CI/CD & Monitoring

CI/CD Pipeline

Drop CI/CD Pipeline

Last updated: 2026-02-13 **Source:** `src/drop-app/package.json`, `Dockerfile`, `fly.toml`, `vitest.config.ts`, `playwright.config.ts`

Current State

Drop is in **MVP/pre-production** stage. Core CI/CD infrastructure exists including a GitHub Actions workflow.

What exists:

- GitHub Actions CI workflow (`.github/workflows/ci.yml`) with 5 jobs: lint-and-typecheck, test, build, e2e, docker-build
- Dockerfile with multi-stage build (`Dockerfile:1-63`)
- docker-compose for local and production (`docker-compose.yml`, `docker-compose.production.yml`)
- Fly.io deployment config (`fly.toml`)
- Vitest unit/integration test framework (`vitest.config.ts`)
- Playwright E2E test framework (`playwright.config.ts`)
- Health check endpoint (`/api/health`)
- QA report generation via `scripts/qa-report.js` (automated in CI)

What does not exist yet:

- Automated deployment pipeline (CI builds but does not deploy)
- Container registry integration
- Automated security scanning (npm audit, Snyk)
- Test coverage reporting
- Staging environment (Fly.io config exists but not deployed)

Build Pipeline

Step 1: Install Dependencies

```
npm ci
```

Installs exact versions from `package-lock.json`.

Step 2: Lint

```
npm run lint     # eslint
```

Step 3: Type Check

```
npx tsc --noEmit
```

Step 4: Unit + Integration Tests

```
npm test         # vitest run
```

Runs all tests in `tests/**/*.test.ts` (from `vitest.config.ts:7`). Test setup: `tests/setup.ts` sets `NODE_ENV=test`.

Step 5: Build

```
npm run build    # next build
```

Produces standalone output for Docker deployment.

Step 6: Docker Build

```
docker build -t drop-app .
```

Multi-stage build: deps -> builder -> runner.

Step 7: E2E Tests (requires running server)

```
npx playwright test
```

Requires dev server on `http://localhost:3000`. Playwright auto-starts it via `webServer` config.

Test Framework Configuration

Vitest (Unit + Integration)

Config: `src/drop-app/vitest.config.ts:1-15`

Setting	Value
Environment	`node`
Include	`tests/**/*.test.ts`
Setup	`tests/setup.ts`
Path alias	`@` -> `./src`

Playwright (E2E)

Config: `src/drop-app/playwright.config.ts:1-39`

Setting	Value
Test dir	`./tests/e2e`
Parallel	`false` (serial -- rate limiter is shared)
Workers	1
Retries (CI)	2
Timeout	30,000ms
Base URL	`http://localhost:3000`
Reporter	HTML
Trace	`on-first-retry`

Test projects:

1. `user-flows` -- Basic user journey tests (`user-flows.spec.ts`)
2. `full-flows` -- Complete feature journeys (`full-flows.spec.ts`)
3. `input-chaos` -- Malicious/edge-case input testing (`input-chaos.spec.ts`). Depends on `user-flows`.

Web server config: Auto-starts `npm run dev` for E2E tests. Reuses existing server if running. 30s timeout.

Deployment Targets

Fly.io (Staging)

Config: `fly.toml:1-28`

```
# Deploy to Fly.io staging
fly deploy

# Set secrets
fly secrets set JWT_SECRET="your-secret"
fly secrets set NEXT_PUBLIC_SERVICE_MODE="mock"
```

Region: `arn` (Stockholm) **Auto-scaling:** Scales to 0 when idle, auto-starts on request.

Docker (Self-hosted)

```
# Local dev (PostgreSQL 16 via Docker)
docker compose up -d

# Apply schema
make db-push
```

Existing GitHub Actions CI Workflow

File: `.github/workflows/ci.yml`

Triggers on push/PR to `main` or `master`:

```
Jobs:
  1. lint-and-typecheck — npm ci, npm run lint, tsc --noEmit
  2. test — npm ci, npm test --if-present (depends on lint-and-typecheck)
  3. build — npm ci, npm run build with JWT_SECRET placeholder (depends on lint-and-typecheck)
  4. e2e — npm ci, npx playwright install chromium, npm run build, npm run start (production
mode), npx playwright test user-flows + full-flows, generate QA report, upload artifacts
(depends on build)
  5. docker-build — docker build -t drop-app:ci (depends on test + build + e2e)
```

Artifacts uploaded:

- `playwright-report/` — Playwright HTML report (7 day retention)
- `qa-report.html` — QA metrics report (pass/fail, execution time)

Not yet implemented:

- Security scan (npm audit, Snyk)
- Deploy to staging (fly deploy)
- Deploy to production (manual approval gate)

Status: Full CI pipeline including E2E tests in place. CD deployment tracked in security hardening checklist (`security/hardening-checklist.md:120-126`).

Monitoring & Alerting

Drop Monitoring

Last updated: 2026-02-17 **Source:** `src/drop-app/src/app/api/health/route.ts`, `docker-compose.yml`, `fly.toml`, `src/lib/alerts.ts`

Health Check Endpoint

Route: `GET /api/health` **Source:** `src/drop-app/src/app/api/health/route.ts:1-35`

What It Checks

1. **Database connectivity** -- Executes `SELECT 1 as ok` against the database
2. **Database latency** -- Measures query execution time in milliseconds
3. **Database driver** -- Reports `pg` (PostgreSQL 16 via Drizzle ORM)
4. **Service mode** -- Reports `NEXT_PUBLIC_SERVICE_MODE` (`mock` or `live`)
5. **Application uptime** -- Tracks seconds since server start
6. **Application version** -- Reads from `npm_package_version` env var, defaults to `0.1.0`

Status Values

- **ok** -- All checks pass (HTTP 200)
- **degraded** -- DB query returned unexpected result (HTTP 200)
- **down** -- DB unreachable (HTTP 503)

Response Format

Healthy (200 OK):

```
{
  "data": {
    "status": "ok",
    "version": "0.1.0",
    "uptime": 3600,
```

```
    "checks": {
      "db": { "status": "pass", "latencyMs": 2, "driver": "pg" },
      "services": { "mode": "live" }
    },
    "timestamp": "2026-02-17T12:00:00.000Z"
  }
}
```

Down (503 Service Unavailable):

```
{
  "data": {
    "status": "down",
    "version": "0.1.0",
    "uptime": 3600,
    "checks": {
      "db": { "status": "fail" },
      "services": { "mode": "live" }
    },
    "timestamp": "2026-02-17T12:00:00.000Z"
  }
}
```

Container Health Checks

Docker Compose (MVP)

Source: `docker-compose.yml:12-17`

```
healthcheck:
  test: ["CMD", "wget", "--no-verbose", "--tries=1", "--spider",
"http://localhost:3000/api/health"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 10s
```

Docker Compose (Production)

Source: `docker-compose.production.yml:9-14`

Same health check configuration as MVP. Additionally, PostgreSQL has its own health check:

```
healthcheck:
  test: ["CMD-SHELL", "pg_isready -U drop"]
  interval: 10s
  timeout: 5s
  retries: 5
```

The `drop-app` service depends on PostgreSQL being healthy before starting (`depends_on.postgres.condition: service_healthy`).

Fly.io

Source: `fly.toml:19-23`

```
[[http_service.checks]]
  grace_period = "10s"
  interval = "30s"
  method = "GET"
  path = "/api/health"
  timeout = "5s"
```

Fly.io uses this health check to determine machine readiness and to route traffic.

Current Monitoring State

What Exists

- Health check endpoint with real database verification (not hardcoded)
- Container-level health checks (Docker + Fly.io)
- Automatic restart on failure (`restart: unless-stopped` in docker-compose)
- Auto-scaling on Fly.io (scale to zero, auto-start on request)

What Does Not Exist Yet

- External uptime monitoring service (see UptimeRobot setup below for recommended configuration)
- Application Performance Monitoring (APM)
- Structured logging (JSON format)
- Log aggregation and forwarding
- Database performance monitoring
- Rate limit monitoring/metrics
- Business metrics dashboard (transactions per hour, success rate)

Sentry Error Tracking

Status: REMOVED (MC #1271 — Sentry deinstalled)

Slack Alerting

Status: Implemented (MC #1183) **Source:** `src/lib/alerts.ts`, `instrumentation.ts`

Features

- Operational alerts sent to Slack webhook
- 10-minute cooldown per alert title (prevents spam)
- Severity-based emoji prefixes (ℹ️ info, ⚠️ warning, 🚨 critical)
- Graceful degradation when webhook URL not set (dev mode)

Setup Instructions

1. Create incoming webhook in Slack workspace:
   - Go to **Slack App Directory → Incoming Webhooks**
   - Choose channel (e.g., `#ops` or `#alerts`)
   - Copy webhook URL
2. Set environment variable:

```
# .env.local (server-side secret)
SLACK_WEBHOOK_URL=https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXX
```

Required Environment Variable

Variable	Required	Description
`SLACK_WEBHOOK_URL`	Yes (production)	Slack incoming webhook URL

Note: When `SLACK_WEBHOOK_URL` is not set, alerts are logged to console but not sent to Slack.

Alert Types and Severities

Severity	Emoji	Use Case
`info`	ℹ	Application startup, normal operations
`warning`	⚠	Degraded performance, non-critical issues
`critical`	🚨	Service outages, data loss, security incidents

Cooldown Behavior

- Each alert **title** has a 10-minute cooldown
- Same title sent within 10 minutes → skipped (prevents spam)
- Different titles → sent immediately (independent tracking)
- Cooldown resets on app restart (in-memory tracking)

Example: If "Database connection failed" is sent at 10:00, the next attempt before 10:10 will be skipped. But "High latency detected" can still be sent at 10:05.

Usage in Code

```
import { sendAlert } from '@/lib/alerts';

// Basic alert
await sendAlert({
  severity: 'critical',
  title: 'Database connection failed',
  message: 'PostgreSQL unreachable after 3 retries',
});

// Alert with details
await sendAlert({
  severity: 'warning',
  title: 'High error rate detected',
```

```
  message: '15 errors in last 5 minutes',
});
```

Current Integrations

- **App startup:** Sends info alert when server starts (`instrumentation.ts`)
- **App shutdown:** Sends info alert on SIGTERM/SIGINT (`instrumentation.ts`)
- **Error spike detection:** Automatically tracks errors and alerts when >5 errors occur in 60 seconds (`src/lib/alerts.ts:trackError`)
- **Unhandled exceptions:** Logged and tracked via process event handlers (`instrumentation.ts`)

Error Spike Detection

The alerting system automatically detects error spikes using a rolling window approach:

How it works:

1. Every server error (HTTP 5xx) is tracked via `trackError()`
2. Maintains rolling 1-minute window of error timestamps
3. When count exceeds threshold (5 errors in 60 seconds), sends critical alert
4. Integrates with middleware error handling

Threshold: 5 errors within 60 seconds **Alert severity:** Critical (□□) **Implementation:** `src/lib/alerts.ts:trackError()`, wired into `src/lib/middleware.ts:jsonError()`

Note: Error counter is in-memory and resets on app restart. For production workloads requiring persistent tracking, consider Redis-backed counters.

BetterStack Uptime Monitoring

Status: Ready to configure (setup guide available) **Documentation:** BETTERSTACK-SETUP.md

Overview

BetterStack provides external uptime monitoring independent of Drop's infrastructure. Unlike internal health checks (Docker, Fly.io) that only work when containers are running, BetterStack detects total infrastructure failures.

Free tier includes:

- 10 monitors (enough for Drop production)
- 3-minute check interval
- Unlimited integrations (Slack, email)
- Public status page
- SSL expiry monitoring

Recommended Monitors

Monitor	URL	Purpose	Expected Response
Health Endpoint	`https://drop.alai.no/api/health`	API + DB connectivity	`200`, body contains `"status":"ok"`
Landing Page	`https://drop.alai.no`	Public website	`200`, body contains `Send penger`
Multi-Region Check	`https://drop.alai.no/api/health`	Geographic availability	`200`, body contains `"status":"ok"`

Alert Escalation

BetterStack sends alerts through multiple channels:

```
Minute 0:   Alert fires → Slack #drop-ops (immediate)
Minute 5:   Still down → Email to alem@alai.no
Minute 15:  Still down → SMS (requires paid plan)
```

Status Page

Public status page shows real-time service status:

- **URL:** `https://drop-status.betteruptime.com`
- **Components:** API Health, Landing Page, Global Network
- **Auto-updates:** Incidents automatically posted and resolved
- **Subscriptions:** Users can subscribe to email updates

Setup Instructions

Complete setup guide with step-by-step instructions: BETTERSTACK-SETUP.md

Setup includes:

1. Account creation (free tier)
2. Configure 3 monitors (health, landing, multi-region)

3. Slack integration (`#drop-ops` channel)
4. On-call schedule and escalation policy
5. Public status page creation
6. Testing and verification

Key Features

Proactive monitoring:

- 3-minute check interval (free tier) or 30s (paid)
- Keyword verification (not just HTTP 200)
- SSL certificate expiry warnings (14 days)
- Multi-region checks (detect geographic issues)

Incident management:

- Automatic incident creation on downtime
- Status page updates (public transparency)
- Escalation to multiple channels (Slack → Email → SMS)
- Maintenance window support (suppress alerts during deployments)

Reporting:

- Uptime SLA tracking (99.9% target)
- Incident history and analysis
- Response time graphs
- Downtime duration reports

Integration with Drop Alerting

BetterStack complements Drop's internal alerting (`src/lib/alerts.ts`):

Feature	Drop Internal Alerts	BetterStack External
Detects	Application errors, error spikes	Infrastructure outages
When	App is running	App is unreachable
Source	Application logs	External HTTP checks
Delivery	Slack webhook (direct)	Escalation policy
Use case	Code bugs, DB issues	Container crashes, network failures

Example: Database connection fails:

1. Drop internal alert: "Database connection failed" → Slack `#drop-ops` (immediate)
2. BetterStack: Health check returns 503 → Slack `#drop-ops` + Email after 5 min

Maintenance Windows

When performing planned maintenance (deployments, upgrades):

1. Create maintenance window in BetterStack
2. Select affected monitors
3. Set duration (e.g., 1 hour)
4. **Effect:** Alerts suppressed, status page shows "Scheduled Maintenance"

Prevents: False downtime alerts during intentional service interruptions.

Best Practices

Do's:

- □ Test alerts monthly (pause monitor to verify escalation)
- □ Use keyword checks (not just HTTP status codes)
- □ Monitor SSL expiry (14-day warnings)
- □ Create maintenance windows for deployments
- □ Review incident history monthly

Don'ts:

- □ Don't ignore degraded status (investigate even if not fully down)
- □ Don't disable monitors (use pause for temporary suppression)
- □ Don't skip keyword checks (HTTP 200 ≠ working API)
- □ Don't rely solely on external monitoring (combine with internal checks)

External Uptime Monitoring (Alternative: UptimeRobot)

Status: Alternative to BetterStack (not recommended)

BetterStack is recommended over UptimeRobot for Drop because:

- Better Slack integration (richer notifications)
- Built-in status page (UptimeRobot charges extra)
- Better UI/UX for incident management
- More flexible escalation policies

UptimeRobot Setup (if BetterStack unavailable)

Cost: Free tier (50 monitors, 5-minute interval)

1. Create account at uptimerobot.com
2. Add HTTP(S) monitor:
   - **Friendly Name:** Drop Production
   - **URL:** `https://drop.alai.no/api/health`
   - **Monitoring Interval:** 5 minutes (free tier) or 1 minute (paid)
3. Configure alert contacts:
   - Slack webhook (via Alert Contacts)
   - Email (`alem@alai.no`)
4. Set **Keyword Monitoring:** Response contains `"status":"ok"`

Limitations:

- No built-in escalation policies (requires third-party integrations)
- Status page requires paid plan
- Less detailed incident reports
- 5-minute check interval (vs 3-minute for BetterStack free)

Monitoring Stack Summary

Implemented (MC #1184)

- ✅ **Health check endpoint** — `/api/health` with real database verification
- ✅ **Container health checks** — Docker + Fly.io auto-restart on failure
- ✅ **Error tracking** — Sentry REMOVED (MC #1271)
- ✅ **Slack alerting** — Operational alerts with cooldown protection
- ✅ **Lifecycle monitoring** — App startup and graceful shutdown alerts
- ✅ **Error spike detection** — Automatic alerting when >5 errors/minute

Recommended (Manual Setup)

- ☐ **External uptime monitoring** — UptimeRobot checking `/api/health` every 5 minutes
- ☐ **Structured logging** — JSON log format with request IDs for correlation
- ☐ **Metrics dashboard** — Request latency, error rates, database query times
- ☐ **Audit logging** — Tracked as security requirement (`security/drop-security-rapport.md` finding L3)

Future Enhancements (TODO)

- Database performance monitoring (slow query alerts)

- Rate limit metrics (track 429 errors per endpoint)
- Business metrics dashboard (transactions per hour, success rate)
- Redis-backed error counter (persistent across restarts)
- Per-endpoint error tracking (isolate problematic routes)

Environment Variables Reference

Required for Production

```
# Slack alerting
SLACK_WEBHOOK_URL=https://hooks.slack.com/services/T00000000/B00000000/XXXX
```

Dev Mode (All Optional)

All monitoring features gracefully degrade when env vars are not set:

- **No SLACK_WEBHOOK_URL:** Alerts logged to console only

This allows development to work without external services configured.

Production Deployment

Drop AWS Amplify Deployment Guide

> **Rebrand note (2026-02-14):** Originally titled "FontelePay". Product rebranded to **Drop**. Some env var references (Swan, Stripe) are FUTURE integrations — Drop uses a PSD2 pass-through model. See Drop CLAUDE.md.

This guide covers deploying Drop to AWS Amplify in the Frankfurt (eu-central-1) region.

Prerequisites

- AWS Account with Amplify access
- GitHub repository with Drop code
- Environment variables ready (see `.env.example`)

Step 1: Create Amplify App

1. Go to AWS Amplify Console
2. Ensure you're in **eu-central-1 (Frankfurt)** region
3. Click **Create new app**
4. Select **Host web app**

Step 2: Connect Repository

1. Choose **GitHub** as your Git provider
2. Authorize AWS Amplify to access your GitHub account
3. Select the **Drop** repository
4. Choose the branch to deploy (e.g., `main` or `production`)

Step 3: Configure Build Settings

Amplify will auto-detect Next.js. Verify the settings match `amplify.yml`:

```
version: 1
frontend:
  phases:
    preBuild:
      commands:
        - npm ci
    build:
      commands:
        - npm run build
  artifacts:
    baseDirectory: .next
    files:
      - '**/*'
  cache:
    paths:
      - node_modules/**/*
      - .next/cache/**/*
```

Step 4: Configure Environment Variables

In Amplify Console, go to **App settings > Environment variables** and add:

Required Variables

Variable	Description	Example
`NODE_ENV`	Environment	`production`
`NEXT_PUBLIC_APP_URL`	Your app URL	`https://drop.amplifyapp.com`

Swan BaaS

Variable	Description

`SWAN_API_URL`	`https://api.swan.io` (production)
`SWAN_CLIENT_ID`	OAuth2 Client ID
`SWAN_CLIENT_SECRET`	OAuth2 Client Secret
`SWAN_PROJECT_ID`	Project ID
`SWAN_WEBHOOK_SECRET`	Webhook validation secret

Stripe

Variable	Description
`NEXT_PUBLIC_STRIPE_PUBLISHABLE_KEY`	Publishable key (pk_live_...)
`STRIPE_SECRET_KEY`	Secret key (sk_live_...)
`STRIPE_WEBHOOK_SECRET`	Webhook secret (whsec_...)

Sumsub KYC

Variable	Description
`SUMSUB_APP_TOKEN`	App token
`SUMSUB_SECRET_KEY`	Secret key
`SUMSUB_WEBHOOK_SECRET`	Webhook secret
`SUMSUB_LEVEL_NAME`	KYC flow level

Database

Variable	Description
`DATABASE_URL`	PostgreSQL connection string
`REDIS_URL`	Redis connection string

Authentication

Variable	Description
`JWT_SECRET`	Min 32 characters
`SESSION_SECRET`	Min 32 characters

Step 5: Configure Next.js for Standalone Output

Update `next.config.ts` to enable standalone output for optimal Amplify deployment:

```
import type { NextConfig } from "next";

const nextConfig: NextConfig = {
  output: 'standalone',
};

export default nextConfig;
```

Step 6: Deploy

1. Click **Save and deploy**
2. Monitor the build in the Amplify Console
3. Once complete, your app will be available at `https://<branch>.<app-id>.amplifyapp.com`

Step 7: Configure Custom Domain (Optional)

1. Go to **App settings > Domain management**
2. Click **Add domain**
3. Enter your domain (e.g., `app.getdrop.no`)
4. Follow DNS configuration instructions
5. SSL certificate is automatically provisioned

Step 8: Set Up Branch Deployments

For staging/production workflows:

1. Go to **App settings > General**
2. Click **Edit**
3. Enable **Branch auto-detection**

4. Configure branch patterns:
   - `main` -> Production
   - `staging` -> Staging
   - `feature/*` -> Preview environments

Monitoring & Health Checks

Health Endpoint

The app exposes `/api/health` for load balancer health checks:

```
curl https://your-app.amplifyapp.com/api/health
```

Response:

```
{
  "status": "healthy",
  "timestamp": "2026-02-05T12:00:00.000Z",
  "version": "0.1.0",
  "uptime": 3600,
  "checks": {}
}
```

CloudWatch Logs

1. Go to **App settings > Monitoring**
2. View build logs and access logs
3. Set up CloudWatch alarms for errors

Troubleshooting

Build Fails

1. Check build logs in Amplify Console
2. Verify `package.json` scripts are correct
3. Ensure all dependencies are in `package.json`

Environment Variables Not Working

1. Verify variables are set in Amplify Console
2. Remember: `NEXT_PUBLIC_` prefix required for client-side access
3. Redeploy after changing environment variables

502/503 Errors

1. Check `/api/health` endpoint
2. Review CloudWatch logs
3. Verify database connections are correct
4. Check memory limits (adjust if needed)

Cold Starts

For serverless functions, cold starts may occur. Mitigate by:

1. Using connection pooling for databases
2. Keeping functions warm with scheduled pings
3. Optimizing bundle size

Security Checklist

- ☐ All secrets in Environment Variables (not in code)
- ☐ HTTPS enforced (automatic in Amplify)
- ☐ CORS configured correctly
- ☐ Rate limiting implemented
- ☐ Webhook signatures validated
- ☐ No sensitive data in logs

Cost Optimization

- Use `cache.paths` in `amplify.yml` to speed up builds
- Enable CloudFront caching for static assets
- Monitor build minutes usage
- Consider reserved concurrency for predictable traffic

Rollback

To rollback to a previous deployment:

1. Go to **Deployments** in Amplify Console
2. Find the previous successful deployment
3. Click **Redeploy this version**

Support

- AWS Amplify Documentation
- Next.js on AWS Amplify
- Drop Internal Docs

BetterStack Setup

BetterStack Uptime Monitoring Setup Guide

Last updated: 2026-02-20 **Related:** MONITORING.md, health-check.sh **Purpose:** External uptime monitoring for Drop production environment

---

Why BetterStack?

BetterStack provides external uptime monitoring independent of Drop's infrastructure:

- Detects infrastructure failures (AWS App Runner crashes, network issues)
- Alerts when the entire application is unreachable
- Provides uptime SLA tracking and historical reports
- Multiple notification channels (Slack, Email, SMS)
- Status page for client transparency

Key difference from internal health checks: Internal checks (Docker, Fly.io) only work when the container is running. BetterStack catches total outages.

Free Tier Limits

Plan: Free tier (no credit card required) **Limits:**

- **10 monitors** (enough for Drop production)
- **3-minute check interval** (paid plan: 30s minimum)
- **1 status page**
- **Unlimited team members**
- **Unlimited integrations** (Slack, email, webhooks)

Upgrade required for:

- Faster check intervals (<3 minutes)

- More than 10 monitors (e.g., multi-region checks)
- Advanced features (maintenance windows, custom headers)

Account Setup

Step 1: Create Account

1. Go to https://betterstack.com/uptime
2. Click **"Start free trial"** (becomes free tier after trial)
3. Sign up with Alem's email: `alem@alai.no`
4. Verify email address
5. Create workspace name: **"ALAI Products"** (shared across Drop, BasicFakta)

Step 2: Configure Team

1. Navigate to **Settings** > **Team**
2. Add team members:
   - `alem@alai.no` (Owner)
   - `john@basicconsulting.no` (Admin)
3. Set **Default timezone:** `Europe/Oslo` (UTC+1)

Monitor Configuration

Monitor 1: Health Endpoint (Primary)

Purpose: Verify API health and database connectivity

1. Go to **Monitors** > **Create Monitor**
2. Configure:
   - **Monitor name:** `Drop Health Check`
   - **Monitor type:** `HTTP`
   - **URL:** `https://drop.alai.no/api/health`
   - **Check interval:** `3 minutes` (free tier)
   - **Request timeout:** `5 seconds`
   - **Method:** `GET`
   - **Confirmation period:** `30 seconds` (1 retry before alerting)
3. **Expected Response:**
   - **Status code:** `200`

- **Keyword check:** Enable
  - Response body contains: `"status":"ok"`
  - **Why:** Ensures health endpoint returns valid JSON, not just HTTP 200
4. **Advanced settings:**
   - **Follow redirects:** `Enabled` (default)
   - **Verify SSL certificate:** `Enabled`
   - **SSL expiry warning:** `14 days before expiration`
5. Click **Create Monitor**

Monitor 2: Landing Page

Purpose: Verify public website availability

1. Go to **Monitors** > **Create Monitor**
2. Configure:
   - **Monitor name:** `Drop Landing Page`
   - **Monitor type:** `HTTP`
   - **URL:** `https://drop.alai.no`
   - **Check interval:** `3 minutes`
   - **Request timeout:** `10 seconds` (landing page has more assets)
   - **Method:** `GET`
   - **Confirmation period:** `30 seconds`
3. **Expected Response:**
   - **Status code:** `200`
   - **Keyword check:** Enable
     - Response body contains: `Send penger` (tagline verification)
4. Click **Create Monitor**

Monitor 3: Multi-Region Health Check

Purpose: Detect regional networking issues

1. Go to **Monitors** > **Create Monitor**
2. Configure:
   - **Monitor name:** `Drop Health (US East)`
   - **Monitor type:** `HTTP`
   - **URL:** `https://drop.alai.no/api/health`
   - **Check interval:** `3 minutes`
   - **Request timeout:** `5 seconds`
   - **Method:** `GET`
   - **Confirmation period:** `30 seconds`
3. **Expected Response:**
   - **Status code:** `200`

- **Keyword check:** Response body contains `"status":"ok"`
4. **Advanced settings:**
   - **Region:** `US East` (different from default EU region)
   - **Why:** Detects if Drop is unreachable from specific geographies
5. Click **Create Monitor**

Slack Integration

Step 1: Create Slack Incoming Webhook

1. Go to your Slack workspace: **alai-talk.slack.com**
2. Navigate to **Slack App Directory** > **Incoming Webhooks**
3. Click **Add to Slack**
4. Select channel: **#drop-ops** (create if doesn't exist)
5. Click **Add Incoming Webhooks Integration**
6. Copy webhook URL (format: `https://hooks.slack.com/services/T.../B.../XXX`)
7. Save this URL securely (needed for BetterStack)

Step 2: Add Slack Integration in BetterStack

1. In BetterStack, go to **Integrations**
2. Click **Add Integration** > **Slack**
3. Paste webhook URL from Step 1
4. Configure:
   - **Integration name:** `Drop Ops Slack`
   - **Notification channel:** `#drop-ops`
5. **Test integration:** Click **Send test message**
   - Verify message appears in `#drop-ops` channel
6. Click **Save Integration**

On-Call Team Setup

Step 1: Create On-Call Schedule

1. Go to **On-Call** > **Create Schedule**
2. Configure:
   - **Schedule name:** `Drop Primary On-Call`
   - **Timezone:** `Europe/Oslo`

3. Add rotation:
   - **Team member:** `alem@alai.no`
   - **Schedule type:** `24/7` (always on-call for now)
4. Click **Create Schedule**

Step 2: Configure Escalation Policy

1. Go to **Escalation Policies** > **Create Policy**
2. Configure:
   - **Policy name:** `Drop Production Incidents`
3. Add escalation steps:
   Step 1 (Immediate):
   - **Who:** `Drop Ops Slack` integration
   - **Delay:** `0 minutes`

   Step 2 (If still down after 5 minutes):
   - **Who:** `alem@alai.no` (Email)
   - **Delay:** `5 minutes`

   Step 3 (If still down after 15 minutes):
   - **Who:** `alem@alai.no` (SMS) — **Requires phone number**
   - **Delay:** `15 minutes`
   - **Note:** SMS requires paid plan or verified phone number
4. Click **Create Policy**

Step 3: Assign Policy to Monitors

1. Go to **Monitors**
2. For each monitor (`Drop Health Check`, `Drop Landing Page`, `Drop Health (US East)`):
   - Click monitor name
   - Go to **Settings** > **Escalation Policy**
   - Select: `Drop Production Incidents`
   - Click **Save**

Status Page Setup

Purpose

Public status page allows clients and stakeholders to check Drop availability without contacting support.

Step 1: Create Status Page

1. Go to **Status Pages** > **Create Status Page**
2. Configure:
   - **Page name:** `Drop Status`
   - **Subdomain:** `drop-status` (URL: `https://drop-status.betteruptime.com`)
   - **Custom domain (optional):** `status.drop.alai.no` (requires DNS setup)
3. **Design settings:**
   - **Logo:** Upload Drop logo (green rounded rectangle)
   - **Brand color:** `#0B6E35` (Drop primary green)
   - **Header text:** `Drop Status`
   - **Tagline:** `Real-time service status and incident updates`
4. **Visibility:**
   - **Public:** Yes (anyone can view)
   - **Search engine indexing:** No (prevent Google indexing)
5. Click **Create Status Page**

Step 2: Add Components

1. In the status page settings, go to **Components**
2. Click **Add Component**
3. Add three components:

   Component 1:
   - **Name:** `API & Health Endpoint`
   - **Linked monitor:** `Drop Health Check`
   - **Description:** `Core API functionality and database connectivity`

   Component 2:
   - **Name:** `Landing Page`
   - **Linked monitor:** `Drop Landing Page`
   - **Description:** `Public website and marketing content`

   Component 3:
   - **Name:** `Global Network`
   - **Linked monitor:** `Drop Health (US East)`
   - **Description:** `International access and routing`
4. Click **Save Components**

Step 3: Configure Incident Communication

1. Go to **Status Pages** > **Settings** > **Incident Updates**
2. Enable:
   - **Auto-create incidents:** Yes (when monitor goes down)
   - **Auto-resolve incidents:** Yes (when monitor recovers)
3. **Notification subscribers:**
   - **Email subscriptions:** Enabled (users can subscribe to updates)
   - **Webhook notifications:** Disabled (optional for future)

Step 4: Share Status Page

Once created, share the status page URL:

- **Internal:** Add to `#drop-ops` Slack channel description
- **External:** Link from Drop landing page footer (optional)
- **Clients:** Include in onboarding emails

Status Page URL: `https://drop-status.betteruptime.com`

Verification Checklist

After completing setup, verify:

- ☐ **Monitors running:** All 3 monitors show green status
- ☐ **Slack alerts working:** Test by pausing a monitor (triggers down alert)
- ☐ **Email notifications working:** Verify Alem receives email on test alert
- ☐ **Status page public:** Open status page URL in incognito mode
- ☐ **Escalation policy assigned:** All monitors use `Drop Production Incidents` policy
- ☐ **SSL expiry alerts:** Monitors configured to warn 14 days before cert expiration

---

Testing the Setup

Test 1: Manual Down Alert

1. Go to **Monitors** > `Drop Health Check`
2. Click **Pause Monitor** (simulates downtime)
3. **Expected behavior:**
   - Slack alert in `#drop-ops` within 30 seconds
   - Email to `alem@alai.no` after 5 minutes (if still paused)
4. Click **Resume Monitor** to clear alert

Test 2: Actual Downtime

1. SSH into production server (or use AWS App Runner console)
2. Stop the Drop application container temporarily
3. Wait for BetterStack to detect downtime (max 3 minutes + 30s confirmation)

4. **Expected behavior:**
   - Monitor shows red status
   - Slack alert in `#drop-ops`
   - Status page component shows "Down"
5. Restart application and verify recovery alert

Test 3: SSL Expiry Warning

1. Go to **Monitors** > `Drop Health Check`
2. Verify **SSL expiry warning** is enabled (14 days)
3. **Expected behavior:**
   - Alert sent 14 days before SSL certificate expiration
   - Action required: Renew certificate before expiry

Alert Examples

Downtime Alert (Slack)

```
□□Drop Health Check is DOWN

Monitor: Drop Health Check
Status: DOWN
Response: Connection timeout
Region: EU West
Time: 2026-02-20 10:30 UTC

View incident: https://betterstack.com/incidents/...
```

Recovery Alert (Slack)

```
□ Drop Health Check is UP

Monitor: Drop Health Check
Status: UP
Response: 200 OK (2ms)
Downtime duration: 3 minutes
Time: 2026-02-20 10:33 UTC
```

```
Incident closed: https://betterstack.com/incidents/...
```

SSL Expiry Warning (Email)

```
Subject: [BetterStack] SSL certificate expiring in 14 days

Monitor: Drop Health Check
Domain: drop.alai.no
Certificate expiry: 2026-03-06 23:59 UTC

Action required: Renew SSL certificate before expiration.
```

Maintenance Mode

When performing planned maintenance (deployments, infrastructure upgrades):

1. Go to **Maintenance Windows** > **Create Window**
2. Configure:
   - **Name:** `Drop Deployment`
   - **Start time:** `2026-02-20 22:00 UTC`
   - **Duration:** `1 hour`
   - **Affected monitors:** Select all Drop monitors
3. **Notification:**
   - **Status page update:** Yes (shows maintenance banner)
   - **Alert suppression:** Yes (no downtime alerts during window)
4. Click **Create Maintenance Window**

Effect: During maintenance, downtime alerts are suppressed and status page shows "Scheduled Maintenance" instead of "Down".

Best Practices

Do's

- **Test alerts monthly** — Pause a monitor to verify escalation works
- **Update on-call schedule** — Rotate on-call duty if team grows

- **Monitor SSL expiry** — Enable 14-day warnings to prevent outages
- **Use maintenance windows** — Prevent false alerts during deployments
- **Review incident history** — Monthly review of downtime patterns

Don'ts

- **Don't ignore degraded status** — Investigate even if not fully down
- **Don't disable monitors** — Use pause for temporary suppression only
- **Don't skip keyword checks** — HTTP 200 alone doesn't guarantee working API
- **Don't forget to update URLs** — When domain changes, update all monitors
- **Don't rely solely on external monitoring** — Combine with internal health checks

---

Troubleshooting

Monitor shows false positives (frequent up/down)

Cause: Network instability or slow response times **Fix:**

1. Increase **Request timeout** from 5s to 10s
2. Increase **Confirmation period** from 30s to 60s
3. Check Drop API latency in logs

Slack alerts not received

Cause: Webhook URL incorrect or channel archived **Fix:**

1. Go to **Integrations** > `Drop Ops Slack`
2. Click **Send test message**
3. If fails, regenerate webhook in Slack and update BetterStack

Email alerts delayed

Cause: Email provider spam filtering **Fix:**

1. Whitelist `notifications@betterstack.com` in email settings
2. Check spam/junk folder
3. Verify email address in BetterStack team settings

Status page not updating

Cause: Monitor not linked to status page component **Fix:**

1. Go to **Status Pages** > `Drop Status` > **Components**
2. Ensure each component has a **Linked monitor** assigned
3. Save changes and trigger test alert

Related Documentation

- MONITORING.md — Full monitoring stack overview
- health-check.sh — Internal health check script
- alerts.ts — Slack alerting implementation
- /api/health route — Health endpoint source code

Support

BetterStack Support:

- Documentation: https://betterstack.com/docs
- Email: support@betterstack.com
- Status: https://status.betterstack.com

Internal Contact:

- Slack: `#drop-ops`
- Email: `alem@alai.no`

Sentry Setup

Drop Sentry Setup

Last updated: 2026-02-20 **Source:** `src/drop-app/src/lib/sentry.ts`, `src/drop-app/src/lib/sentry-server.ts`, `src/drop-api/src/lib/sentry.ts`, `src/drop-app/.env.example`

Overview

Drop uses Sentry for error tracking and performance monitoring across three components:

1. **drop-app (client-side)** - Browser errors via `@sentry/browser`
2. **drop-app (server-side)** - Next.js middleware/API errors via custom envelope API
3. **drop-api** - Backend API errors via `@sentry/node`

All three components share the same DSN and gracefully degrade to console-only logging when Sentry is not configured.

Sentry Account Setup

1. Create Free Sentry Account

1. Visit sentry.io and sign up (free tier: 5,000 errors/month)
2. Confirm email and log in

2. Create Projects

Create **two separate projects** (one for app, one for API):

Project 1: drop-app

1. Click **Projects** → **Create Project**
2. Platform: **Next.js**
3. Project name: `drop-app`

4. Team: Default team (or create `drop-team`)
5. Alert frequency: **On every new issue**
6. Click **Create Project**
7. Copy the DSN (format: `https://examplePublicKey@o0.ingest.sentry.io/0`)

Project 2: drop-api

1. Repeat steps above with platform **Node.js**
2. Project name: `drop-api`
3. Copy the DSN (different from drop-app)

IMPORTANT: Use separate projects to keep frontend and backend errors isolated.

Environment Variables Configuration

drop-app (.env.local)

Add these variables to `src/drop-app/.env.local`:

```
# --- Sentry (Error Tracking) ---
# Client-side error tracking (browser)
NEXT_PUBLIC_SENTRY_DSN=https://YOUR_PUBLIC_KEY@o0.ingest.sentry.io/YOUR_PROJECT_ID

# Server-side error tracking (middleware/API routes)
# NOTE: drop-app server uses custom envelope API (no @sentry/nextjs due to Turbopack
incompatibility)
# Both client and server use the SAME DSN (NEXT_PUBLIC_SENTRY_DSN)

# Optional: Performance monitoring sample rate (0.0 to 1.0, default: 0.1 = 10%)
NEXT_PUBLIC_SENTRY_TRACES_SAMPLE_RATE=0.1

# Optional: For source map uploads (requires auth token from Sentry → Settings → Auth Tokens)
SENTRY_ORG=your-org-slug
SENTRY_PROJECT=drop-app
SENTRY_AUTH_TOKEN=your-auth-token
```

drop-api (.env)

Add these variables to `src/drop-api/.env`:

```
# --- Sentry (Error Tracking) ---
SENTRY_DSN=https://YOUR_PUBLIC_KEY@o0.ingest.sentry.io/YOUR_API_PROJECT_ID

# Optional: Performance monitoring sample rate (0.0 to 1.0, default: 0.1 = 10%)
SENTRY_TRACES_SAMPLE_RATE=0.1

# Optional: For source map uploads
SENTRY_ORG=your-org-slug
SENTRY_PROJECT=drop-api
SENTRY_AUTH_TOKEN=your-auth-token
```

Where to find these values:

- **DSN:** Project Settings → Client Keys (DSN)
- **Org slug:** Settings → Organization → General Settings → Organization Slug
- **Project name:** Project Settings → General → Project Name
- **Auth token:** Settings → Auth Tokens → Create New Token (scopes: `project:releases`, `project:write`)

Verification

Test Client-Side Error Capture (drop-app)

1. Start the app: `npm run dev` (in `src/drop-app/`)
2. Open browser console: `http://localhost:3000`
3. Trigger test error via console:
   ```
   throw new Error("Sentry test error - client-side");
   ```
4. Check Sentry dashboard: **Projects → drop-app → Issues**
5. You should see the test error appear within 10 seconds

Expected behavior:

- Error logged to browser console: `[Sentry] Error captured: Error: Sentry test error - client-side`
- Error appears in Sentry dashboard with stack trace, breadcrumbs, and browser context

Test Server-Side Error Capture (drop-app)

1. Create test API route: `src/drop-app/src/app/api/sentry-test/route.ts`

```
import { NextResponse } from 'next/server';
import { captureServerError } from '@/lib/sentry-server';

export async function GET() {
  try {
    throw new Error('Sentry test error - server-side');
  } catch (error) {
    captureServerError(error as Error, { tags: { test: 'true' } });
    return NextResponse.json({ error: 'Test error sent to Sentry' }, { status: 500
});
  }
}
```

2. Visit: `http://localhost:3000/api/sentry-test`
3. Check server console: `[Sentry Server] Error captured: Error: Sentry test error - server-side`
4. Check Sentry dashboard: **Projects → drop-app → Issues**

Test API Error Capture (drop-api)

1. Start the API: `npm run dev` (in `src/drop-api/`)
2. Trigger test error via curl:

```
curl http://localhost:4000/api/sentry-test
```

3. OR create test endpoint in `src/drop-api/src/routes/test.ts`:

```
import { Router } from 'express';
import { captureError } from '../lib/sentry.js';

const router = Router();

router.get('/sentry-test', (req, res) => {
  try {
    throw new Error('Sentry test error - API');
  } catch (error) {
    captureError(error as Error, { tags: { test: 'true' } });
    res.status(500).json({ error: 'Test error sent to Sentry' });
  }
});
```

```
export default router;
```

4. Check Sentry dashboard: **Projects → drop-api → Issues**

Source Map Upload Setup

Source maps allow Sentry to show readable stack traces instead of minified code.

1. Install Sentry CLI

```
# macOS (Homebrew)
brew install getsentry/tools/sentry-cli

# Or via npm (global)
npm install -g @sentry/cli
```

2. Configure Sentry CLI

Create `.sentryclirc` in project root:

```
[defaults]
url=https://sentry.io/
org=your-org-slug
project=drop-app

[auth]
token=your-auth-token
```

IMPORTANT: Add `.sentryclirc` to `.gitignore` (contains auth token).

3. Add Build Script (drop-app)

Update `src/drop-app/package.json`:

```
{
  "scripts": {
```

```
    "build": "next build",
    "build:sentry": "next build && sentry-cli sourcemaps upload --validate .next/static"
  }
}
```

4. Test Source Map Upload

```
cd src/drop-app
npm run build:sentry
```

Expected output:

```
> Analyzing source maps for sentry
> Uploading source maps to Sentry
✓ Successfully uploaded source maps
```

5. CI/CD Integration

For automated uploads in CI/CD, add these secrets to your deployment platform:

Vercel/Railway/Fly.io:

- `SENTRY_ORG`
- `SENTRY_PROJECT`
- `SENTRY_AUTH_TOKEN`

Then update build command:

```
npm run build && sentry-cli sourcemaps upload --validate .next/static
```

Alert Rules Configuration

Recommended Alert Rules

1. New Issue Alert (drop-app)

1. Go to **Projects → drop-app → Settings → Alerts**
2. Click **Create Alert Rule**
3. Configure:

- **Conditions:** When a new issue is created
- **Filters:** Environment = production
- **Actions:**
  - Send notification to: Slack channel `#drop-alerts`
  - Send email to: alem@alai.no

4. Save rule

2. High Error Rate Alert (drop-app)

1. Create new alert rule
2. Configure:
   - **Conditions:** Number of events in an issue is more than 100 in 1 hour
   - **Filters:** Environment = production, Level = error
   - **Actions:**
     - Send notification to: Slack channel `#drop-alerts`
     - Send email to: alem@alai.no
3. Save rule

3. Critical Error Alert (drop-api)

1. Go to **Projects → drop-api → Settings → Alerts**
2. Create alert rule:
   - **Conditions:** When a new issue is created AND Level = fatal
   - **Filters:** Environment = production
   - **Actions:**
     - Send notification to: Slack channel `#drop-critical`
     - Send email to: alem@alai.no
3. Save rule

4. Performance Degradation Alert (drop-app)

1. Create alert rule:
   - **Conditions:** Average transaction duration is above 2000ms for 5 minutes
   - **Filters:** Environment = production, Transaction = /api/transactions/*
   - **Actions:**
     - Send notification to: Slack channel `#drop-performance`
2. Save rule

Slack Integration (Optional)

1. Go to **Settings → Integrations → Slack**
2. Click **Add Workspace**
3. Authorize Sentry to access your Slack workspace
4. Select channels: `#drop-alerts`, `#drop-critical`, `#drop-performance`
5. Test integration by triggering a test error

PII Scrubbing

All three Sentry integrations automatically scrub sensitive data before sending events:

Scrubbed fields:

- `password`
- `pin`
- `cardNumber`
- `cvv`
- `fødselsnummer`
- `authorization` headers
- `cookie` headers

Implementation:

- **drop-app (client):** `src/drop-app/src/lib/sentry.ts` (lines 51-76)
- **drop-app (server):** Custom envelope API (no PII in server-side events)
- **drop-api:** `src/drop-api/src/lib/sentry.ts` (lines 48-139)

Verification:

1. Trigger error with sensitive data:

```
try {
  throw new Error('Login failed for user with password=secret123');
} catch (error) {
  captureError(error, { extra: { cardNumber: '1234567890123456' } });
}
```

2. Check Sentry event:
   - Message should show: `Login failed for user with password=[REDACTED]`
   - Extra context should show: `cardNumber: [REDACTED]`

Environment-Specific Configuration

Development

- **DSN:** Optional (errors log to console only if not set)
- **Sample rate:** 1.0 (capture all errors for debugging)

- **Source maps:** Not required (local stack traces are readable)

```
# .env.local (development)
NEXT_PUBLIC_SENTRY_DSN=  # Leave empty to disable Sentry in dev
```

Staging

- **DSN:** Required (test Sentry integration before production)
- **Sample rate:** 0.5 (capture 50% of transactions)
- **Source maps:** Enabled (verify uploads work)

```
# .env.staging
NEXT_PUBLIC_SENTRY_DSN=https://YOUR_KEY@sentry.io/YOUR_PROJECT_ID
NEXT_PUBLIC_SENTRY_TRACES_SAMPLE_RATE=0.5
SENTRY_AUTH_TOKEN=your-auth-token
```

Production

- **DSN:** Required (critical for production monitoring)
- **Sample rate:** 0.1 (capture 10% of transactions to stay within free tier)
- **Source maps:** Enabled (required for readable stack traces)

```
# .env.production
NEXT_PUBLIC_SENTRY_DSN=https://YOUR_KEY@sentry.io/YOUR_PROJECT_ID
NEXT_PUBLIC_SENTRY_TRACES_SAMPLE_RATE=0.1
SENTRY_AUTH_TOKEN=your-auth-token
```

Troubleshooting

No errors appearing in Sentry dashboard

Check 1: DSN configured?

```
# drop-app
echo $NEXT_PUBLIC_SENTRY_DSN

# drop-api
```

```
echo $SENTRY_DSN
```

Check 2: Console output?

- Errors should ALWAYS log to console, even if Sentry upload fails
- Look for: `[Sentry] Error captured: ...`

Check 3: Network errors?

- Open browser DevTools → Network tab
- Filter by `sentry.io`
- Check for failed requests (should see POST to `https://o0.ingest.sentry.io/api/.../envelope/`)

Check 4: Environment mismatch?

- Sentry filters events by environment (`production`, `development`, `staging`)
- Verify `NEXT_PUBLIC_APP_ENV` or `NODE_ENV` matches your Sentry project filters

Source maps not working (minified stack traces)

Check 1: Source maps uploaded?

```
cd src/drop-app
sentry-cli releases list
```

Check 2: Release version matches?

- Sentry matches source maps by release version
- Verify `package.json` version matches uploaded release

Check 3: Upload command ran?

```
# Manually test upload
sentry-cli sourcemaps upload --validate .next/static
```

PII still appearing in events

Check 1: Verify beforeSend hook

- Inspect `src/lib/sentry.ts` (client) or `src/lib/sentry.ts` (API)
- Confirm `beforeSend` function is scrubbing sensitive keys

Check 2: Add custom scrubbing

- If new sensitive fields appear, add them to scrubbing list:

```
const sensitiveKeys = ["password", "pin", "yourNewField"];
```

Cost Management

Sentry Free Tier:

- 5,000 errors per month
- 10,000 performance units per month
- 1 GB attachments
- 30 days retention

Staying within free tier:

1. **Lower sample rate:** Set `SENTRY_TRACES_SAMPLE_RATE=0.1` (10%)
2. **Filter noisy errors:** Use `beforeSend` to ignore expected errors (e.g., 404s)
3. **Set up quotas:** Sentry → Settings → Quotas → Set monthly limits

Example: Ignore 404 errors

```
beforeSend(event, hint) {
  // Ignore 404 errors
  if (event.request?.url?.includes('/api/') &&
hint?.originalException?.message?.includes('404')) {
    return null; // Don't send to Sentry
  }
  return event;
}
```

Security Considerations

1. **Auth token storage:**
   - NEVER commit `.sentryclirc` to git
   - Store `SENTRY_AUTH_TOKEN` in CI/CD secrets, not `.env` files
2. **DSN exposure:**
   - `NEXT_PUBLIC_SENTRY_DSN` is exposed to client-side code (safe - it's public)
   - Sentry rate-limits abuse via DSN quotas
3. **PII scrubbing:**

- Always verify PII scrubbing works before deploying to production
- Test with real-world data patterns (Norwegian fødselsnummer, BankID tokens)

4. **Access control:**
   - Limit Sentry dashboard access to authorized team members only
   - Use Sentry Teams to restrict project access

References

- **Sentry Docs:** https://docs.sentry.io/platforms/javascript/guides/nextjs/
- **Sentry CLI:** https://docs.sentry.io/product/cli/
- **Source Maps:** https://docs.sentry.io/platforms/javascript/sourcemaps/
- **PII Scrubbing:** https://docs.sentry.io/platforms/javascript/data-management/sensitive-data/
- **Alert Rules:** https://docs.sentry.io/product/alerts/

Next Steps

1. Create Sentry account and projects (drop-app, drop-api)
2. Add DSN to `.env.local` (development) and `.env.production` (production)
3. Test error capture in all three components
4. Configure alert rules (new issues, high error rate, critical errors)
5. Set up source map uploads for production builds
6. Integrate Slack notifications (optional)
7. Monitor error dashboard daily during initial deployment

CloudWatch Logs Setup

CloudWatch Logs Setup — Drop Production

Date: 2026-02-22 **Priority:** P0 (Production Blocker) **Effort:** 2 hours **Cost:** ~$5/month (30 GB ingestion)

Overview

AWS App Runner automatically streams application logs (stdout/stderr) to CloudWatch Logs. This setup guide configures **retention policies**, **log insights queries**, and **alarms** for production monitoring.

Prerequisites

- AWS CLI configured with credentials
- App Runner service deployed to `eu-west-1`
- Application writes JSON logs to stdout (already implemented via `src/lib/logger.ts`)

Configuration

1. Set Log Retention Policy

Default: CloudWatch Logs retain forever (expensive) **Recommendation:** 30 days (production), 7 days (staging)

```
# Production: 30 days retention
aws logs put-retention-policy \
  --log-group-name /aws/apprunner/drop-production \
```

```
  --retention-in-days 30 \
  --region eu-west-1

# Staging: 7 days retention
aws logs put-retention-policy \
  --log-group-name /aws/apprunner/drop-staging \
  --retention-in-days 7 \
  --region eu-west-1
```

Verify retention:

```
aws logs describe-log-groups \
  --log-group-name-prefix /aws/apprunner/drop \
  --region eu-west-1 \
  | jq '.logGroups[] | {name: .logGroupName, retention: .retentionInDays}'

# Expected:
# {
#   "name": "/aws/apprunner/drop-production",
#   "retention": 30
# }
```

---

2. Create Log Insights Queries

Purpose: Pre-built queries for common investigations.

Query 1: All Errors (Last Hour)

```
fields @timestamp, level, message, metadata.error, metadata.userId, requestId
| filter level = "error"
| sort @timestamp desc
| limit 100
```

Save as: `drop-errors-last-hour`

Query 2: User Activity Trace

```
fields @timestamp, level, message, metadata.userId, metadata.action, requestId
| filter metadata.userId = "usr_123"
| sort @timestamp desc
```

```
| limit 500
```

Save as: `drop-user-activity-trace`

Query 3: Request Trace by ID

```
fields @timestamp, level, message, metadata
| filter requestId = "req_abc123"
| sort @timestamp asc
```

Save as: `drop-request-trace`

Query 4: API Endpoint Performance

```
fields @timestamp, message, metadata.endpoint, metadata.latencyMs
| filter metadata.latencyMs > 1000
| stats avg(metadata.latencyMs) as avg_latency, max(metadata.latencyMs) as max_latency,
count() as slow_requests by metadata.endpoint
| sort slow_requests desc
```

Save as: `drop-slow-endpoints`

Query 5: Authentication Events

```
fields @timestamp, level, message, metadata.action, metadata.userId, metadata.ip
| filter metadata.action in ["login_success", "login_failure", "logout"]
| sort @timestamp desc
| limit 100
```

Save as: `drop-auth-events`

Query 6: Payment Failures

```
fields @timestamp, level, message, metadata.errorCode, metadata.transactionId, metadata.userId
| filter metadata.errorCode in ["INSUFFICIENT_FUNDS", "PAYMENT_REJECTED", "TIMEOUT"]
| sort @timestamp desc
| limit 50
```

Save as: `drop-payment-failures`

---

3. Create CloudWatch Alarms

Alarm 1: High Error Rate

Metric: Error log entries per minute **Threshold:** >10 errors/minute for 2 consecutive periods
Action: Send SNS notification → Slack webhook

```
# Create metric filter
aws logs put-metric-filter \
  --log-group-name /aws/apprunner/drop-production \
  --filter-name drop-error-count \
  --filter-pattern '{ $.level = "error" }' \
  --metric-transformations \
    metricName=ErrorCount,metricNamespace=Drop/Logs,metricValue=1,unit=Count \
  --region eu-west-1

# Create alarm
aws cloudwatch put-metric-alarm \
  --alarm-name drop-high-error-rate \
  --alarm-description "Alert when error rate exceeds threshold" \
  --metric-name ErrorCount \
  --namespace Drop/Logs \
  --statistic Sum \
  --period 60 \
  --evaluation-periods 2 \
  --threshold 10 \
  --comparison-operator GreaterThanThreshold \
  --treat-missing-data notBreaching \
  --alarm-actions <SNS-TOPIC-ARN> \
  --region eu-west-1
```

Alarm 2: No Logs Received (Service Down)

Metric: Log ingestion stopped **Threshold:** No logs for 5 minutes **Action:** Send SNS notification

```
aws cloudwatch put-metric-alarm \
  --alarm-name drop-no-logs-received \
  --alarm-description "Alert when no logs received (service may be down)" \
  --metric-name IncomingLogEvents \
  --namespace AWS/Logs \
  --dimensions Name=LogGroupName,Value=/aws/apprunner/drop-production \
  --statistic Sum \
  --period 300 \
```

```
  --evaluation-periods 1 \
  --threshold 1 \
  --comparison-operator LessThanThreshold \
  --treat-missing-data breaching \
  --alarm-actions <SNS-TOPIC-ARN> \
  --region eu-west-1
```

Alarm 3: Database Errors

Metric: Database connection errors **Threshold:** >5 DB errors in 5 minutes

```
aws logs put-metric-filter \
  --log-group-name /aws/apprunner/drop-production \
  --filter-name drop-db-errors \
  --filter-pattern '{ $.message = "*database*" && $.level = "error" }' \
  --metric-transformations \
    metricName=DatabaseErrors,metricNamespace=Drop/Logs,metricValue=1,unit=Count \
  --region eu-west-1

aws cloudwatch put-metric-alarm \
  --alarm-name drop-database-errors \
  --metric-name DatabaseErrors \
  --namespace Drop/Logs \
  --statistic Sum \
  --period 300 \
  --evaluation-periods 1 \
  --threshold 5 \
  --comparison-operator GreaterThanThreshold \
  --alarm-actions <SNS-TOPIC-ARN> \
  --region eu-west-1
```

---

4. SNS Topic for Alerts

Create SNS topic (if not exists):

```
aws sns create-topic \
  --name drop-cloudwatch-alerts \
  --region eu-west-1
```

```
# Output:
# {
#   "TopicArn": "arn:aws:sns:eu-west-1:324480209768:drop-cloudwatch-alerts"
# }
```

Subscribe Slack webhook:

```
# Option 1: Email subscription (immediate)
aws sns subscribe \
  --topic-arn arn:aws:sns:eu-west-1:324480209768:drop-cloudwatch-alerts \
  --protocol email \
  --notification-endpoint alem@alai.no \
  --region eu-west-1

# Confirm subscription via email link

# Option 2: Lambda → Slack (requires Lambda function)
# See: infrastructure/cloudwatch-to-slack-lambda.md (future enhancement)
```

---

5. Export Logs to S3 (Compliance/Archival)

Purpose: Long-term storage (>30 days) for compliance, cheaper than CloudWatch.

Create S3 bucket:

```
aws s3 mb s3://drop-logs-archive --region eu-west-1

# Set lifecycle policy (move to Glacier after 90 days)
cat > lifecycle.json <<EOF
{
  "Rules": [
    {
      "Id": "archive-old-logs",
      "Status": "Enabled",
      "Transitions": [
        {
          "Days": 90,
          "StorageClass": "GLACIER"
        }
```

```
      ],
      "Expiration": {
        "Days": 1825
      }
    }
  ]
}
EOF

aws s3api put-bucket-lifecycle-configuration \
  --bucket drop-logs-archive \
  --lifecycle-configuration file://lifecycle.json
```

Create export task (manual, run monthly):

```
# Export last 30 days to S3 (run on day 1 of each month)
START_TIME=$(date -u -d '60 days ago' +%s)000
END_TIME=$(date -u -d '30 days ago' +%s)000

aws logs create-export-task \
  --log-group-name /aws/apprunner/drop-production \
  --from $START_TIME \
  --to $END_TIME \
  --destination drop-logs-archive \
  --destination-prefix logs/$(date +%Y-%m) \
  --region eu-west-1

# Check export status
aws logs describe-export-tasks --region eu-west-1
```

Automate with Lambda (future):

- Schedule Lambda to run monthly
- Export previous month's logs to S3
- Delete from CloudWatch after successful export

Log Format

Current Format (Structured JSON)

Example log entry:

```
{
  "timestamp": "2026-02-22T10:30:45.123Z",
  "level": "info",
  "message": "User logged in",
  "requestId": "req_abc123",
  "metadata": {
    "userId": "usr_456",
    "email": "user@example.com",
    "ip": "1.2.3.4",
    "action": "login_success"
  }
}
```

CloudWatch Logs Insights automatically parses JSON fields, enabling queries like:

```
| filter metadata.userId = "usr_456"
```

Cost Estimate

CloudWatch Logs Pricing (EU-West-1)

- **Ingestion:** $0.50 per GB
- **Storage:** $0.03 per GB/month
- **Log Insights queries:** $0.005 per GB scanned

Expected Usage (Production)

- **Log volume:** ~1 GB/day (30 GB/month)
- **Ingestion cost:** 30 GB × $0.50 = $15/month
- **Storage cost (30-day retention):** 30 GB × $0.03 = $0.90/month
- **Query cost:** ~10 queries/day × 1 GB × $0.005 × 30 = $1.50/month

Total: ~$17/month

Cost Optimization

1. **Reduce log verbosity** (filter debug logs in production):

```
// src/lib/logger.ts
const minLevel = process.env.NODE_ENV === 'production' ? 'info' : 'debug';
```

2. **Use sampling for high-volume events**:

```
if (Math.random() < 0.1) { // Log 10% of requests
  logger.debug('Request details', { ... });
}
```

3. **Export to S3 for long-term storage** ($0.023/GB/month, 23% cheaper)

Querying Logs

Via AWS Console

1. Open CloudWatch Console: https://console.aws.amazon.com/cloudwatch/
2. Navigate to: Logs → Log groups → `/aws/apprunner/drop-production`
3. Click "Search log group" or "Insights queries"
4. Select saved query or write custom query

Via AWS CLI

```
# Run saved query
aws logs start-query \
  --log-group-name /aws/apprunner/drop-production \
  --start-time $(date -u -d '1 hour ago' +%s) \
  --end-time $(date -u +%s) \
  --query-string 'fields @timestamp, level, message | filter level = "error" | sort @timestamp
desc' \
  --region eu-west-1

# Get query results (use queryId from previous command)
aws logs get-query-results --query-id <query-id> --region eu-west-1
```

Via Log Streaming (Real-Time)

```
# Stream logs in real-time (like tail -f)
aws logs tail /aws/apprunner/drop-production \
  --follow \
  --format short \
  --region eu-west-1

# Filter by error level
aws logs tail /aws/apprunner/drop-production \
  --follow \
  --filter-pattern '{ $.level = "error" }' \
  --region eu-west-1
```

Troubleshooting

Issue: No logs appearing in CloudWatch

Diagnosis:

```
# Check if log group exists
aws logs describe-log-groups \
  --log-group-name-prefix /aws/apprunner/drop \
  --region eu-west-1

# Check App Runner service logs integration
aws apprunner describe-service \
  --service-arn <ARN> \
  --region eu-west-1 \
  | jq '.Service.ObservabilityConfiguration'
```

Solution:

- App Runner auto-creates log group on first log output
- Verify app is writing to stdout (not file)
- Check IAM permissions (App Runner role needs `logs:CreateLogStream`, `logs:PutLogEvents`)

Issue: Logs not in JSON format

Diagnosis:

```
# Check log entries
aws logs tail /aws/apprunner/drop-production --format short --region eu-west-1 | head -10
```

Solution:

- Ensure app uses `logger.ts` for all logging (not `console.log`)
- Verify `process.stdout.write(JSON.stringify(entry) + "\n")` is used

Checklist

- ☐ Retention policy set (30 days production, 7 days staging)
- ☐ Log Insights queries saved (6 queries)
- ☐ Metric filters created (error count, DB errors)
- ☐ CloudWatch alarms configured (3 alarms)
- ☐ SNS topic created and subscribed (email/Slack)
- ☐ S3 export bucket created (with lifecycle policy)
- ☐ Cost estimate reviewed and approved
- ☐ Team trained on log querying (AWS Console + CLI)
- ☐ Documentation updated

Next Steps

1. **Deploy retention policies** (run commands above)
2. **Test alarms** (trigger error spike, verify alert received)
3. **Save Log Insights queries** (via AWS Console)
4. **Schedule monthly S3 export** (manual for now, automate later)
5. **Monitor costs** (set billing alert at $20/month)

Related Documentation

- `docs/infrastructure/MONITORING.md` — Overall monitoring setup
- `src/lib/logger.ts` — Structured logging implementation
- `infrastructure/error-tracking-setup.md` — Sentry integration
- AWS CloudWatch Logs docs: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/

Last Updated: 2026-02-22 **Owner:** John (AI Director)