Pulse Architecture Documentation

System Overview

Pulse is a distributed uptime monitoring service built with modern web technologies. The system consists of three main components: frontend, backend, and infrastructure.

Architecture Diagram

Component Architecture

Frontend (Client)

Technology Stack: - React 19 with TypeScript for type-safe UI - Next.js for server-side rendering - Tailwind CSS 4 for styling - shadcn/ui for component library - tRPC for end-to-end type safety - Wouter for lightweight routing

Key Pages: 1. **Home Page**: Landing page with feature overview 2. **Dashboard**: Real-time monitoring overview 3. **Targets**: CRUD operations for monitoring targets 4. **Target Detail**: Detailed view with checks and alerts 5. **Alerts**: Alert management and history 6. **Settings**: User preferences and notifications

State Management: - React Query for server state - tRPC hooks for data fetching - Context API for authentication

Backend (Server)

Technology Stack: - Express.js 4 for HTTP server - tRPC 11 for type-safe RPC - Node.js 22 runtime - Drizzle ORM for database operations

Core Modules:

1. **Authentication Module** (`server/_core/`)
2. OAuth2 integration with Manus
3. Session management with JWT
4. Role-based access control
5. **Monitoring Module** (`server/monitoring.ts`)
6. Periodic health check execution
7. HTTP request handling
8. Response time measurement
9. Status code validation
10. Alert triggering logic
11. **Notification Module** (`server/notifications.ts`)

12. Email notifications
13. Slack webhook integration
14. Discord webhook integration
15. Notification queuing
16. **Database Module** (`server/db.ts`)
17. Query helpers for all entities
18. Transaction management
19. Connection pooling
20. **Router Module** (`server/routers.ts`)
21. tRPC procedure definitions
22. Input validation with Zod
23. Authorization checks
24. Business logic implementation

Database (Data)

Schema Design:

```
users
├── id (PK)
├── openId (UNIQUE)
├── name
├── email
├── role (admin | user)
└── timestamps

monitoring_targets
├── id (PK)
├── userId (FK)
├── name
├── url
├── protocol (http | https)
├── method (GET | POST | HEAD)
├── checkInterval
├── timeout
├── expectedStatusCode
├── isActive
└── timestamps

monitoring_checks
├── id (PK)
├── targetId (FK)
├── statusCode
├── responseTime
├── isSuccess
├── errorMessage
└── checkedAt

alert_rules
├── id (PK)
├── targetId (FK)
├── userId (FK)
├── name
├── ruleType
├── threshold
├── notificationChannels (JSON)
├── isActive
└── timestamps

alerts
├── id (PK)
├── ruleId (FK)
├── targetId (FK)
├── userId (FK)
├── status (triggered | acknowledged | resolved)
├── message
├── severity (low | medium | high | critical)
└── timestamps

notification_settings
├── id (PK)
├── userId (FK, UNIQUE)
├── emailEnabled
├── slackWebhookUrl
├── discordWebhookUrl
└── timestamps

uptime_statistics
├── id (PK)
├── targetId (FK)
├── period (daily | weekly | monthly)
├── date
├── totalChecks
```

```
├── successfulChecks
├── uptimePercentage
├── averageResponseTime
└── timestamps

audit_logs
├── id (PK)
├── userId (FK)
├── action
├── entityType
├── entityId
├── details
└── createdAt
```

Data Flow

Monitoring Flow

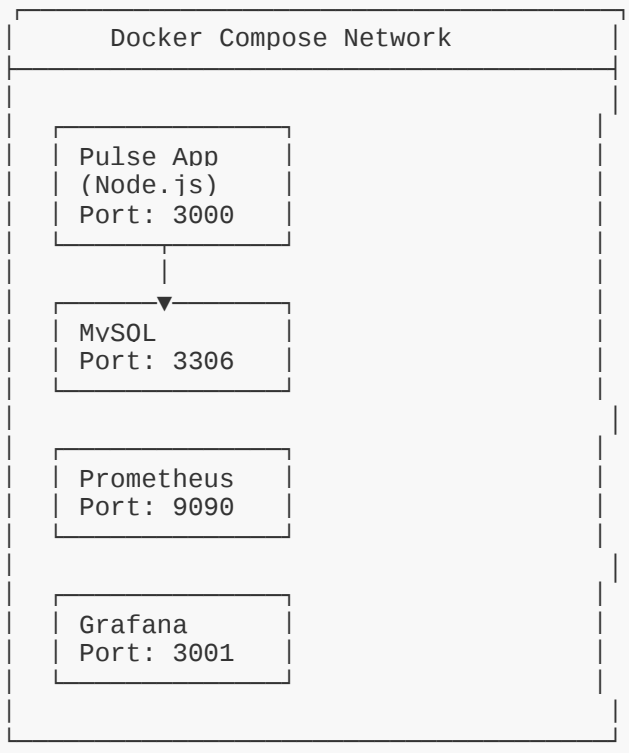
```
1. Scheduler triggers monitoring cycle
   ↓
2. Fetch all active targets from database
   ↓
3. For each target:
   a. Perform HTTP health check
   b. Measure response time
   c. Validate status code
   d. Save check result to database
   ↓
4. Evaluate alert rules
   ↓
5. If threshold exceeded:
   a. Create alert record
   b. Send notifications (email, Slack, Discord)
   ↓
6. Calculate uptime statistics
```

User Request Flow

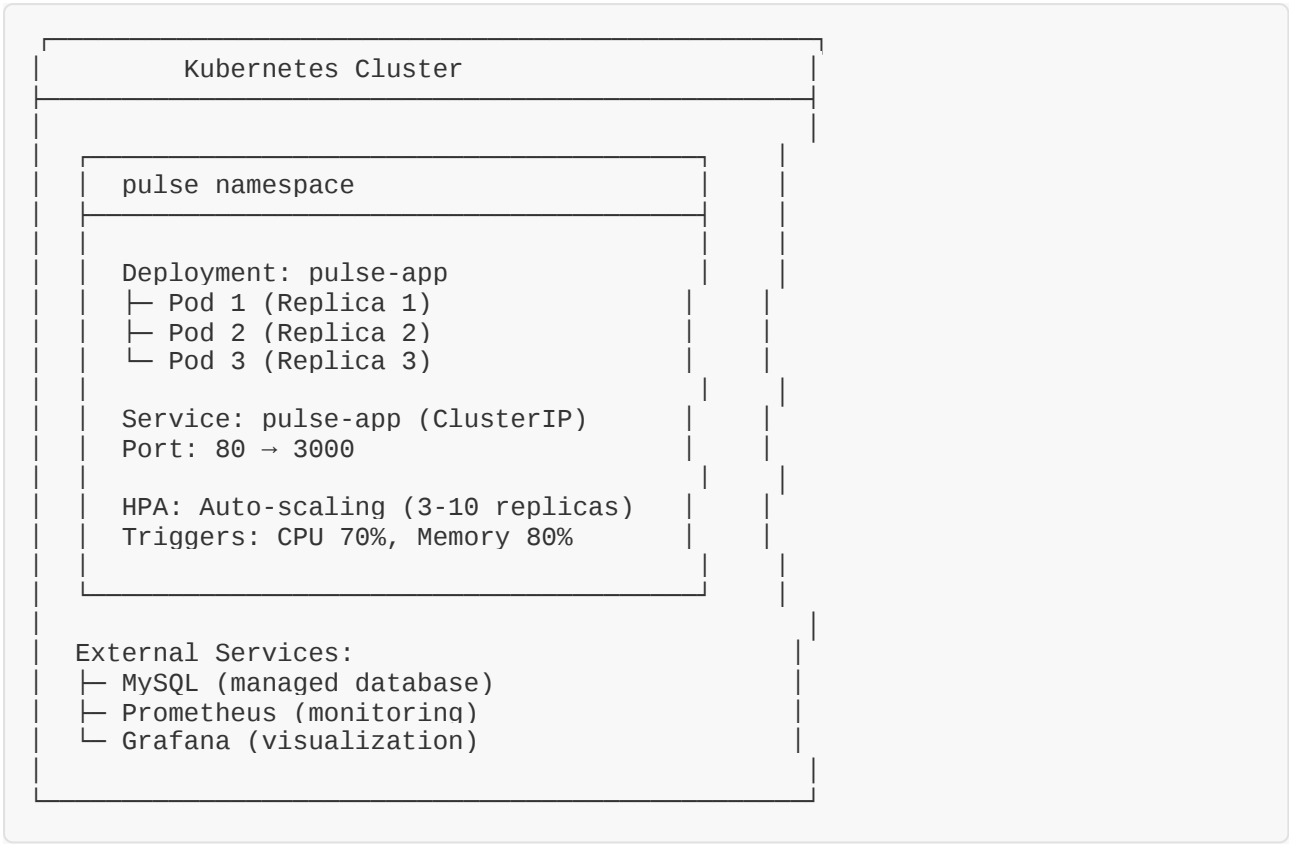
```
1. User action in React component
   ↓
2. Call tRPC hook (useQuery/useMutation)
   ↓
3. tRPC client serializes request
   ↓
4. Express server receives request
   ↓
5. Authentication middleware validates session
   ↓
6. Authorization check (protectedProcedure)
   ↓
7. Input validation with Zod
   ↓
8. Business logic execution
   ↓
9. Database operation via Drizzle ORM
   ↓
10. Response serialization with SuperJSON
    ↓
11. React component updates with new data
```

Deployment Architecture

Docker Deployment



Kubernetes Deployment



Security Architecture

Authentication Flow

```
1. User clicks "Sign In"
   ↓
2. Redirected to Manus OAuth portal
   ↓
3. User authenticates with credentials
   ↓
4. OAuth server redirects to /api/oauth/callback
   ↓
5. Backend exchanges code for token
   ↓
6. User record created/updated in database
   ↓
7. Session cookie set with JWT
   ↓
8. Redirected to dashboard
```

Authorization Model

```
Public Routes:
├─ / (home page)
└─ /404 (error page)

Protected Routes (require authentication):
├─ /dashboard
├─ /targets
├─ /targets/:id
├─ /alerts
└─ /settings

Admin-Only Operations:
├─ User management
├─ System configuration
└─ Audit log access
```

Monitoring & Observability

Metrics Collection

Prometheus scrapes metrics from: - Application metrics endpoint (`/metrics`) - MySQL database metrics - System resource metrics

Key Metrics

1. **Application Metrics**
2. Request count and latency
3. Error rates
4. tRPC procedure execution time
5. Database query duration
6. **Monitoring Metrics**
7. Health check success/failure rates
8. Average response times
9. Alert trigger frequency
10. Target availability
11. **Infrastructure Metrics**

12. CPU usage
13. Memory usage
14. Disk usage
15. Network I/O

Alerting Rules

Prometheus alert rules trigger when: - Application error rate > 5% - Response time > 1000ms - Database connection pool exhausted - Pod restart frequency > threshold

Scalability Considerations

Horizontal Scaling

1. **Stateless Application**
2. No server-side sessions (JWT-based)
3. All state in database
4. Can run multiple instances
5. **Load Balancing**
6. Kubernetes Service distributes traffic
7. Round-robin load balancing
8. Health checks ensure pod readiness
9. **Database Scaling**
10. Read replicas for analytics queries
11. Connection pooling
12. Query optimization

Vertical Scaling

- Increase pod resource limits

- Upgrade database instance type
- Optimize code and queries

Performance Optimization

Frontend Optimization

- Code splitting with dynamic imports
- Image optimization
- CSS minification
- JavaScript bundling with Vite

Backend Optimization

- Database connection pooling
- Query result caching
- Batch operations
- Async/await for non-blocking I/O

Infrastructure Optimization

- CDN for static assets
- Database indexing
- Query optimization
- Caching strategies

Disaster Recovery

Backup Strategy

- Daily database backups
- Off-site backup storage

- Point-in-time recovery capability

High Availability

- Multi-replica deployment
- Database replication
- Automated failover
- Health checks and pod restart

Disaster Recovery Plan

1. Identify failure
2. Failover to backup systems
3. Restore from latest backup
4. Verify data integrity
5. Resume normal operations

Future Enhancements

1. **Advanced Analytics**
2. Machine learning for anomaly detection
3. Predictive alerting
4. Trend analysis
5. **Enhanced Monitoring**
6. Custom metrics collection
7. Synthetic monitoring
8. Real user monitoring (RUM)
9. **Integrations**
10. PagerDuty integration
11. Opsgenie integration

12. Custom webhook support
13. **Performance**
14. GraphQL API
15. WebSocket support for real-time updates
16. Edge computing for distributed monitoring

Version: 1.0.0
Last Updated: 2024