CUET MICRO-OPS HACKATHON 2025

Delineate Hackathon Challenge - CUET Fest 2025

GITHUB REPO
https://github.com/bongodev/cuet-micro-ops-hackthon-2025

The Scenario

This microservice simulates a real-world file download system where processing times vary significantly:

```
┌──────────────────────────────────────────────────────────────────────┐
│                      Download Processing Time                        │
├──────────────────────────────────────────────────────────────────────┤
│  Fast Downloads    ████████▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  ~10-15s  │
│  Medium Downloads  ████████████████████▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  ~30-60s  │
│  Slow Downloads    ████████████████████████████████████████  ~60-120s │
└──────────────────────────────────────────────────────────────────────┘
```

Why does this matter?

When you deploy this service behind a reverse proxy (Cloudflare, nginx, AWS ALB), you'll encounter:

Problem	Impact
Connection Timeouts	Cloudflare's 100s timeout kills long requests
Gateway Errors	Users see 504 errors for slow downloads
Poor UX	No progress feedback during long waits
Resource Waste	Open connections consume server memory

Try it yourself:

```
# Start the server (10-120s random delays)
npm run start

# This request will likely timeout (REQUEST_TIMEOUT_MS=30s)
curl -X POST http://localhost:3000/v1/download/start \
  -H "Content-Type: application/json" \
  -d '{"file_id": 70000}'

# Watch the server logs - you'll see:
# [Download] Starting file_id=70000 | delay=85.3s (range: 10s-120s) | enabled=true
```

Your challenge: Design solutions that handle these variable processing times gracefully!

Hackathon Challenges

Challenge	Max Points	Difficulty
Challenge 1: S3 Storage Integration	15	Medium
Challenge 2: Architecture Design	15	Hard
Challenge 3: CI/CD Pipeline	10	Medium
Challenge 4: Observability (Bonus)	10	Hard
Maximum Total	50	

Challenge 1: Self-Hosted S3 Storage Integration

Your Mission

The current Docker configuration does not include a self-hosted S3-compatible storage service. Your challenge is to:

1. Modify the Docker Compose files (`docker/compose.dev.yml` and/or `docker/compose.prod.yml`) to include a self-hosted S3-compatible storage service
2. Configure the API to connect to your storage service
3. Verify the health endpoint returns `"storage": "ok"`

Recommended S3-Compatible Storage Options

Option 1: RustFS (Recommended)

RustFS is a lightweight, high-performance S3-compatible object storage written in Rust.

Option 2: MinIO

MinIO is a popular, production-ready S3-compatible object storage.

Requirements

Your solution must:

> Add an S3-compatible storage service to Docker Compose
> Create the required bucket (`downloads`) on startup
> Configure proper networking between services
> Update environment variables to connect the API to storage
> Pass all E2E tests (`npm run test:e2e`)
> Health endpoint must return `{"status": "healthy", "checks": {"storage": "ok"}}`

Hints

1. The API expects these S3 environment variables:
    - `S3_ENDPOINT` - Your storage service URL (e.g., `http://minio:9000`)
    - `S3_ACCESS_KEY_ID` - Access key
    - `S3_SECRET_ACCESS_KEY` - Secret key
    - `S3_BUCKET_NAME` - Bucket name (use `downloads`)
    - `S3_FORCE_PATH_STYLE` - Set to `true` for self-hosted S3
2. Services in Docker Compose can communicate using service names as hostnames
3. You may need an init container or script to create the bucket
4. Check the `/health` endpoint to verify storage connectivity

Testing Your Solution

```
# Run the full test suite
npm run test:e2e

# Or test manually
curl http://localhost:3000/health
# Expected: {"status":"healthy","checks":{"storage":"ok"}}

curl -X POST http://localhost:3000/v1/download/check \
  -H "Content-Type: application/json" \
  -d '{"file_id": 70000}'
```

Challenge 2: Long-Running Download Architecture Design

The Problem

This microservice handles file downloads that can vary significantly in processing time:

- Fast downloads: Complete within ~10 seconds
- Slow downloads: Can take up to 120+ seconds

When integrating this service with a frontend application or external services behind a reverse proxy (like Cloudflare, nginx, or AWS ALB), you will encounter critical issues:

1. Connection Timeouts: Proxies like Cloudflare have default timeouts (100 seconds) and will terminate long-running HTTP connections
2. User Experience: Users waiting 2+ minutes with no feedback leads to poor UX
3. Resource Exhaustion: Holding HTTP connections open for extended periods consumes server resources
4. Retry Storms: If a client's connection is dropped, they may retry, creating duplicate work

Experience the Problem

```
# Start with production delays (10-120 seconds)
npm run start

# Try to download - this will likely timeout!
curl -X POST http://localhost:3000/v1/download/start \
  -H "Content-Type: application/json" \
  -d '{"file_id": 70000}'

# Server logs will show something like:
# [Download] Starting file_id=70000 | delay=95.2s (range: 10s-120s) | enabled=true
# But your request times out at 30 seconds (REQUEST_TIMEOUT_MS)
```

Your Mission

Write a complete implementation plan that addresses how to integrate this download microservice with a fullstack application while handling variable download times gracefully.

Deliverables

Create a document (`ARCHITECTURE.md`) that includes:

1. Architecture Diagram

- Visual representation of the proposed system
- Show all components and their interactions
- Include data flow for both fast and slow downloads

2. Technical Approach

Choose and justify ONE of these patterns (or propose your own):

Option A: Polling Pattern

```
Client → POST /download/initiate → Returns jobId immediately
Client → GET /download/status/:jobId (poll every N seconds)
Client → GET /download/:jobId (when ready)
```

Option B: WebSocket/SSE Pattern

```
Client → POST /download/initiate → Returns jobId
Client → WS /download/subscribe/:jobId (real-time updates)
Server → Pushes progress updates → Client
```

Option C: Webhook/Callback Pattern

```
Client → POST /download/initiate { callbackUrl: "..." }
Server → Processes download asynchronously
Server → POST callbackUrl with result when complete
```

Option D: Hybrid Approach

Combine multiple patterns based on use case.

3. Implementation Details

For your chosen approach, document:

- API contract changes required to the existing endpoints
- New endpoints that need to be created
- Database/cache schema for tracking job status
- Background job processing strategy (queue system, worker processes)
- Error handling and retry logic
- Timeout configuration at each layer

4. Proxy Configuration

Provide example configurations for handling this with:

- Cloudflare (timeout settings, WebSocket support)
- nginx (proxy timeouts, buffering)
- Or your preferred reverse proxy

5. Frontend Integration

Describe how a React/Next.js frontend would:

- Initiate downloads
- Show progress to users
- Handle completion/failure states

- Implement retry logic

Hints

1. Consider what happens when a user closes their browser mid-download
2. Think about how to handle multiple concurrent downloads per user
3. Consider cost implications of your chosen queue/database system
4. Research: Redis, BullMQ, AWS SQS, Server-Sent Events, WebSockets
5. Look into presigned S3 URLs for direct downloads

Challenge 3: CI/CD Pipeline Setup

Your Mission

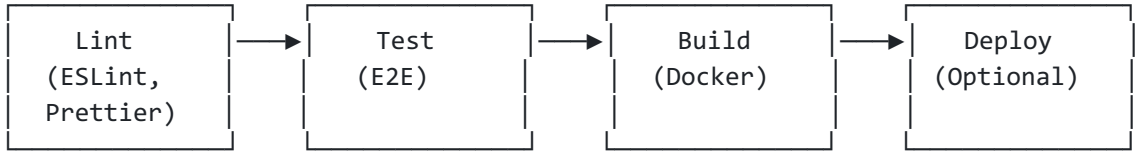
Set up a complete CI/CD pipeline for this service using a cloud provider's CI/CD platform. The pipeline must run all tests automatically on every push.

Requirements

Choose One Cloud Provider

Pipeline Stages

Your pipeline must include these stages:



Deliverables

1. Pipeline Configuration File
    - `.github/workflows/ci.yml` (GitHub Actions)
    - Equivalent for your chosen provider
2. Pipeline must:
    - Trigger on push to `main`/`master` branch

    - Trigger on pull requests
    - Run linting (`npm run lint`)
    - Run format check (`npm run format:check`)
    - Run E2E tests (`npm run test:e2e`)
    - Build Docker image
    - Cache dependencies for faster builds
    - Fail fast on errors
    - Report test results clearly
3. Documentation
    - Add a "CI/CD" section to README with:
        - Badge showing pipeline status
        - Instructions for contributors
        - How to run tests locally before pushing

Example: GitHub Actions (Reference)

A basic GitHub Actions workflow is already provided at `.github/workflows/ci.yml`. You may:

- Enhance the existing workflow
- Migrate to a different provider
- Add additional features (caching, parallelization, deployment)

Bonus Points

- Set up automatic deployment to a cloud platform (Railway, Render, Fly.io, etc.)
- Add security scanning (Snyk, CodeQL, Trivy)
- Implement branch protection rules
- Add Slack/Discord notifications for build status

Challenge 4: Observability Dashboard (Bonus)

Your Mission

Build a simple React UI that integrates with Sentry for error tracking and OpenTelemetry for distributed tracing, providing visibility into the download service's health and performance.

Testing Sentry Integration

The API includes a built-in way to test Sentry error tracking:

```
# Trigger an intentional error for Sentry testing
curl -X POST "http://localhost:3000/v1/download/check?sentry_test=true" \
  -H "Content-Type: application/json" \
  -d '{"file_id": 70000}'

# Response: {"error":"Internal Server Error","message":"Sentry test error..."}
# This error should appear in your Sentry dashboard!
```

Requirements

1. React Application Setup

Create a React application (using Vite or Next.js) that:

- Connects to this download API
- Displays download job status
- Shows real-time error tracking
- Visualizes trace data

2. Sentry Integration

Features to implement:

Error boundary wrapping the entire app
Automatic error capture for failed API calls
User feedback dialog on errors
Performance monitoring for page loads
Custom error logging for business logic errors

3. OpenTelemetry Integration

Features to implement:

Trace propagation from frontend to backend
Custom spans for user interactions
Correlation of frontend and backend traces
Display trace IDs in the UI for debugging

4. Dashboard Features

Build a dashboard that displays:

Feature	Description
Health Status	Real-time API health from `/health` endpoint
Download Jobs	List of initiated downloads with status
Error Log	Recent errors captured by Sentry
Trace Viewer	Link to Jaeger UI or embedded trace view
Performance Metrics	API response times, success/failure rates

5. Correlation

Ensure end-to-end traceability:

```
User clicks "Download" button
    │
    ▼
Frontend creates span with trace-id: abc123
    │
    ▼
API request includes header: traceparent: 00-abc123-...
    │
    ▼
Backend logs include: trace_id=abc123
    │
    ▼
Errors in Sentry tagged with: trace_id=abc123
```

Deliverables

1. React Application in a `frontend/` directory
2. Docker Compose update to include:
    - Frontend service
    - Jaeger UI accessible for trace viewing
3. Documentation on how to:
    - Set up Sentry project and get DSN

    - Configure OpenTelemetry collector
    - Run the full stack locally

Resources

- Sentry React SDK
- OpenTelemetry JavaScript
- Jaeger UI
- W3C Trace Context

Technical Requirements

Requirement	Version
Node.js	>= 24.10.0
npm	>= 10.x
Docker	>= 24.x
Docker Compose	>= 2.x

Tech Stack

- Runtime: Node.js 24 with native TypeScript support
- Framework: Hono - Ultra-fast web framework
- Validation: Zod with OpenAPI integration
- Storage: AWS S3 SDK (S3-compatible)
- Observability: OpenTelemetry + Jaeger
- Error Tracking: Sentry
- Documentation: Scalar OpenAPI UI

Quick Start

Local Development

```
# Install dependencies
npm install

# Create environment file
cp .env.example .env

# Start development server (with hot reload, 5-15s delays)
npm run dev

# Or start production server (10-120s delays)
npm run start
```

The server will start at http://localhost:3000

- API Documentation: http://localhost:3000/docs
- OpenAPI Spec: http://localhost:3000/openapi

Using Docker

```
# Development mode (with Jaeger tracing)
npm run docker:dev

# Production mode
npm run docker:prod
```

Environment Variables

Create a `.env` file in the project root:

```
# Server
NODE_ENV=development
PORT=3000

# S3 Configuration
S3_REGION=us-east-1
S3_ENDPOINT=http://localhost:9000
S3_ACCESS_KEY_ID=minioadmin
S3_SECRET_ACCESS_KEY=minioadmin
S3_BUCKET_NAME=downloads
S3_FORCE_PATH_STYLE=true

# Observability (optional)
SENTRY_DSN=
OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4318

# Rate Limiting
REQUEST_TIMEOUT_MS=30000
RATE_LIMIT_WINDOW_MS=60000
RATE_LIMIT_MAX_REQUESTS=100

# CORS
CORS_ORIGINS=*

# Download Delay Simulation
DOWNLOAD_DELAY_ENABLED=true
DOWNLOAD_DELAY_MIN_MS=10000
DOWNLOAD_DELAY_MAX_MS=200000
```

API Endpoints

Method	Endpoint	Description
GET	`/`	Welcome message
GET	`/health`	Health check with storage status
POST	`/v1/download/initiate`	Initiate bulk download job
POST	`/v1/download/check`	Check single file availability
POST	`/v1/download/start`	Start download with simulated delay

Testing the Long-Running Download

```
# With dev server (5-15s delays)
npm run dev
curl -X POST http://localhost:3000/v1/download/start \
  -H "Content-Type: application/json" \
  -d '{"file_id": 70000}'

# With production server (10-120s delays) - may timeout!
npm run start
curl -X POST http://localhost:3000/v1/download/start \
  -H "Content-Type: application/json" \
  -d '{"file_id": 70000}'
```

Available Scripts

```
npm run dev          # Start dev server (5-15s delays, hot reload)
npm run start        # Start production server (10-120s delays)
npm run lint         # Run ESLint
npm run lint:fix     # Fix linting issues
npm run format       # Format code with Prettier
npm run format:check # Check code formatting
npm run test:e2e     # Run E2E tests
npm run docker:dev   # Start with Docker (development)
npm run docker:prod  # Start with Docker (production)
```

Project Structure

```
.
├── src/
│   └── index.ts          # Main application entry point
├── scripts/
│   ├── e2e-test.ts       # E2E test suite
│   └── run-e2e.ts        # Test runner with server management
├── docker/
│   ├── Dockerfile.dev    # Development Dockerfile
│   ├── Dockerfile.prod   # Production Dockerfile
│   ├── compose.dev.yml   # Development Docker Compose
│   └── compose.prod.yml  # Production Docker Compose
├── .github/
│   └── workflows/
│       └── ci.yml        # GitHub Actions CI pipeline
├── package.json
├── tsconfig.json
└── eslint.config.mjs
```

Security Features

- Request ID tracking for distributed tracing
- Rate limiting with configurable windows
- Security headers (HSTS, X-Frame-Options, etc.)
- CORS configuration
- Input validation with Zod schemas
- Path traversal prevention for S3 keys
- Graceful shutdown handling