DevOps Incident & On-Call Platform Hackathon 2026

OFFICIAL HACKATHON SUBJECT DOCUMENT

Event Duration: 29 hours
Date: February 9-10, 2026
Location: ENSA Khouribga (On-site)
Organization: OpenSource Days Event

TABLE OF CONTENTS

1. INTRODUCTION

Welcome Message

Welcome to the **DevOps Incident & On-Call Platform Hackathon 2026 – Local Edition**! This challenge pushes you to build a production-ready incident management and on-call platform—similar to PagerDuty or incident.io—while implementing the complete DevOps lifecycle using **entirely local infrastructure**.

This is the backup edition: Designed to run when cloud infrastructure is unavailable or for teams who prefer developing entirely on their local machines.

Event Theme

> **"Build Reliability: Automate Incident Response with Local DevOps Excellence"**

In modern DevOps and SRE practices, incidents are inevitable. What separates high-performing teams from struggling ones is how quickly and effectively they respond. Your mission is to build an intelligent incident management platform that automates alert aggregation, on-call scheduling, escalation, and response tracking—while demonstrating mastery of the complete DevOps toolchain **running entirely on Docker**.

Why This Matters

- Average cost of downtime: $5,600 per minute for enterprises
- MTTR (Mean Time To Resolve) is a critical SRE metric—top teams resolve incidents 60% faster
- On-call burnout affects 65% of engineers without proper tooling
- Organizations with incident automation reduce service disruptions by 45%

Your platform could be the foundation for the next generation of SRE tooling.

Key Differences from Cloud Edition

Aspect	Cloud Edition	Local Edition
Orchestration	Kubernetes + HPA	Docker Compose
Infrastructure	Cloud VMs, Load Balancers	localhost, Docker networks
Deployment Target	Remote K8s cluster	Local Docker daemon
Monitoring	Shared Prometheus/Grafana	Local containerized stack
Scaling	HorizontalPodAutoscaler	docker compose scale
IaC Tool	Terraform (cloud resources)	Docker Compose as IaC
Registry	Cloud container registry	Local Docker registry

Table 1: Cloud vs Local Edition Comparison

---

2. CHALLENGE OVERVIEW

Your Mission

Build a **comprehensive Incident & On-Call Management Platform** with Service-Oriented Architecture that handles the complete incident lifecycle—from alert ingestion through resolution—while implementing full DevOps automation including CI/CD pipelines, containerization, Docker Compose orchestration, monitoring, and automated security/quality gates.

Everything runs on your laptop. No cloud account required.

What You'll Build

A real-world SRE platform featuring:

1. **Alert Ingestion** - Receive and normalize alerts from monitoring systems
2. **Incident Management** - Track incident lifecycle and status
3. **On-Call Scheduling** - Manage rotations and escalation policies
4. **Smart Notifications** - Alert the right people at the right time
5. **SRE Dashboards** - Real-time incident metrics and MTTR tracking
6. **SOA Design** - Minimum 4 microservices with clear boundaries
7. **Full Containerization** - Docker containers for every service
8. **Docker Compose Orchestration** - Single-command deployment
9. **Complete CI/CD** - 7-stage automated pipeline (local execution)
10. **Observability Stack** - Prometheus + Grafana (containerized)
11. **Infrastructure as Code** - docker-compose.yml as IaC
12. **Security Gates** - Credentials scanning and code quality enforcement

Core Problem Statement

DevOps and SRE teams face critical operational challenges:

- **Alert Overload**: Hundreds of alerts with no intelligent correlation
- **Delayed Response**: Manual on-call processes slow incident acknowledgment
- **Poor Escalation**: Engineers miss critical alerts leading to extended downtime
- **No Visibility**: Lack of real-time metrics on incident response effectiveness
- **Manual Coordination**: Teams waste time coordinating who should respond
- **Inconsistent Process**: No standardized incident response workflow
- **MTTR Tracking Gaps**: No automated tracking of time-to-acknowledge and time-to-resolve

Your platform solves ALL of these problems through intelligent automation and a world-class DevOps implementation—**running entirely on localhost**.

---

3. PLATFORM SPECIFICATION

3.1 Platform Overview: Incident & On-Call Management

Core Functionality:

Your platform receives alerts from monitoring systems (Prometheus, custom apps, etc.), intelligently groups them into incidents, determines the current on-call engineer based on schedules, sends notifications, tracks the incident lifecycle (created → acknowledged → resolved), and provides comprehensive SRE metrics dashboards.

3.2 User Flows

Flow 1: Alert to Incident

1. Monitoring system (Prometheus) detects an issue (e.g., high CPU, 5xx errors)
2. Sends HTTP POST to your **Alert Ingestion Service** (http://localhost:8001/api/v1/alerts)
3. Alert Ingestion:
   - Validates and normalizes alert payload
   - Checks for similar open incidents (correlation by service + severity + time window)
   - Creates new incident OR attaches alert to existing incident
4. **Incident Management Service**:
   - Stores incident with metadata (service, severity, timestamp)
   - Calls **On-Call Service** to determine current on-call engineer
   - Calls **Notification Service** to send alert (mocked or real)
5. On-call engineer:
   - Opens web UI at http://localhost:8080, sees incident details
   - Clicks "Acknowledge" → incident status changes
   - Investigates and resolves issue
   - Clicks "Resolve" → incident closed, MTTR calculated

Flow 2: On-Call Schedule Management

1. SRE manager defines rotation:
   - Team "Platform Engineering": Alice (Week 1), Bob (Week 2), Carol (Week 3)
   - Primary and secondary engineers
   - Escalation policy: if no acknowledge in 5 minutes → escalate to secondary
2. **On-Call Service** computes "current on-call" based on:
   - Current date/time
   - Rotation schedule
   - Team assignment

3. When incident created → automatically assigns to current on-call

Flow 3: Incident Review & Metrics

1. After resolution, platform has complete timeline:
    - Alert timestamps
    - Incident created time
    - Time to acknowledge (MTTA)
    - Time to resolve (MTTR)
    - Status changes and notes
2. **Grafana dashboards** (http://localhost:3000) display:
    - Open incidents by severity
    - MTTA and MTTR trends
    - Incidents per service/team
    - Alert noise metrics
    - On-call load distribution

3.3 Required Services (SOA)

Your platform MUST implement at least these **4 core microservices**:

Service	Responsibility
Alert Ingestion Service	Receives alerts from external systems, validates, normalizes, correlates into incidents
Incident Management Service	Manages incident lifecycle, status tracking, assignments, timeline, MTTA/MTTR calculation
On-Call & Escalation Service	Manages schedules, rotations, determines current on-call, handles escalation logic
Web UI / API Gateway	Frontend for users to view/manage incidents, dashboards, schedules

Table 2: Core Microservices

Optional 5th Service (bonus points):

- **Notification Service**: Handles sending alerts via multiple channels (mock email, webhook logs)

3.4 Key Features Checklist

Your platform must support:

Alert Management:

☐ Accept alerts via HTTP POST webhook

☐ Validate alert schema (service, severity, message, labels)

☐ Correlate alerts into incidents (same service + severity + 5min window)

☐ Store raw alert data with timestamps

Incident Management:

☐ Create incidents (manually or from alerts)

☐ Track status: open, acknowledged, in_progress, resolved

☐ Assign incidents to on-call engineers

☐ Calculate MTTA (time from creation to acknowledgment)

☐ Calculate MTTR (time from creation to resolution)

☐ Support adding notes/comments to incidents

☐ Link multiple alerts to one incident

On-Call Scheduling:

☐ Define rotation schedules per team/service

☐ Support primary and secondary on-call

☐ Calculate "who is on-call now" based on current time

☐ Support weekly or daily rotations

☐ Implement basic escalation (if no ACK in X minutes → escalate)

Notifications (optional but recommended):

☐ Send notifications when incident created (mock or real)

☐ Send escalation notifications

☐ Log all notification attempts

Web Interface:

☐ Dashboard showing current open incidents

☐ Incident detail view with timeline

☐ Acknowledge/Resolve buttons

☐ Display current on-call schedule

☐ Basic SRE metrics summary

---

4. TECHNICAL REQUIREMENTS

4.1 Mandatory Components

All projects MUST implement these **8 core DevOps components** adapted for local development:

Component 1: Service-Oriented Architecture (SOA)

Requirement	Specification
Architecture	Minimum 4 microservices (Alert Ingestion, Incident Management, On-Call, Web UI)
Communication	REST APIs over Docker network
Independence	Each service has own codebase and container
Data	Each service manages its own data (can share database with schema separation)
Tech Stack	Your choice (Node.js, Python, Go, Java, etc.)

Table 3: SOA Requirements

Service Requirements:

- Each service exposes /health endpoint
- Each service exposes /metrics endpoint (Prometheus format)
- Services communicate via HTTP REST APIs over Docker network
- Clear API contracts between services

Component 2: Containerization (Docker)

Requirement	Specification
Container Engine	Docker with Dockerfile for each service
Optimization	Multi-stage builds, Alpine/distroless base images
Size Limit	Maximum 500MB per container image
Security	Non-root user, no hardcoded secrets
Health Checks	HEALTHCHECK instruction in Dockerfile

Table 4: Containerization Standards

Dockerfile Standards:

- Use multi-stage builds for compiled languages
- Combine RUN commands to reduce layers
- Include .dockerignore file
- Set non-root USER directive
- Define HEALTHCHECK instruction
- Expose ports and set proper CMD/ENTRYPOINT

Component 3: Monitoring Stack (Prometheus + Grafana)

Requirement	Specification

Deployment	Prometheus and Grafana as Docker containers
Metrics Collection	Prometheus scraping all services via Docker network
Visualization	Grafana with minimum 2 dashboards (reduced for sprint format)
Custom Metrics	Application-level metrics (incidents, MTTA, MTTR, alerts)
System Metrics	Container CPU, memory, request rates per service

Table 5: Monitoring Requirements

Required Custom Metrics:

1. incidents_total{status="open|ack|resolved"}
2. incident_mtta_seconds (histogram)
3. incident_mttr_seconds (histogram)
4. alerts_received_total{severity="critical|high|medium|low"}
5. alerts_correlated_total{result="new_incident|existing_incident"}
6. oncall_notifications_sent_total{channel="mock|webhook"}
7. escalations_total{team="platform|frontend|backend"}

Required Grafana Dashboards — Minimum 2 (simplified for sprint format):

1. **Live Incident Overview** (Required)
    - Open incidents count by severity
    - MTTA gauge (current average)
    - MTTR gauge (current average)
    - Incidents created over time (time series)
    - Top noisy services (bar chart)
2. **SRE Performance Metrics** (Required)
    - MTTA trend (moving average)
    - MTTR trend (moving average)
    - Incident volume by service
    - Acknowledgment time distribution
    - Resolution time distribution
3. **System Health Dashboard** (Optional - Bonus +2 points)
    - Service availability (up/down status)
    - Request rate per service
    - Error rate per service
    - Container resource usage (CPU, memory)

Component 4: Orchestration (Docker Compose)

Requirement	Specification
Orchestrator	Docker Compose (single docker-compose.yml)
Services	All microservices + database + Prometheus + Grafana
Networking	Single Docker network for inter-service communication
Volumes	Named volumes for data persistence
Configuration	Environment variables and config files mounted as volumes

Table 6: Docker Compose Requirements

Docker Compose Requirements:

- Single docker-compose.yml defining entire stack
- All services on shared Docker network
- Health checks defined for each service
- Depends_on relationships properly configured
- Named volumes for database, Prometheus, Grafana
- Port mappings for external access (UI, Grafana, Prometheus)
- Environment variables for configuration
- Resource limits (optional but recommended)

Example structure:

```
version: '3.8'

services:
alert-ingestion:
build: ./services/alert-ingestion
ports:
- "8001:8001"
networks:
- incident-platform
healthcheck:
test: ["CMD", "curl", "-f", "http://localhost:8001/health"]
interval: 30s

incident-management:
build: ./services/incident-management
ports:
- "8002:8002"
networks:
- incident-platform
depends_on:
- database

database:
image: postgres:15-alpine
volumes:
```

```
- db-data:/var/lib/postgresql/data
environment:
POSTGRES_PASSWORD: hackathon2026

prometheus:
image: prom/prometheus:latest
volumes:
- ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
- prometheus-data:/prometheus
ports:
- "9090:9090"

grafana:
image: grafana/grafana:latest
volumes:
- grafana-data:/var/lib/grafana
- ./monitoring/grafana-dashboards:/etc/grafana/provisioning/dashboards
ports:
- "3000:3000"

volumes:
db-data:
prometheus-data:
grafana-data:

networks:
incident-platform:
driver: bridge
```

Component 5: CI/CD Pipeline (7 Stages)

Requirement	Specification
Pipeline Tool	**Local automation** (Shell scripts, Makefiles, or act for GitHub Actions)
Execution	**Strictly Local.** Must run on your machine without external cloud runners.
Stages	Quality → Security → Build → Scan → Test → Deploy → Verify
Pipeline as Code	Pipeline defined as code (e.g., pipeline.sh, Makefile, or .github/workflows)

Table 7: CI/CD Requirements

Mandatory Pipeline Stages (4 Required)

Stage 1: Code Quality & Testing

- Run linters
- Execute unit tests

Requirement: Test coverage must be ≥ 60%

Stage 2: Build Container Images

- Build Docker images for all services
- Tag images using commit SHA or version

Stage 3: Automated Deployment

- Run docker compose down (cleanup)
- Run docker compose up -d (start stack)

Constraint: Deployment must require **zero manual steps** once the script is triggered.

Stage 4: Post-Deployment Verification

- Polls /health endpoints until services are ready.
- Verify services are reachable on localhost

Recommended Implementation Strategy

Since you cannot rely on cloud runners (like GitHub Actions servers), you have two valid options:

1. **The "Simulated" Cloud approach:** Write standard GitHub Actions YAML files (.github/workflows/pipeline.yml) and run them locally using **nektos/act**.
2. **The "Scripted" approach:** Create a master executable script (e.g., ./run-pipeline.sh or make all) that sequentially executes the 7 stages on your local machine.

Advanced CI/CD Bonuses (Up to +10 Points)

- **Security Scanning (+3)**: GitLeaks, TruffleHog for secrets detection
- **Container Vulnerability Scanning (+3)**: Trivy or Grype against images
- **Integration Testing (+2)**: Automated API tests against running stack
- **Automated Rollback (+2)**: Detect failures and rollback to previous version

Component 6: Infrastructure as Code (Docker Compose as IaC)

Requirement	Specification
IaC Definition	docker-compose.yml + configuration files
Reproducibility	Anyone can run docker compose up and get full environment
Configuration	Prometheus config, Grafana provisioning, env files
Documentation	README with setup instructions

Table 8: IaC Requirements

Infrastructure to Define:

- Complete docker-compose.yml with all services
- Prometheus configuration (prometheus.yml)
- Grafana dashboard provisioning files
- Database initialization scripts (if needed)

- Environment variable files (.env)
- Network definitions
- Volume definitions

Component 7: Credentials Checking Tool

Requirement	Specification
Purpose	Detect and prevent exposed secrets in code
Integration	Pre-commit hook + CI/CD pipeline
Detection	Pattern matching, entropy analysis
Blocking	Prevent commits/deployments with secrets

Table 9: Credentials Checking

Must Detect:

- API keys and tokens
- Database passwords in code (use environment variables)
- Private keys
- High-entropy strings (potential secrets)

Implementation: Use tools like TruffleHog, GitLeaks, detect-secrets.

Component 8: Code Quality Overview Tool

Requirement	Specification
Purpose	Automated code quality analysis and gates
Coverage	Minimum 70% test coverage enforced
Analysis	Code complexity, duplication detection
Integration	CI/CD pipeline with quality gates
Visualization	Report showing quality metrics

Table 10: Code Quality Requirements

Quality Gates:

- Test coverage ≥ 60% (FAIL if below) — reduced for 28.5-hour sprint format
- No critical bugs
- Code duplication < 3%

4.2 Technology Stack Guidelines

You have freedom to choose your stack, but it must work with Docker.

Recommended Stacks:

Option 1: Node.js/TypeScript

- Services: Node.js + Express/NestJS
- Frontend: React/Vue + TypeScript
- Database: PostgreSQL or MongoDB
- Why: Fast development, excellent ecosystem

Option 2: Python

- Services: FastAPI/Flask
- Frontend: React/Vue
- Database: PostgreSQL
- Why: Great for data processing, simple deployment

Option 3: Go

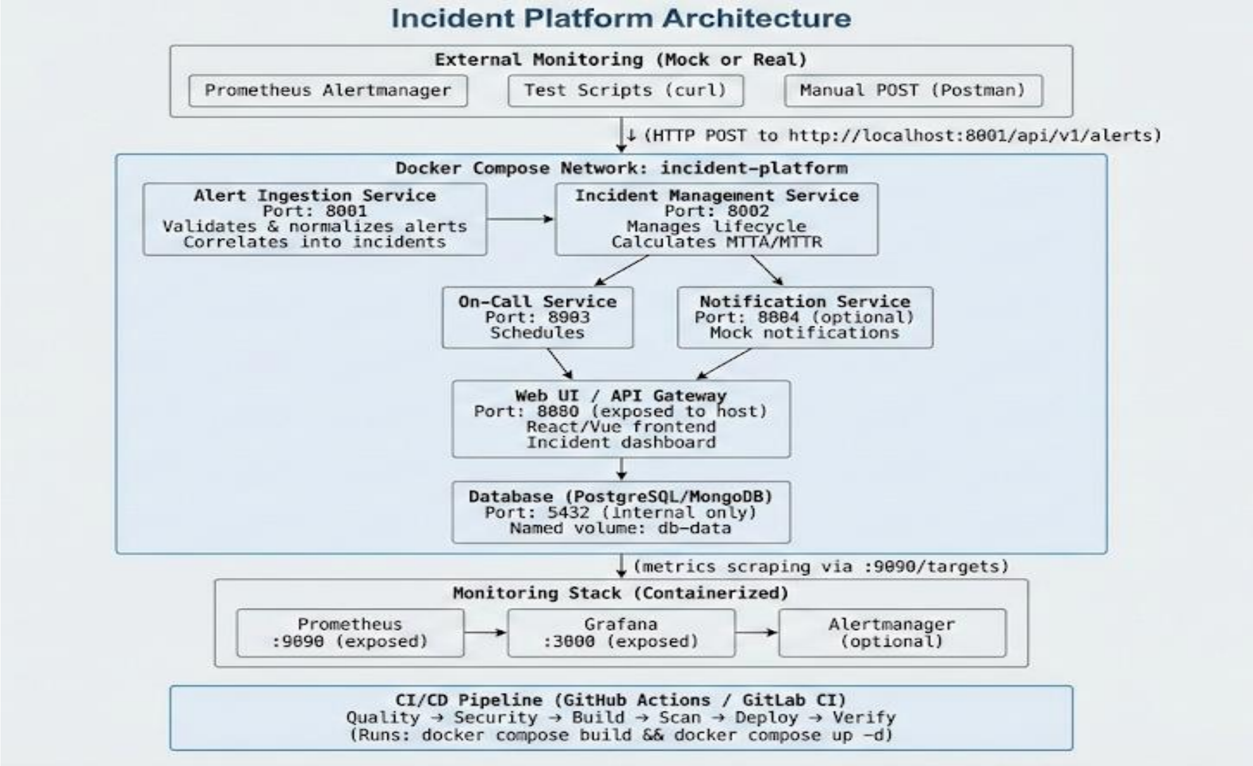
- Services: Go + Gin/Echo
- Frontend: React/Vue
- Database: PostgreSQL
- Why: High performance, minimal containers

Option 4: Polyglot (Mix)

- Alert Ingestion: Go (high throughput)
- Incident Management: Python (business logic)
- On-Call Service: Node.js
- Frontend: React

5. ARCHITECTURE DESIGN

5.1 System Architecture Diagram


Incident Platform Architecture
External Monitoring (Mock or Real)
Prometheus Alertmanager
Test Scripts (curl)
Manual POST (Postman)
↓ (HTTP POST to http://localhost:8001/api/v1/alerts)
Docker Compose Network: incident-platform
Alert Ingestion Service
Port: 8001
Validates & normalizes alerts
Correlates into incidents
Incident Management Service
Port: 8002
Manages lifecycle
Calculates MTTA/MTTR
On-Call Service
Port: 8003
Schedules
Notification Service
Port: 8004 (optional)
Mock notifications
Web UI / API Gateway
Port: 8080 (exposed to host)
React/Vue frontend
Incident dashboard
Database (PostgreSQL/MongoDB)
Port: 5432 (internal only)
Named volume: db-data
(metrics scraping via :9090/targets)
Monitoring Stack (Containerized)
Prometheus
:9090 (exposed)
Grafana
:3000 (exposed)
Alertmanager
(optional)
CI/CD Pipeline (GitHub Actions / GitLab CI)
Quality → Security → Build → Scan → Deploy → Verify
(Runs: docker compose build && docker compose up -d)

5.2 Service Communication Patterns

Synchronous (REST over Docker Network):

- Alert Ingestion → Incident Management: POST http://incident-management:8002/api/v1/incidents
- Incident Management → On-Call Service: GET http://oncall-service:8003/api/v1/oncall/current
- Incident Management → Notification Service: POST http://notification-service:8004/api/v1/notify
- Web UI → All Services: HTTP requests to respective ports

Container DNS Resolution:

- Docker Compose creates DNS entries for each service name
- Services communicate using service names (e.g., http://incident-management:8002)
- No hardcoded IPs needed

5.3 Data Flow: Alert to Resolution

1. **Alert Received**: External monitoring sends POST to http://localhost:8001/api/v1/alerts
2. **Normalization**: Alert Ingestion validates schema, extracts key fields
3. **Correlation**: Check for existing open incidents (same service + severity + 5min window)
4. **Incident Creation/Update**:
    - If new → create incident via Incident Management Service
    - If existing → attach alert to incident
5. **On-Call Lookup**: Incident Management calls On-Call Service for current on-call
6. **Notification**: Incident Management calls Notification Service (logs or mock)
7. **Acknowledgment**: Engineer opens http://localhost:8080, clicks "Acknowledge"
8. **Resolution**: Engineer resolves issue, clicks "Resolve" → MTTR calculated
9. **Metrics**: All events recorded, exposed via /metrics, scraped by Prometheus
10. **Dashboard**: Grafana (http://localhost:3000) displays real-time status

---

6. CORE SERVICES SPECIFICATION

6.1 Alert Ingestion Service

Responsibility: Receive alerts from external systems, validate, normalize, and correlate into incidents.

API Endpoints:

```
POST /api/v1/alerts
Request:
{
"service": "frontend-api",
"severity": "high",
"message": "HTTP 5xx error rate > 10%",
"labels": {
"environment": "production",
"region": "us-east-1"
},
"timestamp": "2026-03-08T15:30:00Z"
}
Response:
{
```

```
"alert_id": "alert-12345",
"incident_id": "incident-789",
"status": "correlated",
"action": "attached_to_existing_incident"
}
```

```
GET /api/v1/alerts/{alert_id}
GET /health
GET /metrics (Prometheus format)
```

Key Logic:

- Validate alert schema (required fields: service, severity, message)
- Normalize severity levels (critical/high/medium/low)
- Correlation algorithm: check for open incidents with same service AND severity within 5 minutes
- Store raw alert in database
- Expose metrics: alerts_received_total, alerts_correlated_total

6.2 Incident Management Service

Responsibility: Manage complete incident lifecycle, track status, calculate MTTA/MTTR.

API Endpoints:

```
POST /api/v1/incidents
GET /api/v1/incidents (with filters: status, severity, service)
GET /api/v1/incidents/{incident_id}
PATCH /api/v1/incidents/{incident_id}
GET /api/v1/incidents/{incident_id}/metrics
GET /health
GET /metrics
```

Key Logic:

- Create incidents from alerts or manual creation
- Track status: open → acknowledged → in_progress → resolved
- On incident creation: call On-Call Service, call Notification Service
- Calculate metrics: MTTA = acknowledged_at - created_at, MTTR = resolved_at - created_at
- Support adding notes/comments
- Expose metrics: incidents_total{status}, incident_mtta_seconds, incident_mttr_seconds

6.3 On-Call & Escalation Service

Responsibility: Manage on-call schedules, determine current on-call engineer, handle escalation.

API Endpoints:

```
GET /api/v1/schedules
POST /api/v1/schedules
GET /api/v1/oncall/current?team=platform-engineering
POST /api/v1/escalate
GET /health
GET /metrics
```

Key Logic:

- Store rotation schedules (weekly, daily)
- Calculate current on-call based on current timestamp and rotation rules
- Support primary and secondary on-call
- Escalation: if incident not acknowledged in X minutes → escalate to secondary
- Expose metrics: oncall_current{team}, escalations_total

6.4 Notification Service (Optional)

Responsibility: Send notifications via multiple channels (mock or real).

API Endpoints:

```
POST /api/v1/notify
GET /health
GET /metrics
```

Key Logic:

- For local edition, can mock notifications (log to console or file)
- Track delivery attempts
- Expose metrics: notifications_sent_total{channel,status}

6.5 Web UI / API Gateway

Responsibility: Frontend for users, dashboard, incident management interface.

Pages Required:

- Dashboard: List of open incidents, metrics summary
- Incident Detail: Timeline, alerts, status, actions
- On-Call Schedule: Current on-call, upcoming rotations
- SRE Metrics: Charts showing MTTA/MTTR trends

Access: http://localhost:8080

---

7. CONSTRAINTS & RULES

7.1 Hard Constraints (Mandatory)

Container Requirements:

- Image size < 500MB per service
- Multi-stage builds required
- Non-root user execution
- Health checks defined
- No hardcoded secrets

API Standards:

- Health endpoint at /health
- Metrics endpoint at /metrics (Prometheus format)
- Proper HTTP status codes
- API versioning (/api/v1/)

Code Quality:

- Test coverage ≥ 60% (reduced for sprint format)
- No critical bugs
- No hardcoded credentials
- Linter passes

Docker Compose:

- Single docker-compose.yml for entire stack
- All services on shared network
- Health checks and depends_on properly configured
- Named volumes for persistence

7.2 Bonus Features (Extra Points)

1. Real email integration (SendGrid, Mailgun) +3 pts
2. Webhook notifications +2 pts
3. Automated escalation workflow +2 pts
4. Historical incident analytics +1 pt
5. Log aggregation (Loki container) +2 pts
6. Distributed tracing (Jaeger container) +2 pts
7. Docker Compose scaling demo (docker compose up --scale) +1 pt

7.3 What's NOT Allowed

❌ Using managed platforms (PagerDuty, incident.io) as backend

❌ Hardcoded credentials anywhere (automatic disqualification)

✕ Missing core services (need minimum 4)

✕ Non-functional demo

✕ Kubernetes or cloud services

---

8. RESOURCES PROVIDED

8.1 Starter Templates

Available at: `https://github.com/oussamahc/incident-platform-local-templates.git`

Template 1: Docker Compose Starter

- Complete docker-compose.yml skeleton
- Prometheus + Grafana pre-configured
- Sample service definitions

Template 2: Node.js Microservice

- Express.js with Prometheus metrics
- Dockerfile with multi-stage build
- Health check endpoint

Template 3: Python FastAPI Service

- FastAPI with prometheus-client
- Optimized Dockerfile
- Sample API structure

Template 4: Sample Alert Payloads

- JSON examples
- curl commands to test Alert Ingestion
- Mock schedule data

8.2 Documentation & Guides

Visit: `https://competition.opensource14.com/hackathon`

Guides:

- Quick Start: Docker Compose Setup (10 min)
- Docker Optimization Best Practices
- Prometheus Custom Metrics Guide
- Building SRE Dashboards in Grafana
- CI/CD Pipeline Setup (Local)

8.3 Sample Commands

Deploy entire stack:

Clone your repo

```
git clone https://github.com/yourteam/incident-platform
cd incident-platform
```

Build all images

```
docker compose build
```

Start stack

```
docker compose up -d
```

View logs

```
docker compose logs -f
```

Check health

```
curl http://localhost:8001/health
curl http://localhost:8002/health
curl http://localhost:8003/health
curl http://localhost:8080/health
```

Send test alert

```
curl -X POST http://localhost:8001/api/v1/alerts
-H "Content-Type: application/json"
-d '{
"service": "frontend-api",
"severity": "high",
"message": "Test alert",
"labels": {"env": "prod"}
}'
```

Access services

Web UI: http://localhost:8080

Grafana: http://localhost:3000 (admin/admin)

Prometheus: http://localhost:9090

---

9. JUDGING CRITERIA

Scoring Breakdown (100 Points)

1. Platform Functionality (30 points)

Evaluation:

- All 4 core services working
- Alert ingestion and correlation
- Incident management (create, update, resolve)
- On-call scheduling and lookups
- Web UI usability
- End-to-end flow working

Scoring:

- **25-30**: Flawless functionality, excellent UX
- **18-24**: Good functionality, most features working
- **10-17**: Basic functionality, some incomplete
- **0-9**: Critical features missing or broken

2. DevOps Implementation (30 points)

Evaluation:

- CI/CD pipeline completeness (7 stages)
- Containerization quality
- Docker Compose configuration
- Automation level
- Infrastructure as code (docker-compose.yml + configs)

Scoring:

- **25-30**: Complete automation, excellent IaC, optimized containers
- **18-24**: Good pipeline, functional Compose setup
- **10-17**: Basic automation, acceptable setup
- **0-9**: Manual steps, poor configuration

3. Monitoring & SRE Metrics (20 points)

Evaluation:

- Prometheus metrics quality and coverage
- Grafana dashboards (3 required)
- Custom incident metrics (MTTA, MTTR, incidents)
- Real-time visibility

Scoring:

- **16-20**: Comprehensive metrics, excellent dashboards
- **11-15**: Good monitoring, functional dashboards

- **6-10**: Basic metrics, simple dashboards
- **0-5**: Minimal or missing monitoring

4. Architecture & Design (15 points)

Evaluation:

- SOA quality (service boundaries)
- API design and documentation
- Error handling
- Code organization

Scoring:

- **12-15**: Excellent SOA, production-ready design
- **8-11**: Good architecture, clear services
- **4-7**: Basic separation, functional
- **0-3**: Poor design, monolithic tendencies

5. Security & Quality (5 points)

Evaluation:

- Credentials scanning working
- Code quality gates enforced (60% coverage)
- No exposed secrets
- Container security

Scoring:

- **5**: Perfect security, strong quality gates
- **3-4**: Good security practices
- **1-2**: Basic security
- **0**: Security issues present

---

10. SUBMISSION REQUIREMENTS

10.1 Required Deliverables

1. Source Code Repository

- GitHub/GitLab repository
- All source code for services
- docker-compose.yml
- Dockerfiles for each service

- CI/CD pipeline configuration
- README.md with setup instructions

2. Documentation

- README.md with:
  - Architecture overview
  - Setup instructions (should be ≤ 5 commands)
  - API documentation
  - Team member roles
- API documentation (OpenAPI spec or equivalent)
- Architecture diagram

3. Demo Instructions

In README.md, provide:

Example expected content:

```
git clone <repo-url>
cd incident-platform
docker compose up -d
```

Wait 30 seconds for services to start

```
curl http://localhost:8001/health  # Should return 200
```

Open http://localhost:8080 for Web UI

Open http://localhost:3000 for Grafana (admin/admin)

Judges will run these exact commands to evaluate your project.

10.2 Submission Format

Primary Submission:

THE SUBMISSION SHOULD BE HERE : https://forms.gle/3dvbwCHcFmvJBpRR6

- GitHub/GitLab repository URL
- README.md must be comprehensive
- Repository must be public or judges granted access
- Architecture slides
- Performance metrics screenshots

Supplementary Materials (optional):

- Demo video (3-5 minutes)

Emergency Backup (In case of submission issues):

If the submission form is down or you face technical issues, email your **Repository URL** and **Team Name** to **oussama.ouadia.1@gmail.com** or DM **SAMATI** on Discord before 14:00.

10.3 Demo Checklist

Before submission, verify:

- ☐ docker compose up -d starts all services
- ☐ All services pass health checks
- ☐ Web UI accessible at http://localhost:8080
- ☐ Can send test alert via curl
- ☐ Alert creates incident in UI
- ☐ Can acknowledge and resolve incident
- ☐ Grafana shows metrics at http://localhost:3000
- ☐ Prometheus targets all healthy at http://localhost:9090
- ☐ README instructions are accurate
- ☐ CI/CD pipeline passes
- ☐ No hardcoded secrets in code

---

11. TIMELINE & MILESTONES

11.1 Hackathon Timeline (28.5 Hours)

Day 1: Monday, February 9

- **09:00** - Opening ceremony (virtual)
- **09:30 - OFFICIAL START**
- **09:30-11:00** - Architecture design, repository setup, docker-compose.yml skeleton
- **11:00-13:00** - Alert Ingestion Service (basic API)
- **13:00-14:00** - Lunch break
- **14:00-16:00** - Incident Management Service (basic CRUD)
- **16:00-18:00** - Database integration, services communicating
- **18:00-19:00** - Dinner break
- **19:00-21:00** - On-Call Service (basic scheduling logic)
- **21:00-24:00** - Web UI (basic dashboard)
- **Milestone**: By midnight, 4 core services containerized and communicating

Day 2: Tuesday, February 10

- **00:00-03:00** - Optional: Polish features, bug fixes

- **03:00-08:00** - Sleep break (recommended)
- **08:00-09:30** - Monitoring stack (Prometheus + Grafana with 2-3 dashboards)
- **09:30-11:30** - CI/CD pipeline (minimum 4 stages), testing
- **11:30-12:30** - Documentation, final testing, video demo (optional)
- **12:30-13:00** - Final submission preparation
- **13:00** - **HARD DEADLINE** - Submissions close
- **13:00-15:00** - Judging period
- **15:00-17:30** - Presentation Preparation for 5 finalist teams
- **17:30 – Pitching**
- **19:45** - Winner announcement

11.3 Recommended

Hour 0-3: Foundation (9:30 AM - 12:30 PM)

- Finalize architecture (keep it simple!)
- Create repository with clear structure
- Create docker-compose.yml skeleton with all services
- Setup database container
- First service containerized and running

Hour 3-8: Core Services (12:30 PM - 5:30 PM)

- Alert Ingestion Service with /health and /metrics endpoints
- Incident Management Service with basic CRUD
- Services can communicate over Docker network
- Database schema created
- Test with curl commands

Hour 8-14: Complete MVP (5:30 PM - 11:30 PM)

- On-Call Service with basic schedule lookup
- Web UI with incident list and detail pages
- End-to-end flow: Alert → Incident → Display in UI
- Docker Compose stack runs with single command

Hour 14-24: Integration & Monitoring (11:30 PM - 9:30 AM)

- Prometheus container scraping all services
- Grafana with 2-3 basic dashboards
- Custom metrics (incidents_total, mtta, mttr)
- Optional: Sleep 3-5 hours (highly recommended!)

Hour 24-28: Polish & Submit (9:30 AM - 1:00 PM)

- CI/CD pipeline (minimum 4 stages: Quality, Build, Deploy, Verify)
- Credentials scanning integrated
- README with clear setup instructions
- Test complete deployment from scratch
- Record 3-minute demo video (optional but recommended)

---

12. SUPPORT & MENTORSHIP

12.1 Communication Channels

Discord Server: https://discord.gg/DZeckXvahn

Channels:

- #announcements - Official updates
- #general - General discussion
- #tech-help - Technical questions
- #docker-help - Docker/Compose issues
- #prometheus-grafana - Monitoring help
- #cicd-help - Pipeline questions

12.2 Our Mentors

- Mr. Abdelfattah Hilmi
- Mr. Yasser Ramy

12.3 Common Issues & Solutions

Issue: Docker Compose services can't communicate

Check network

```
docker network ls
docker network inspect incident-platform_default
```

Services should use service names

CORRECT: http://incident-management:8002

WRONG: http://localhost:8002

Issue: Prometheus not scraping metrics

In prometheus.yml, add:

```
scrape_configs:
```

- job_name: 'incident-services'
  static_configs:
  - targets:
    - 'alert-ingestion:8001'
    - 'incident-management:8002'
    - 'oncall-service:8003'

Issue: Port already in use

Check what's using port

lsof -i :8080

Kill process or change port in docker-compose.yml

---

13. PRIZES & RECOGNITION

13.1 Prize Categories

1st Place

- Feature on hackathon website
- Premium Gifts (10000 DH Value)

2nd Place

- Feature on hackathon website
- Premium Gifts

3rd Place

- Feature on hackathon website
- Gifts

13.2 Recognition

All participants receive:

- Certificate of participation

---

14. FREQUENTLY ASKED QUESTIONS

Q: Do I need a cloud account?

A: No! This is the Local Edition. Everything runs on your laptop using Docker.

Q: What are the minimum hardware requirements?

A:

- 8 GB RAM (16 GB recommended)
- 20 GB free disk space
- Docker Desktop installed (Windows/Mac) or Docker Engine (Linux)

Q: Can I use Kubernetes?

A: No. Local Edition must use Docker Compose only. This is to ensure everyone can run it on their laptop without needing cloud access.

Q: Can services share a database?

A: Yes, but use schema separation (e.g., different tables or databases). Each service should manage its own data.

Q: Do notifications need to be real?

A: No. You can mock them (log to console). Real email integration is a bonus (+3 points).

Q: How will judges test my project?

A: Judges will clone your repo and run `docker compose up -d`. Your README must have clear instructions.

Q: Can I use external APIs?

A: Yes, but ensure your platform works without them. External APIs should be optional enhancements.

Q: What if docker compose doesn't work on judge's machine?

A: Provide a demo video as backup. But your docker-compose.yml must be correct and tested.

Q: Can I pre-build parts before hackathon starts?

A: No. All code must be written during the 29-hour window. You can read docs and plan architecture beforehand.

Q: How many team members allowed?

A: 3-5 members per team.

Q: Can I use AI coding assistants?

A: Yes, tools like GitHub Copilot are allowed. But you must understand and be able to explain all code.

Q: What if I can't finish all features?

A: Submit what you have. Partial implementations are accepted. Focus on core functionality first.

SUPPORT CONTACTS

General Questions: Join Discord
Technical Issues: Join Discord
Discord https://discord.gg/DZeckXvahn
Documentation: https://competition.opensource14.com/hackathon

CONCLUSION

You're ready to build an incredible incident management platform **entirely on your laptop**!

What you'll accomplish:

- Full microservices architecture
- Complete Docker Compose orchestration
- Professional monitoring with Prometheus + Grafana
- Automated CI/CD pipeline
- Production-quality incident management system

Key Success Factors:

1. Start with architecture—plan before coding
2. Get docker-compose.yml working early
3. Implement core features before polish
4. Test end-to-end flow frequently
5. Document as you build

Good luck!

Build something amazing!

Document Version: 1.0 (Local Edition)
Questions? Join #tech-help on Discord