AI Application Specification: Architecture Review & Complete Sample

Architecture Decisions Through the AI Lens

Design Decision Review

✅ Brilliant Decisions for AI Application Generation

1. Database-Driven Deployment Coordination

- **Why Brilliant:** AI agents can update deployment tables programmatically
- **AI Benefit:** No complex Kubernetes API calls, just SQL-like operations
- **Enables:** Agents can deploy/update/rollback applications declaratively

2. Separated Base + Application Container Layers

- **Why Brilliant:** AI agents only generate small application layers (5-15MB)
- **AI Benefit:** Fast iteration, cached infrastructure, minimal bandwidth
- **Enables:** Agents can rapidly prototype and deploy variations

3. MQTT Message Bus Architecture

- **Why Brilliant:** Universal communication protocol across all platforms
- **AI Benefit:** Same messaging model for web, mobile, AR, IoT
- **Enables:** AI agents generate one communication pattern, works everywhere

4. Service Functions with Schema-Driven Code Generation

- **Why Brilliant:** AI agents can generate JSON schemas → automatic integration code
- **AI Benefit:** Type safety without manual coding, universal interfaces
- **Enables:** Agents create robust applications with proper validation

5. Factory Pattern for Platform Interfaces

- **Why Brilliant:** Consistent API surface across all languages and platforms
- **AI Benefit:** Agents learn one pattern, generate for any runtime
- **Enables:** True write-once, deploy-anywhere applications

6. Declarative UI Templates

- **Why Brilliant:** AI agents can generate high-level UI descriptions
- **AI Benefit:** No platform-specific UI code, automatic responsive design
- **Enables:** Same UI specification↦ web, mobile, AR interfaces

🔄 Decisions That Need AI-First Enhancement

1. Container Lifecycle Management

- **Current:** Sophisticated but developer-focused
- **AI Need:** Simple, declarative scaling policies
- **Enhancement:** YAML-based resource specifications

2. Error Handling Strategy

- **Current:** Comprehensive but manual configuration
- **AI Need:** Automatic error recovery patterns
- **Enhancement:** Self-healing application specifications

3. Security Configuration

- **Current:** RBAC at library level (excellent)
- **AI Need:** Declarative security policies
- **Enhancement:** Security-as-code in app specifications

Complete AI Application Sample

Scenario: Smart Factory Monitoring System

An AI agent generates a complete application from this prompt:

> "Build a smart factory monitoring system with real-time sensor data, web dashboard for managers, mobile app for technicians, AR overlay for equipment maintenance, and predictive maintenance alerts."

Generated Application Structure

```
smart-factory-monitor/
├── app-spec.yml              # Universal deployment specification
├── services/                 # Business logic functions
│   ├── sensor-processor.js
│   ├── alert-manager.py
│   ├── maintenance-scheduler.cs
│   └── prediction-engine.py
├── schemas/                  # Data contracts
│   ├── sensor-reading.json
│   ├── maintenance-request.json
│   ├── alert-notification.json
│   └── equipment-status.json
├── ui/                       # Cross-platform interfaces
│   ├── manager-dashboard.template
│   ├── technician-mobile.template
│   ├── ar-maintenance-overlay.template
│   └── iot-device-interface.template
├── data/                     # Database schemas
│   ├── equipment.schema.json
│   ├── sensors.schema.json
│   └── maintenance-logs.schema.json
└── deployment/               # Environment configurations
    ├── dev-local.yml
    ├── staging-cloud.yml
    └── production-global.yml
```

---

1. Universal Application Specification

app-spec.yml

yaml

```yaml
# Universal AI Application Specification
application:
 name: "SmartFactoryMonitor"
 version: "1.0.0"
 description: "AI-generated smart factory monitoring system"

# Service Functions (Business Logic)
services:
 sensor-processor:
  runtime: "nodejs"
  file: "./services/sensor-processor.js"
  schema: "./schemas/sensor-reading.json"
  scaling:
   min-instances: 2
   max-instances: 100
   trigger: "message-queue-depth > 1000"

 alert-manager:
  runtime: "python"
  file: "./services/alert-manager.py"
  schema: "./schemas/alert-notification.json"
  scaling:
   min-instances: 1
   max-instances: 10
   trigger: "cpu > 70%"

 maintenance-scheduler:
  runtime: "dotnet"
  file: "./services/maintenance-scheduler.cs"
  schema: "./schemas/maintenance-request.json"
  permissions:
   - "maintenance:create"
   - "equipment:update"

 prediction-engine:
  runtime: "python"
  file: "./services/prediction-engine.py"
  resources:
   gpu: true
   memory: "8GB"

# Multi-Platform User Interfaces
interfaces:
```

```
  manager-dashboard:
    type: "web"
    template: "./ui/manager-dashboard.template"
    auth-required: true
    roles: ["manager", "admin"]
    real-time: true

  technician-mobile:
    type: "mobile"
    template: "./ui/technician-mobile.template"
    platforms: ["ios", "android"]
    offline-capable: true
    push-notifications: true

  ar-maintenance:
    type: "ar"
    template: "./ui/ar-maintenance-overlay.template"
    platforms: ["hololens", "magic-leap", "webxr"]
    spatial-tracking: true
    hand-tracking: true

  iot-interface:
    type: "iot"
    template: "./ui/iot-device-interface.template"
    protocols: ["mqtt", "coap"]
    security: "certificate-based"

# Data Models
data:
  databases:
    primary:
      type: "postgresql"
      collections: ["equipment", "sensors", "maintenance_logs"]

    timeseries:
      type: "mongodb"
      collections: ["sensor_readings", "performance_metrics"]

    cache:
      type: "redis"
      ttl: "1h"

# Messaging and Events
messaging:
  topics:
    "sensors/+/data"       # All sensor data
```

```
    - "sensors/+/data"        # All sensor data
    - "alerts/critical"       # Critical alerts
    - "maintenance/requests"  # Maintenance requests
    - "ui/updates/+/+"        # UI real-time updates

  qos: 1
  persistence: true
  replay-protection: true

# Security and Access Control
security:
  authentication: "oidc"
  provider: "azure-ad"

  roles:
    admin:
      permissions: ["*:*:*"]
    manager:
      permissions: ["equipment:read:*", "reports:read:*", "alerts:read:*"]
    technician:
      permissions: ["equipment:read:assigned", "maintenance:*:assigned"]
    iot-device:
      permissions: ["sensors:write:own", "status:read:own"]

# Deployment Configuration
deployment:
  targets:
    development:
      infrastructure: "local"
      scaling: "minimal"

    staging:
      infrastructure: "cloud"
      provider: "azure"
      regions: ["us-east"]

    production:
      infrastructure: "multi-cloud"
      providers: ["aws", "azure"]
      regions: ["us-east", "eu-west", "asia-pacific"]
      disaster-recovery: true

    edge:
      infrastructure: "edge"
      locations: ["factory-floor", "mobile-units"]
      offline-capable: true
```

```
offline-capable: true
```

---

2. Service Functions (Business Logic)

services/sensor-processor.js

javascript

```javascript
// AI-generated service function for processing sensor data
import { IDataStore, IMessaging, IAuthorizer } from '@cloud-rails/platform';

export class SensorProcessor {
 constructor(dataStore, messaging, authorizer) {
  this.dataStore = dataStore;
  this.messaging = messaging;
  this.authorizer = authorizer;
 }

 // Process incoming sensor readings
 async processSensorData(sensorReading) {
  // Validate sensor data against schema
  if (!this.validateReading(sensorReading)) {
   throw new ValidationError("Invalid sensor reading format");
  }

  // Store in time-series database
  const readingId = await this.dataStore.insert('sensor_readings', {
   sensorId: sensorReading.sensorId,
   timestamp: sensorReading.timestamp,
   temperature: sensorReading.temperature,
   vibration: sensorReading.vibration,
   pressure: sensorReading.pressure,
   equipmentId: sensorReading.equipmentId
  });

  // Check for anomalies
  const anomaly = await this.detectAnomaly(sensorReading);
  if (anomaly.severity > 0.7) {
   // Trigger alert
   await this.messaging.send('alerts/critical', {
    type: 'sensor_anomaly',
    sensorId: sensorReading.sensorId,
    severity: anomaly.severity,
    description: anomaly.description,
    timestamp: new Date().toISOString()
   });
  }

  // Update real-time dashboards
  await this.messaging.send(`ui/updates/sensors/${sensorReading.sensorId}`, {
   latest: sensorReading,
```

```
    anomaly: anomaly
  });

  // Update equipment status
  await this.updateEquipmentStatus(sensorReading.equipmentId, sensorReading);

  return { readingId, anomaly };
 }

 async detectAnomaly(reading) {
  // AI model inference for anomaly detection
  const historicalData = await this.dataStore.query('sensor_readings', {
    sensorId: reading.sensorId,
    timestamp: { $gte: new Date(Date.now() - 24*60*60*1000) }
  });

  // Simple anomaly detection (AI agent would generate ML model call)
  const avgTemp = historicalData.reduce((a, b) => a + b.temperature, 0) / historicalData.length;
  const tempDeviation = Math.abs(reading.temperature - avgTemp) / avgTemp;

  return {
    severity: Math.min(tempDeviation * 2, 1.0),
    description: tempDeviation > 0.2 ?
      `Temperature anomaly: ${reading.temperature}°C vs avg ${avgTemp.toFixed(1)}°C` : null
  };
 }

 async updateEquipmentStatus(equipmentId, reading) {
  const equipment = await this.dataStore.getById('equipment', equipmentId);

  // Update equipment with latest sensor data
  await this.dataStore.update('equipment', equipmentId, {
    lastSensorUpdate: reading.timestamp,
    currentTemperature: reading.temperature,
    currentVibration: reading.vibration,
    currentPressure: reading.pressure,
    status: reading.temperature > 80 ? 'warning' : 'normal'
  });

  // Notify all interfaces of equipment status change
  await this.messaging.send(`ui/updates/equipment/${equipmentId}`, {
    equipment: { ...equipment, currentTemperature: reading.temperature }
  });
 }

 validateReading(reading) {
```

```
  validateReading(reading) {
    return reading.sensorId &&
           reading.timestamp &&
           typeof reading.temperature === 'number' &&
           typeof reading.vibration === 'number' &&
           typeof reading.pressure === 'number';
  }
}
```

services/alert-manager.py

python

```python
# AI-generated alert management service
from cloud_rails.platform import IDataStore, IMessaging, IAuthorizer, UserContext
import asyncio
from datetime import datetime, timedelta

class AlertManager:
  def __init__(self, data_store: IDataStore, messaging: IMessaging, authorizer: IAuthorizer):
    self.data_store = data_store
    self.messaging = messaging
    self.authorizer = authorizer

  async def process_alert(self, alert_data: dict, user_context: UserContext):
    """Process incoming alerts and route to appropriate channels"""

    # Store alert in database
    alert_id = await self.data_store.insert('alerts', {
      'type': alert_data['type'],
      'severity': alert_data['severity'],
      'sensor_id': alert_data.get('sensorId'),
      'equipment_id': await self.get_equipment_for_sensor(alert_data.get('sensorId')),
      'description': alert_data['description'],
      'timestamp': alert_data['timestamp'],
      'status': 'open',
      'created_by': 'system'
    })

    # Determine notification recipients based on severity and equipment assignment
    recipients = await self.get_alert_recipients(alert_data)

    # Send notifications to all appropriate channels
    for user_id in recipients:
      await self.send_notification(user_id, alert_data, alert_id)

    return {'alertId': alert_id, 'notifiedUsers': recipients}

  async def get_alert_recipients(self, alert_data: dict):
    """Determine who should be notified based on alert type and severity"""
    recipients = []

    # Critical alerts go to all managers and on-call technicians
    if alert_data['severity'] == 'critical':
      managers = await self.data_store.query('users', {'role': 'manager'})
      on_call = await self.data_store.query('users', {'on_call': True})
```

```
        recipients.extend([user['id'] for user in managers + on_call])

    # Equipment-specific alerts go to assigned technicians
    if alert_data.get('sensorId'):
        equipment_id = await self.get_equipment_for_sensor(alert_data['sensorId'])
        if equipment_id:
            assigned_techs = await self.data_store.query('user_equipment_assignments',
                                    {'equipment_id': equipment_id})
            recipients.extend([assignment['user_id'] for assignment in assigned_techs])

    return list(set(recipients)) # Remove duplicates

async def send_notification(self, user_id: str, alert_data: dict, alert_id: str):
    """Send notification to user across all their platforms"""

    # Get user preferences
    user = await self.data_store.get_by_id('users', user_id)

    # Send to mobile app (push notification)
    if user.get('mobile_notifications', True):
        await self.messaging.send(f'notifications/mobile/{user_id}', {
            'type': 'alert',
            'title': f"Alert: {alert_data['type']}",
            'body': alert_data['description'],
            'data': {'alertId': alert_id}
        })

    # Send to web dashboard (real-time update)
    await self.messaging.send(f'ui/updates/user/{user_id}/alerts', {
        'alertId': alert_id,
        'alert': alert_data
    })

    # Send to AR interface if user is in AR session
    ar_sessions = await self.data_store.query('ar_sessions', {
        'user_id': user_id,
        'status': 'active'
    })

    for session in ar_sessions:
        await self.messaging.send(f'ar/session/{session["id"]}/alerts', {
            'alertId': alert_id,
            'alert': alert_data,
            'spatial_location': session.get('current_equipment_location')
        })
```

```
async def get_equipment_for_sensor(self, sensor_id: str):
    """Get equipment ID associated with sensor"""
    if not sensor_id:
        return None

    sensor = await self.data_store.get_by_id('sensors', sensor_id)
    return sensor.get('equipment_id') if sensor else None
```

---

3. Data Schemas (AI-Generated Contracts)

schemas/sensor-reading.json

json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "SensorReading",
  "description": "Real-time sensor data from factory equipment",
  "type": "object",
  "required": ["sensorId", "equipmentId", "timestamp", "temperature", "vibration", "pressure"],
  "properties": {
    "sensorId": {
      "type": "string",
      "pattern": "^SENSOR_[A-Z0-9]{8}$",
      "description": "Unique sensor identifier"
    },
    "equipmentId": {
      "type": "string",
      "pattern": "^EQ_[A-Z0-9]{6}$",
      "description": "Associated equipment identifier"
    },
    "timestamp": {
      "type": "string",
      "format": "date-time",
      "description": "ISO 8601 timestamp of reading"
    },
    "temperature": {
      "type": "number",
      "minimum": -40,
      "maximum": 200,
      "description": "Temperature in Celsius"
    },
    "vibration": {
      "type": "number",
      "minimum": 0,
      "maximum": 100,
      "description": "Vibration level (0-100 scale)"
    },
    "pressure": {
      "type": "number",
      "minimum": 0,
      "maximum": 1000,
      "description": "Pressure in PSI"
    },
    "location": {
      "type": "object",
      "properties": {
```

```
        "x": {"type": "number"},
        "y": {"type": "number"},
        "z": {"type": "number"}
      },
      "description": "3D coordinates for AR overlay"
    },
    "metadata": {
      "type": "object",
      "additionalProperties": true,
      "description": "Additional sensor-specific data"
    }
  }
}
```

---

4. Cross-Platform UI Templates

ui/manager-dashboard.template

javascript

```
// AI-generated declarative UI for manager web dashboard
dashboard("Smart Factory Monitor") {
 header("Factory Overview") {
  userMenu(user: svc::Auth.currentUser)
  notifications(source: svc::Alerts.unreadCount)
 }

 sidebar {
  navigation {
   item("Overview", onclick: svc::Navigation.showOverview())
   item("Equipment", onclick: svc::Navigation.showEquipment())
   item("Alerts", onclick: svc::Navigation.showAlerts())
   item("Reports", onclick: svc::Navigation.showReports())
  }

  statusPanel {
   metric("Equipment Online", value: svc::Equipment.onlineCount)
   metric("Active Alerts", value: svc::Alerts.activeCount, color: "red")
   metric("Today's Production", value: svc::Production.todayMetrics)
  }
 }

 main {
  if (svc::Navigation.currentView === "overview") {
   row {
    column(width: 8) {
     card("Equipment Status") {
      equipmentGrid {
       for (equipment in svc::Equipment.all) {
        equipmentCard(
         name: equipment.name,
         status: equipment.status,
         temperature: equipment.currentTemperature,
         lastUpdate: equipment.lastSensorUpdate,
         onclick: svc::Equipment.showDetails(equipment.id)
        ) {
         if (equipment.status === "warning") {
          badge("Warning", color: "orange")
         }
         if (equipment.hasActiveAlerts) {
          badge("Alert", color: "red")
         }
        }
```

```
      }
     }
    }
   }

   column(width: 4) {
    card("Real-time Alerts") {
     alertList {
      for (alert in svc::Alerts.recent) {
       alertItem(
        type: alert.type,
        severity: alert.severity,
        description: alert.description,
        timestamp: alert.timestamp,
        onclick: svc::Alerts.showDetails(alert.id)
       )
      }
     }
    }

    card("Performance Trends") {
     chart(
      type: "line",
      data: svc::Analytics.performanceTrends,
      realtime: true
     )
    }
   }
  }
 }

if (svc::Navigation.currentView === "equipment") {
 dataTable(
  data: svc::Equipment.all,
  columns: [
   { field: "name", sortable: true },
   { field: "type", filterable: true },
   { field: "status", render: status => statusBadge(status) },
   { field: "currentTemperature", format: "temperature" },
   { field: "lastMaintenance", format: "date" },
   {
    field: "actions",
    render: equipment => buttonGroup(
     button("Details", onclick: svc::Equipment.showDetails(equipment.id)),
     button("Maintain", onclick: svc::Maintenance.schedule(equipment.id))
    )
```

```
        )
       }
      ],
      pagination: true,
      realtime: true
    )
   }
  }

  // Real-time updates via messaging
  onMessage("ui/updates/equipment/+", (equipmentId, data) => {
   svc::Equipment.updateRealtime(equipmentId, data);
  });

  onMessage("ui/updates/alerts/new", (alertData) => {
   svc::Alerts.addRealtime(alertData);
   showNotification("New Alert", alertData.description);
  });
 }
```

ui/technician-mobile.template

javascript

```
// AI-generated mobile interface for technicians
mobileApp("Factory Tech") {
 tabNavigation {
  tab("Equipment", icon: "gear") {
   equipmentList {
    for (equipment in svc::Equipment.assignedTo(svc::Auth.currentUser.id)) {
     equipmentCard(
      name: equipment.name,
      status: equipment.status,
      lastMaintenance: equipment.lastMaintenance,
      onclick: svc::Equipment.showDetails(equipment.id)
     ) {
      if (equipment.needsMaintenance) {
       badge("Maintenance Due", color: "orange")
      }

      quickActions {
       button("QR Scan", onclick: svc::Scanner.scanEquipment())
       button("Report Issue", onclick: svc::Issues.report(equipment.id))
       button("AR Guide", onclick: svc::AR.openMaintenanceGuide(equipment.id))
      }
     }
    }
   }
  }

  tab("Tasks", icon: "list") {
   taskList {
    for (task in svc::Tasks.assignedTo(svc::Auth.currentUser.id)) {
     taskCard(
      title: task.title,
      equipment: task.equipment.name,
      priority: task.priority,
      dueDate: task.dueDate,
      progress: task.progress
     ) {
      actions {
       button("Start", onclick: svc::Tasks.start(task.id))
       button("AR Guide", onclick: svc::AR.openTaskGuide(task.id))
      }
     }
    }
   }
  }
```

```
    }

    tab("Profile", icon: "user") {
      userProfile {
        avatar(src: svc::Auth.currentUser.avatar)
        text(svc::Auth.currentUser.name)

        settings {
          toggle("Push Notifications", bind: svc::Settings.pushEnabled)
          toggle("AR Assistance", bind: svc::Settings.arEnabled)
          select("Language", options: svc::Settings.languages, bind: svc::Settings.language)
        }
      }
    }
  }

  // Push notification handling
  onPushNotification((notification) => {
    if (notification.type === "alert") {
      svc::Navigation.showAlert(notification.data.alertId);
    }
  });

  // Real-time updates
  onMessage("notifications/mobile/" + svc::Auth.currentUser.id, (notification) => {
    showPushNotification(notification.title, notification.body);
  });
}
```

ui/ar-maintenance-overlay.template

javascript

```
// AI-generated AR interface for maintenance guidance
arApplication("Maintenance Assistant") {
 spatialUI {
  // Equipment recognition and overlay
  equipmentRecognition {
   onEquipmentDetected((equipment) => {
    // Overlay information panel next to detected equipment
    spatialPanel(
     position: equipment.boundingBox.topRight,
     worldLocked: true
    ) {
     equipmentInfo {
      title(equipment.name)
      status(equipment.status, color: equipment.status === "warning" ? "orange" : "green")
      temperature(equipment.currentTemperature + "°C")
      lastMaintenance(equipment.lastMaintenance)
     }

     if (equipment.needsMaintenance) {
      maintenanceSteps {
       for (step in svc::Maintenance.getStepsFor(equipment.id)) {
        stepIndicator(
         number: step.number,
         description: step.description,
         completed: step.completed,
         position: step.spatial3DLocation
        ) {
         if (step.requiresTool) {
          toolIndicator(
           type: step.tool.type,
           position: step.tool.placement3D,
           toolType: step.tool.type,
           animation: "float"
          )
         }
        }
       }
      }
     }
    }
   })
  }
```

```
    // Voice commands for hands-free operation
    voiceCommands {
      command("next step", action: svc::AR.nextStep())
      command("previous step", action: svc::AR.previousStep())
      command("complete task", action: svc::AR.completeTask())
      command("call expert", action: svc::Communication.startVoiceCall())
      command("report problem", action: svc::AR.openVoiceReport())
    }

    // Gesture recognition
    gestureRecognition {
      gesture("pinch", target: "equipment", action: svc::AR.selectEquipment)
      gesture("tap", target: "button", action: svc::AR.activateButton)
      gesture("swipe_left", action: svc::AR.nextStep)
      gesture("swipe_right", action: svc::AR.previousStep)
    }
  }

  // Real-time collaboration with remote experts
  onMessage("ar/session/" + svc::AR.sessionId + "/expert_join", (expertData) => {
    showSpatialNotification("Expert " + expertData.name + " joined to assist");

    // Show expert's cursor in AR space
    expertCursor(
      position: expertData.cursor3D,
      label: expertData.name,
      color: "green"
    )
  });

  // Equipment sensor data updates
  onMessage("ui/updates/equipment/+", (equipmentId, data) => {
    if (svc::AR.currentEquipment.id === equipmentId) {
      updateEquipmentOverlay(data);
    }
  });
}
```

---

5. Deployment Configurations

deployment/production-global.yml

yaml

```yaml
# Production deployment across multiple clouds
deployment:
 name: "smart-factory-monitor-production"
 target: "multi-cloud"

infrastructure:
 clouds:
  primary:
   provider: "aws"
   regions: ["us-east-1", "eu-west-1"]

  secondary:
   provider: "azure"
   regions: ["eastus", "westeurope"]

  edge:
   provider: "aws-wavelength"
   locations: ["factory-floor-1", "factory-floor-2"]

scaling:
 sensor-processor:
  min-instances: 10
  max-instances: 1000
  auto-scale:
   metric: "message-queue-depth"
   target: 100
   scale-up-cooldown: "30s"
   scale-down-cooldown: "5m"

 alert-manager:
  min-instances: 3
  max-instances: 50
  auto-scale:
   metric: "cpu-utilization"
   target: 70

 prediction-engine:
  min-instances: 2
  max-instances: 20
  resources:
   gpu: "nvidia-v100"
   memory: "16GB"
```

```yaml
networking:
 load-balancer:
  type: "application"
  ssl-termination: true
  ddos-protection: true

 cdn:
  provider: "cloudflare"
  cache-policy: "aggressive"
  edge-locations: "global"

security:
 encryption:
  at-rest: "aes-256"
  in-transit: "tls-1.3"

 network:
  vpc-isolation: true
  private-subnets: true
  bastion-hosts: true

 certificates:
  provider: "letsencrypt"
  auto-renewal: true

monitoring:
 metrics:
  provider: "prometheus"
  retention: "30d"

 logging:
  provider: "elasticsearch"
  retention: "90d"

 alerting:
  provider: "pagerduty"
  escalation-policy: "critical-infrastructure"

backup:
 databases:
  frequency: "hourly"
  retention: "30d"
  cross-region: true

 disaster-recovery:
  rto "15m" # Recovery Time Objective
```

```
    rto: "15m" # Recovery Time Objective
    rpo: "5m" # Recovery Point Objective
    failover: "automatic"
```

deployment/dev-local.yml

yaml

```yaml
# Local development - same code, different scale
deployment:
 name: "smart-factory-monitor-dev"
 target: "local"

infrastructure:
 type: "docker-compose"
 services:
  - postgresql
  - mongodb
  - redis
  - emqx-mqtt

scaling:
 # Everything runs as single instances locally
 sensor-processor:
  instances: 1
  resources:
   memory: "512MB"

 alert-manager:
  instances: 1
  resources:
   memory: "256MB"

 prediction-engine:
  instances: 1
  resources:
   memory: "1GB"
   gpu: false # Use CPU inference locally

networking:
 ports:
  web: "3000"
  api: "8080"
  mqtt: "1883"

development:
 hot-reload: true
 debug-mode: true
 mock-data: true

testing:
```

```
unit-tests: true
integration-tests: true
mock-iot-devices: 10
```

Key Architecture Insights for AI Application Generation

1. Universal Abstraction Layer

The platform provides consistent interfaces that work identically whether running:

- **Locally:** Direct method calls, single process
- **Cloud:** MQTT messaging, distributed services
- **Edge:** Hybrid local/remote execution

AI Benefit: Agents generate one application specification that works everywhere.

2. Declarative Everything

Every aspect is declarative and file-based:

- **Business Logic:** Schema-driven service functions
- **User Interfaces:** Template-based, multi-platform
- **Data Models:** JSON Schema definitions
- **Deployment:** YAML configuration files
- **Security:** Policy-as-code specifications

AI Benefit: Agents work with high-level descriptions, not imperative code.

3. Real-time by Default

MQTT messaging provides:

- **Instant updates** across all platforms (web, mobile, AR, IoT)
- **Event-driven architecture** with reliable delivery
- **Pub/sub scaling** to millions of concurrent connections

AI Benefit: Real-time features are automatic, not custom implementations.

4. Cross-Platform Native

Same application specification generates:

- **Web dashboards** with real-time updates
- **Mobile apps** with push notifications
- **AR interfaces** with spatial tracking
- **IoT device interfaces** with optimized protocols

AI Benefit: One UI description→ all platform implementations.

5. Enterprise Security Integration

Security is embedded at the platform level:

- **OIDC authentication** works across all platforms
- **RBAC authorization** enforced at library level
- **Audit logging** automatic for compliance
- **Data encryption** transparent to applications

AI Benefit: Security is declarative policy, not custom code.

Why This Architecture Wins for AI Applications

1. Specification-Driven Development

AI agents work with high-level specifications, not low-level implementation details. The platform translates specifications into optimized runtime behavior.

2. Universal Deployment Model

Same application code runs from laptop development to global production without changes. AI agents generate portable applications.

3. Platform Effect Amplification

As more AI agents use the specification, the platform becomes more valuable:

- More deployment targets supported
- Better optimization algorithms
- Richer template libraries
- Stronger ecosystem effects

4. Future-Proof Architecture

New platforms (future AR devices, IoT protocols, cloud providers) integrate by extending the specification format, not rewriting applications.

Complete Sample Deployment

An AI agent generates this complete application from a single prompt. The developer (or another AI agent) can then:

```bash
# Deploy locally for development
$ cloud-rails deploy smart-factory-monitor/ --config dev-local.yml

# Deploy to staging cloud
$ cloud-rails deploy smart-factory-monitor/ --config staging-cloud.yml

# Deploy to global production
$ cloud-rails deploy smart-factory-monitor/ --config production-global.yml
```

Same application code. Same specification files. Different scale and infrastructure.

This is the "SQL for AI" - a universal specification that AI agents can generate to create sophisticated, multi-platform, real-time applications that deploy anywhere.