records`
3. Enable metrics collection (`enable_metrics`)
4. Roll out v2 to **10% of traffic** (canary)
5. Monitor success metrics (latency, error rate, DB writes)
6. Incrementally increase traffic until **100% rollout**

Rollback

- Toggle `enable_v2_api` off → traffic reverts to v1
- Database writes are **idempotent**; v2 writes tracked in `audit_history`
- Optionally restore DB snapshot if schema changes cause failures

Phase 1 — Internal & Beta

- enable v2 features via flags
- limited rollout
- monitor metrics

Phase 2 — Regional Rollout

- enable by country/region
- measure latency & adoption

Phase 3 — Global Release

- default v2
- maintain v1 for compatibility

🔁 Rollback Strategy

If issues detected:

- ✔ disable feature flag instantly
- ✔ revert to v1 behavior
- ✔ replay event log to restore state
- ✔ use audit history to recover data

🛠️ Development Strategy

Step 1 — Foundation

- ✔ core entities & audit history
- ✔ event logging
- ✔ v1 APIs

Step 2 — Version & Feature Infrastructure

- ✔ feature flag system
- ✔ API version routing

Step 3 — Advanced Capabilities

- ✔ time travel queries
- ✔ event-driven notifications
- ✔ observability metrics

Step 4 — v2 Enhancements

- ✔ realtime streaming
- ✔ enhanced risk engine

📊 Observability & Success Metrics

Platform Health

- API latency < 200ms
- 99.9% uptime
- error rate < 0.5%

Business Success

- policy update turnaround time
- premium recalculation accuracy
- regional adoption growth

Metric	Source	Success Criteria
`http_requests_total`	Prometheus	No request drops, proper counts per endpoint
`http_request_duration_seconds`	Prometheus	95th percentile < 200ms for GET, < 500ms for POST
`feature_flag_evaluations_total`	Prometheus	Evaluations match flag-enabled endpoints
`observability_metrics` DB table	SQLite	Persisted metrics match in-memory counters

Alerting:

- ≥5% error rate triggers alert
- Latency spikes >2× baseline triggers alert
- Missing audit_history entry triggers alert

🚢 Deployment & Docker Setup

Dockerfile:

```
FROM golang:1.26-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o timetravel ./...

FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/timetravel .
EXPOSE 8000
CMD ["./timetravel"]
```

docker-compose.yml (minimal):

```
version: "3.9"
services:
  timetravel:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - ./db:/app/db
    environment:
      - TZ=UTC
```

Testing via curl:

```
# Create/Update records
curl -s -X POST "http://127.0.0.1:8000/api/v2/records/3" \
  -H "Content-Type: application/json" \
  -H "X-User-ID: 10" \
  -d '{"name": "Rainbow Corp 3", "country": "US", "email":
"contact@rainbow.com", "status": "active"}'

# Fetch record
curl -s -X GET "http://127.0.0.1:8000/api/v2/records/2" \
  -H "X-User-ID: 10"
```

```
# Fetch versions
curl -s "http://127.0.0.1:8000/api/v2/records/2/versions" \
  -H "X-User-ID: 10"

# Prometheus metrics
curl -s "http://127.0.0.1:8000/metrics"
```

✅ Exit Criteria (Project Success)

The project is considered complete when:

- ✔ Risk history supports full time travel queries
- ✔ Audit compliance reconstruction works
- ✔ Feature flags enable safe rollout
- ✔ v1 & v2 APIs coexist safely
- ✔ Event-driven updates function correctly
- ✔ Observability dashboards show health & business metrics
- ✔ Rollback & recovery tested successfully
- ✔ System supports global scaling

🎯 Summary

- v2 ensures **type consistency**, **versioned records**, **audit logging**, **observability**
- Feature flags allow **safe gradual rollout**
- Rollback is supported **without data loss**
- Prometheus + SQLite metrics provide **full visibility**

📘 Runbook

📘 Timetravel v2 – Production Runbook

Service Name: Timetravel API
Version: v2
Tech Stack: Go, SQLite, Prometheus
Port: 8000
Metrics Endpoint: `/metrics`
Health Endpoint: `/api/v2/health`

1️⃣ Service Overview

Timetravel v2 provides:

- Policyholder record management
- Versioned records
- Audit history tracking
- Event logging
- Feature flag controlled functionality
- Prometheus observability
- SQLite persistence

2️⃣ Architecture Summary

```
Client
   |
   v
HTTP API (/api/v2)
   |
   v
Controller Layer
   |
   v
Service Layer
   |
```

```
   v
SQLite Database
   |
   +--> audit_history
   +--> event_logs
   +--> observability_metrics
   |
   v
Prometheus /metrics
```

3️⃣ Deployment

Local Run

```
go build ./...
go run main.go
```

Docker

```
docker build -t timetravel .
docker run -p 8000:8000 timetravel
```

Verify Startup

```
curl http://localhost:8000/api/v2/health
```

Expected:

```
200 OK
```

4️⃣ Health Checks

Liveness

```
GET /api/v2/health
```

Checks:

- Application running
- DB connection available

Metrics

```
GET /metrics
```

Verify:

- http_requests_total
- http_request_duration_seconds
- feature_flag_evaluations_total

5️⃣ Feature Flags Operations

Feature flags stored in `feature_flags` table.

View Flags

```
SELECT * FROM feature_flags;
```

Disable v2 API (Rollback)

```
UPDATE feature_flags
SET enabled = 0
WHERE flag_key = 'enable_v2_api';
```

Effect:

- Traffic falls back to v1
- No redeploy required

Disable Metrics

```
UPDATE feature_flags
SET enabled = 0
WHERE flag_key = 'enable_metrics';
```

6️⃣ Common Operational Tasks

Create Record

```
curl -X POST http://localhost:8000/api/v2/records/101 \
-H "Content-Type: application/json" \
-H "X-User-ID: 10" \
-d '{"status":"active"}'
```

Retrieve Record

```
curl http://localhost:8000/api/v2/records/101 \
-H "X-User-ID: 10"
```

Check Versions

```
curl http://localhost:8000/api/v2/records/101/versions \
-H "X-User-ID: 10"
```

7️⃣ Database Operations

Inspect Records

```
SELECT record_id, version, updated_at
FROM policyholder_records;
```

Inspect Audit History

```
SELECT record_id, version, event_type, changed_at
FROM audit_history
ORDER BY changed_at DESC;
```

Verify Foreign Keys Enabled

```
PRAGMA foreign_keys;
```

Expected:

```
1
```

---

8 Observability & Monitoring

Key Metrics

Metric	Meaning	Alert Threshold
http_requests_total	Total API traffic	sudden drop or spike
http_request_duration_seconds	Latency	P95 > 500ms
feature_flag_evaluations_total	Flag usage	mismatch with expected
DB write failures	Logged errors	>5% failure rate

Log Patterns

Successful request:

```
INFO request method=GET path=/records/123 status=200
```

Header errors:

```
ERROR missing X-User-ID header
ERROR invalid X-User-ID header
```

Investigate repeated header errors → possible client misconfiguration.

9️⃣ Incident Response

🔴 High Error Rate (>5%)

1. Check logs
2. Check `/metrics`
3. Verify DB accessible
4. Disable v2 API flag if needed

```
UPDATE feature_flags SET enabled=0 WHERE flag_key='enable_v2_api';
```

🔴 Database Locked / Corrupted

Symptoms:

- SQLite “database is locked”
- Writes failing

Actions:

1. Restart service
2. Check for long-running transactions
3. Validate file permissions
4. Restore from backup if corrupted

🔴 Latency Spike

Check:
`http_request_duration_seconds`

1.
2. Inspect DB write volume
3. Check audit_history table growth
4. Temporarily disable audit logging if necessary:

```
UPDATE feature_flags
SET enabled = 0
WHERE flag_key = 'enable_audit_logging';
```

🔄 Rollback Strategy

Soft Rollback (Recommended)

Disable feature flag:

```
UPDATE feature_flags
SET enabled = 0
WHERE flag_key = 'enable_v2_api';
```

No restart required.

Hard Rollback

```
git revert <commit>
docker rebuild
```

If DB reset required:

```
rm db/timetravel.db
go run main.go
```

⚠ Use only in non-production or controlled environment.

🔟 Backup & Recovery

Backup

```
cp db/timetravel.db db/timetravel_$(date +%F).db
```

Restore

```
cp db/timetravel_backup.db db/timetravel.db
```

Restart service.

1️⃣1️⃣ Performance Expectations

Endpoint	Expected Latency
GET record	<200ms
POST record	<500ms
Metrics endpoint	<50ms

1️⃣2️⃣ Exit Criteria for Stable Production

✔ No DB lock errors
✔ Error rate <1%
✔ P95 latency within SLA
✔ Audit records created for each update
✔ Metrics endpoint healthy
✔ Feature flags operating correctly

1️⃣3️⃣ Escalation Matrix

Severity	Action
Sev-1 (Outage)	Disable v2 flag immediately
Sev-2 (High latency)	Investigate DB, audit logs
Sev-3 (Minor bug)	Patch next release

📌 Quick Troubleshooting Checklist

- Health endpoint 200?
- Metrics endpoint reachable?
- DB file present?
- Foreign keys enabled?
- Feature flags correct?
- Error logs present?

🏁 Summary

This runbook enables:

- Safe production operations
- Controlled rollout & rollback
- Clear monitoring strategy
- Rapid incident mitigation
- Predictable recovery

On-Call Cheat Sheet

🚨 Timetravel v2 – On-Call Cheat Sheet

Service: Timetravel API
Port: 8000
Health: `/api/v2/health`
Metrics: `/metrics`
DB: SQLite (`db/timetravel.db`)

✅ 1. Quick Health Check (30 seconds)

Check service

```
curl -i http://localhost:8000/api/v2/health
```

Expected: `200 OK`

Check metrics

```
curl -s http://localhost:8000/metrics | head
```

Check logs

```
docker logs <container>
# or
tail -f logs/app.log
```

🔥 2. If Production Is Broken

🔴 High Error Rate (>5%)

Symptoms

- 5xx responses
- Alert firing
- Logs show repeated errors

Immediate Action (SAFE ROLLBACK)

```
UPDATE feature_flags
SET enabled = 0
WHERE flag_key = 'enable_v2_api';
```

✔ Traffic falls back to v1
✔ No restart required

🔴 Latency Spike (P95 > 500ms)

Check

- `/metrics` → `http_request_duration_seconds`
- DB lock messages

Quick Mitigation
Disable audit logging:

```
UPDATE feature_flags
SET enabled = 0
WHERE flag_key = 'enable_audit_logging';
```

🔴 Database Locked

Symptoms

- "database is locked"
- Write failures

Actions

1. Restart container
2. Ensure no long-running transactions
3. Check file permissions

4. If corrupted → restore backup

Backup restore:

```
cp db/timetravel_backup.db db/timetravel.db
```

🧪 3. Validate Core Functionality

Create Record

```
curl -X POST http://localhost:8000/api/v2/records/101 \
-H "Content-Type: application/json" \
-H "X-User-ID: 10" \
-d '{"status":"active"}'
```

Get Record

```
curl http://localhost:8000/api/v2/records/101 \
-H "X-User-ID: 10"
```

Check Versions

```
curl http://localhost:8000/api/v2/records/101/versions \
-H "X-User-ID: 10"
```

📊 4. Critical Metrics to Watch

Metric	Healthy
http_requests_total	Increasing steadily
http_request_duration_seconds	P95 < 500ms
feature_flag_evaluations_total	Matches traffic

5xx rate	<1%

🗂️ 5. Database Quick Queries

Check records:

```
SELECT record_id, version, updated_at
FROM policyholder_records;
```

Check audit:

```
SELECT record_id, version, event_type
FROM audit_history
ORDER BY changed_at DESC;
```

Check flags:

```
SELECT flag_key, enabled
FROM feature_flags;
```

🧯 6. Full Emergency Rollback

If needed:

```
git revert <commit>
docker rebuild
docker restart
```

⚠ Use only if feature flag rollback insufficient.

🎯 7. Severity Guide

Severity	Action
Sev-1 (Outage)	Disable v2 flag immediately
Sev-2 (High latency)	Disable audit logging
Sev-3 (Minor issue)	Log & patch next release

🧠 8. Golden Rules

- Prefer **feature flag rollback first**
- Never delete DB in production
- Always verify `/health` after mitigation
- Confirm metrics stabilize before closing incident
- Document timeline in incident report

🏁 Fast Recovery Flow

```
Alert →
Check /health →
Check logs →
Check metrics →
Disable feature flag →
Verify recovery →
Document
```

TestPlan

🧪 Timetravel v2 – Test Plan

Version: v2
Scope: API, Persistence, Versioning, Audit, Feature Flags, Observability
Goal: Ensure correctness, stability, performance, and safe rollout.

1️⃣ Objectives

Validate that:

- v2 APIs function correctly
- Record versioning works
- Audit history is consistent
- Feature flags control behavior properly
- Observability metrics are accurate
- Rollback mechanisms are safe
- Docker deployment works
- Backward compatibility (v1) is maintained

2️⃣ Test Environments

Environment	Purpose
Local	Unit + integration
Docker	Deployment validation
Staging	Canary rollout
Production	Gradual rollout

3️⃣ Test Categories

✅ A. Unit Tests

Scope

- Service layer
- Controller validation
- Metrics logic
- Feature flag logic

Command

```
go test ./... -cover
```

Acceptance Criteria

- Coverage ≥ 85%
- No test failures
- No race conditions (`-race`)

✅ B. API Functional Tests

1. Health Endpoint

```
curl http://localhost:8000/api/v2/health
```

Expected:

- 200 OK
- JSON response valid

---

2. Create Record (v2)

```
curl -X POST http://localhost:8000/api/v2/records/101 \
-H "Content-Type: application/json" \
-H "X-User-ID: 10" \
-d '{"status":"active"}'
```

Expected:

- 200 OK
- Version = 1
- created_at populated

---

3. Update Record (Version Increment)

```
curl -X POST http://localhost:8000/api/v2/records/101 \
-H "Content-Type: application/json" \
-H "X-User-ID: 10" \
-d '{"status":"inactive"}'
```

Expected:

- Version = 2
- updated_at changed
- audit entry created

---

4. Fetch Versions

```
curl http://localhost:8000/api/v2/records/101/versions
```

Expected:

- Versions list includes 1 and 2

---

5. Fetch Specific Version

```
curl http://localhost:8000/api/v2/records/101/versions/1
```

Expected:

- Returns original record data

---

6. Header Validation

Missing header:

```
GET /records/101
```

Expected:

- 400 error
- Log: missing X-User-ID header

✅ C. Database Validation Tests

Check version consistency

```
SELECT version FROM policyholder_records WHERE record_id=101;
```

Verify audit logging

```
SELECT record_id, version, event_type
FROM audit_history
WHERE record_id=101;
```

Expected:

- One audit row per update
- Version numbers sequential

✅ D. Feature Flag Tests

Disable v2 API

```
UPDATE feature_flags
SET enabled=0
WHERE flag_key='enable_v2_api';
```

Expected:

- v2 endpoints unavailable
- v1 still functional

Disable Audit Logging

```
UPDATE feature_flags
SET enabled=0
WHERE flag_key='enable_audit_logging';
```

Expected:

- Updates work
- No new audit_history rows

Disable Metrics

```
UPDATE feature_flags
SET enabled=0
WHERE flag_key='enable_metrics';
```

Expected:

- No DB metric inserts
- In-memory metrics may still increment

✅ E. Observability Tests

Check metrics endpoint

```
curl http://localhost:8000/metrics
```

Verify presence of:

- http_requests_total
- http_request_duration_seconds
- feature_flag_evaluations_total

Latency Validation

Run:

```
ab -n 100 -c 10 http://localhost:8000/api/v2/records/101
```

Expected:

- P95 < 500ms
- No errors

✅ F. Concurrency Tests

Simulate parallel updates:

```
for i in {1..20}; do
  curl -X POST http://localhost:8000/api/v2/records/101 \
  -H "Content-Type: application/json" \
  -H "X-User-ID: 10" \
  -d "{\"status\":\"test$i\"}" &
done
```

Expected:

- No DB lock crash
- Versions increment sequentially
- No duplicate version numbers

✅ G. Rollback Tests

1. Enable v2
2. Create records
3. Disable flag
4. Verify:
    - v2 inaccessible
    - Data remains intact
    - No schema corruption

✅ H. Docker Tests

```
docker build -t timetravel .
docker run -p 8000:8000 timetravel
```

Validate:

- App starts
- DB created
- Metrics exposed

✅ I. Migration Tests

1. Start with old schema
2. Apply migration
3. Validate:
    - Data preserved
    - Version default = 1
    - Foreign keys enabled

Check:

```
PRAGMA foreign_keys;
```

Expected:

```
1
```

4️⃣ Performance Testing

Endpoint	Expected
GET record	<200ms
POST record	<500ms

Metrics	<50ms

Load test:

```
wrk -t4 -c50 -d30s http://localhost:8000/api/v2/records/101
```

Pass Criteria:

- Error rate <1%
- Stable latency

5️⃣ Chaos Testing (Optional)

Simulate:

- Kill container during write
- Disable audit flag mid-traffic
- Corrupt DB copy (non-prod)

Verify:

- Recovery possible
- No unrecoverable corruption

6️⃣ Security Tests

- Header validation
- SQL injection attempts
- Invalid JSON payload
- Oversized payload

Example:

```
curl -X POST http://localhost:8000/api/v2/records/101 \
-d "' OR 1=1 --"
```

Expected:

- 400 error
- No DB corruption

7️⃣ Exit Criteria

Release is approved when:

✔ All unit tests pass
✔ Coverage ≥ 85%
✔ Functional tests pass
✔ No DB lock issues
✔ Feature flag toggles validated
✔ Metrics visible & correct
✔ Rollback tested
✔ Docker validated

8️⃣ Test Artifacts

- Test logs
- Coverage report
- Load test results
- DB validation screenshots
- Metrics snapshot

🎯 Final Success Definition

System is considered production-ready when:

- Stable under load
- Fully observable
- Safely rollback-able
- Data consistent
- Versioning accurate

- Audit complete