Guile Deploy Ledger - Technical Deep Dive

Architecture, Implementation, and Best Practices

Engineering Team

January 2024

Architecture Overview

System Layers

```
+-----------------------------------------------+
|                 CLI Interface                 |
|            (deploy-ledger cli main)           |
+-----------------------------------------------+
|             Query & Metrics Layer             |
|         (deploy-ledger query metrics)         |
+-----------------------------------------------+
|                Core Data Types                |
|           (deploy-ledger core types)          |
+-----------------------------------------------+
|               Storage Abstraction             |
|        (deploy-ledger storage sqlite)         |
+-----------------------------------------------+
```

Data Model Design

SRFI-9 Records Architecture

```
(define-record-type <deployment-event>
  (make-deployment-event-internal
    id service-name version environment
    deployment-type started completed status
    initiator metadata parent-id)
  deployment-event?
  (id deployment-event-id)
  (service-name deployment-event-service-name)
  ;; ... additional fields
  )
```

Benefits:

Storage Layer

SQLite Backend Choice

- Zero configuration
- ACID compliance
- Excellent for embedded use
- Single file deployment
- WAL mode for concurrency

- Limited write concurrency
- Not suitable for >1GB/day
- No built-in replication
- Single-node limitation

Transaction Management

```
(define (with-transaction db thunk)
  "Execute thunk within database transaction"
  (sqlite-exec db "BEGIN TRANSACTION")
  (catch #t
    (lambda ()
```

Query Engine

Metrics Calculation Pipeline

```
(define* (calculate-mttr db service-name
          #:key (period-days 30))
  (let* ((failures (get-failures db service-name))
         (recoveries (map find-recovery failures)))
    (average (map time-difference
                  failures recoveries))))
```

Pipeline stages:

1. Data retrieval with filters
2. Time-based aggregation
3. Statistical calculation
4. Result formatting

Performance Optimization

Caching Strategy

```
(define *metrics-cache*
  (make-hash-table))

(define (cached-metric key thunk)
  (or (hash-ref *metrics-cache* key)
      (let ((value (thunk)))
        (hash-set! *metrics-cache*
                   key value)
        value)))
```

- Time-based expiry
- Event-based invalidation
- LRU eviction
- Manual flush

Query Optimization

Techniques employed:

1. Index usage: All queries use covering indexes

Integration Patterns

Webhook Processing

```
(define (webhook-handler request)
  (let* ((payload (parse-json-body request))
         (deployment (webhook->deployment payload)))
    (with-transaction db
      (lambda ()
        (store-deployment! db deployment)
        (trigger-notifications deployment)
        (update-metrics-cache deployment)))
    (respond-200 "OK")))
```

Features:

- Idempotent processing

Testing Strategy

Test Pyramid

```
    /\        UI Tests
   /  \       (5%)
  /    \
 /      \     Integration Tests
/        \    (20%)
/          \
/            \ Unit Tests
+-----------+ (75%)
```

Unit Testing with SRFI-64

```
(test-group "deployment-event"
  (test-assert "create deployment"
```

Deployment Strategies

Zero-Downtime Deployment

1. Deploy to blue
2. Run health checks
3. Switch traffic
4. Keep green as backup
5. Cleanup after validation

1. Deploy to 5% traffic
2. Monitor metrics
3. Gradual increase
4. Full rollout at 100%
5. Automatic rollback on errors

Rollback Strategies

```
(define (intelligent-rollback service)
  (let* ((current (get-current-version service))
         (previous (get-last-stable-version service))
         (impact (analyze-rollback-impact
                   service current previous)))
```

Monitoring & Observability

Metrics Export

```
# HELP deployments_total Total deployments
# TYPE deployments_total counter
deployments_total{service="api"} 42

# HELP deployment_duration_seconds
# TYPE deployment_duration_seconds histogram
deployment_duration_seconds_bucket{le="60"} 35
```

```
(define (create-span deployment)
  (make-span
    #:name "deployment"
    #:attributes
    `((service . ,service)
      (version . ,version))))
```

Alerting Rules

```
groups:
- name: deployment_alerts
  rules:
  - alert: HighFailureRate
```

Security Considerations

Authentication & Authorization

```
(define (validate-token token)
  (let ((claims (jwt-decode token)))
    (and (valid-signature? token)
         (not-expired? claims)
         (authorized-scope? claims))))
```

- Viewer: Read-only access
- Deployer: Record deployments
- Admin: Full access
- Auditor: Export reports

Data Protection

Measures implemented:

1. Encryption at rest: SQLCipher support
2. TLS for transit: HTTPS only
3. PII handling: Automatic redaction
4. Audit logging: All modifications tracked

Future Enhancements

Planned Features

- Kubernetes operator
- GraphQL API
- Real-time streaming
- ML predictions
- Multi-cluster federation
- GitOps integration
- Cost tracking
- SLO management
- Auto-remediation
- Chaos engineering
- Compliance automation
- AI-driven insights

Extension Points

```
;; Custom storage backend
(define-storage-backend postgresql
  #:connect pg-connect
  #:store pg-store
  #:retrieve pg-retrieve)

;; Custom metrics
```

Q&A

Questions?

./guile-logo.png