Monitoring Guide - PRO PDF

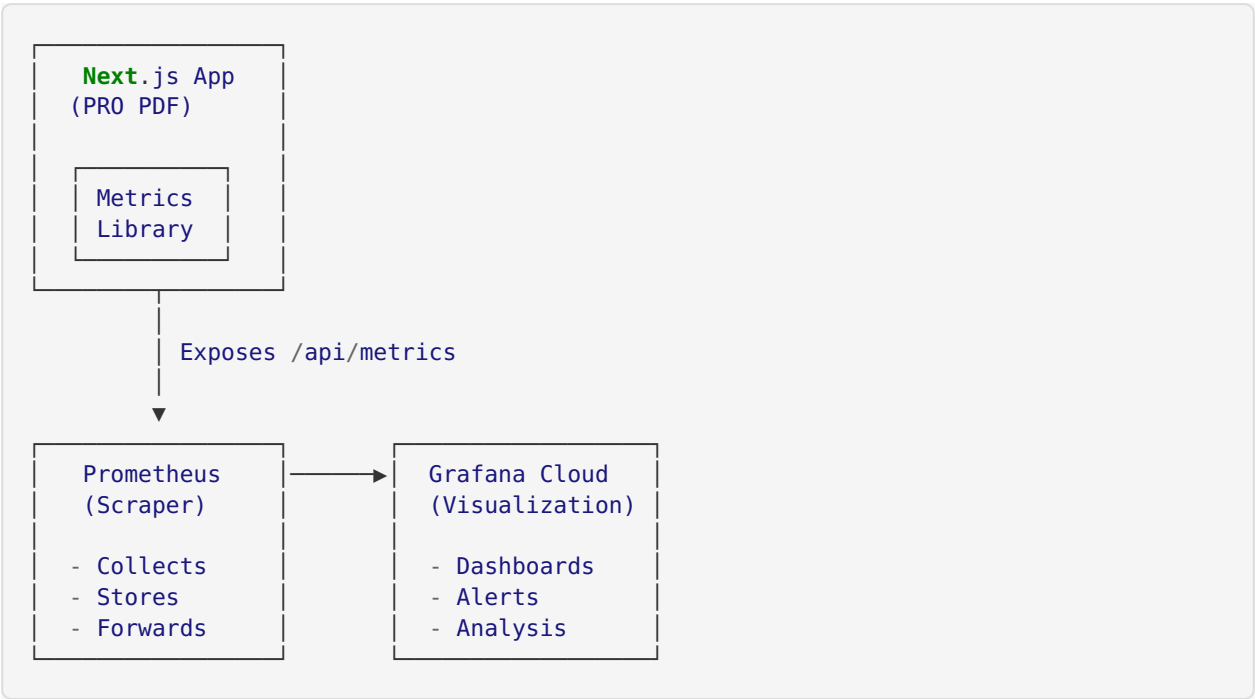
Overview

PRO PDF uses a comprehensive monitoring stack with **Prometheus** for metrics collection and **Grafana Cloud** for visualization and alerting. This guide covers setup, configuration, and usage of the monitoring system.

Table of Contents

- Architecture
- Quick Start
- Grafana Cloud Setup
- Local Prometheus Setup
- Available Metrics
- Pre-built Dashboards
- Adding Custom Metrics
- Alerting
- Troubleshooting

Architecture



Components:

1. **Metrics Library** (`lib/monitoring/metrics.ts`)
   - Collects application metrics
   - Exposes Prometheus-format metrics

2. **Prometheus**
   - Scrapes `/api/metrics` endpoint
   - Stores time-series data
   - Forwards to Grafana Cloud

3. **Grafana Cloud**
   - Visualizes metrics in dashboards
   - Triggers alerts
   - Provides long-term storage

Quick Start

1. Check Metrics Endpoint

```
# Start your Next.js app
yarn dev

# Visit the metrics endpoint
curl http://localhost:3000/api/metrics
```

You should see Prometheus-format metrics:

```
# HELP propdf_http_requests_total Total number of HTTP requests
# TYPE propdf_http_requests_total counter
propdf_http_requests_total{method="GET",route="/",status_code="200"} 42

# HELP propdf_http_request_duration_seconds Duration of HTTP requests in seconds
# TYPE propdf_http_request_duration_seconds histogram
prop-
df_http_request_duration_seconds_bucket{le="0.1",method="GET",route="/",status_code="2
00"} 40
...
```

2. Check Health Endpoint

```
curl http://localhost:3000/api/metrics/health
```

Response:

```
{
  "status": "healthy",
  "timestamp": "2025-12-01T10:00:00.000Z",
  "checks": {
    "database": "healthy",
    "app": "healthy"
  }
}
```

Grafana Cloud Setup

Step 1: Create Grafana Cloud Account

1. Go to https://grafana.com/auth/sign-up

2. Sign up for a free account
3. Create a stack (e.g., “pro-pdf-production”)

Step 2: Get Prometheus Credentials

1. In Grafana Cloud, go to **My Account → Stacks**
2. Click on your stack
3. Click **Details → Prometheus**
4. Note down:
   - **Remote Write URL** (e.g., `https://prometheus-prod-01-us-west-0.grafana.net/api/prom/push`)
   - **Username** (e.g., `123456`)
   - **API Key** (generate a new one with “MetricsPublisher” role)

Step 3: Configure Environment Variables

Add to your `.env` file:

```
# Grafana Cloud Configuration
GRAFANA_CLOUD_PROMETHEUS_URL=https://prometheus-prod-01-us-west-0.grafana.net/api/
prom/push
GRAFANA_CLOUD_PROMETHEUS_USER=123456
GRAFANA_CLOUD_API_KEY=your_api_key_here
```

Step 4: Update Prometheus Configuration

The `monitoring/prometheus/prometheus.yml` file uses these environment variables:

```
remote_write:
  - url: ${GRAFANA_CLOUD_PROMETHEUS_URL}
    basic_auth:
      username: ${GRAFANA_CLOUD_PROMETHEUS_USER}
      password: ${GRAFANA_CLOUD_API_KEY}
```

Step 5: Import Dashboards

1. In Grafana Cloud, go to **Dashboards → Import**
2. Upload the JSON files from `monitoring/grafana/dashboards/`:
   - `application-overview.json`
   - `pdf-operations.json`
   - `business-metrics.json`
3. Select your Prometheus datasource
4. Click **Import**

Local Prometheus Setup

Using Docker Compose (Recommended)

1. Navigate to the Prometheus directory:
   `bash`
   `cd monitoring/prometheus`
2. Set environment variables:
   `bash`
   `export GRAFANA_CLOUD_PROMETHEUS_URL="your_url"`

```
   export GRAFANA_CLOUD_PROMETHEUS_USER="your_user"
   export GRAFANA_CLOUD_API_KEY="your_key"
```

3. Start Prometheus:
   ```
   bash
   docker-compose up -d
   ```

4. Verify Prometheus is running:
   ```bash
   # Prometheus UI
   open http://localhost:9090

# Check targets
open http://localhost:9090/targets
```

1. You should see:
   - `pro-pdf-app` target (scraping `/api/metrics`)
   - `pro-pdf-health` target (scraping `/api/metrics/health`)
   - Both targets should be **UP** (green)

Manual Installation

If you prefer to run Prometheus directly:

1. Download Prometheus:
   ```
   bash
   wget https://github.com/prometheus/prometheus/releases/download/v2.45.0/prometheus-2.45.0.linux-amd64.tar.gz
   tar xvfz prometheus-2.45.0.linux-amd64.tar.gz
   cd prometheus-2.45.0.linux-amd64
   ```

2. Copy configuration:
   ```
   bash
   cp /path/to/monitoring/prometheus/prometheus.yml .
   ```

3. Run Prometheus:
   ```
   bash
   ./prometheus --config.file=prometheus.yml
   ```

Available Metrics

HTTP Request Metrics

Metric	Type	Description	Labels
`propdf_http_requests_total`	Counter	Total HTTP requests	`method`, `route`, `status_code`
`propdf_http_request_duration_seconds`	Histogram	Request duration	`method`, `route`, `status_code`
`propdf_http_request_size_bytes`	Histogram	Request body size	`method`, `route`
`propdf_http_response_size_bytes`	Histogram	Response body size	`method`, `route`

Example Queries:

```
# Request rate by route
sum(rate(propdf_http_requests_total[5m])) by (route)

# P95 response time
histogram_quantile(0.95, sum(rate(propdf_http_request_duration_seconds_bucket[5m]))
by (le))

# Error rate
sum(rate(propdf_http_requests_total{status_code=~"5.."}[5m]))
```

PDF Processing Metrics

Metric	Type	Description	Labels
`propdf_conversions_total`	Counter	Total PDF operations	`operation`, `status`
`propdf_conversion_duration_seconds`	Histogram	Processing duration	`operation`, `status`
`propdf_file_size_bytes`	Histogram	PDF file sizes	`operation`
`propdf_page_count`	Histogram	PDF page counts	`operation`
`propdf_active_jobs`	Gauge	Active processing jobs	`operation`

Example Queries:

```
# Conversion rate by operation
sum(rate(propdf_conversions_total[5m])) by (operation)

# Average conversion time
avg(rate(propdf_conversion_duration_seconds_sum[5m]) / rate(propdf_conversion_duration_seconds_count[5m]))

# Success rate
sum(rate(propdf_conversions_total{status="success"}[5m])) / sum(rate(propdf_conversions_total[5m]))
```

Authentication Metrics

Metric	Type	Description	Labels
`propdf_login_attempts_total`	Counter	Login attempts	`status`
`propdf_signup_attempts_total`	Counter	Signup attempts	`status`
`propdf_2fa_verifications_total`	Counter	2FA verifications	`status`
`propdf_active_sessions`	Gauge	Active user sessions	-

Example Queries:

```
# Login success rate
sum(rate(propdf_login_attempts_total{status="success"}[5m])) / sum(rate(propdf_login_attempts_total[5m]))

# Failed login attempts
sum(rate(propdf_login_attempts_total{status="failure"}[5m]))
```

Database Metrics

Metric	Type	Description	Labels
`propdf_db_queries_total`	Counter	Database queries	`operation`, `table`, `status`
`propdf_db_query_duration_seconds`	Histogram	Query duration	`operation`, `table`
`propdf_db_connections`	Gauge	Database connections	`state`

Business Metrics

Metric	Type	Description	Labels
`propdf_total_users`	Gauge	Total registered users	`subscription_type`
`propdf_subscription_events_total`	Counter	Subscription events	`event_type`
`propdf_api_usage_by_tier_total`	Counter	API usage by tier	`tier`, `operation`

Error Metrics

Metric	Type	Description	Labels
`propdf_errors_total`	Counter	Application errors	`type`, `severity`
`propdf_rate_limit_hits_total`	Counter	Rate limit hits	`endpoint`

System Metrics (Node.js)

Automatically collected by `prom-client`:

- `propdf_nodejs_heap_size_total_bytes`
- `propdf_nodejs_heap_size_used_bytes`
- `propdf_nodejs_external_memory_bytes`
- `propdf_nodejs_gc_duration_seconds`
- `propdf_nodejs_eventloop_lag_seconds`
- `propdf_nodejs_version_info`

Pre-built Dashboards

1. Application Overview

File: `monitoring/grafana/dashboards/application-overview.json`

Panels:
- Request Rate (gauge)
- Request Duration by Route (time series)
- HTTP Status Codes (stacked area)
- Memory Usage (time series)

Use Case: High-level application health and performance

2. PDF Operations

File: `monitoring/grafana/dashboards/pdf-operations.json`

Panels:
- PDF Operations Rate (stacked area)
- Conversion Duration p95 (time series)
- Active Jobs (gauge)
- Conversions Last Hour (stacked area)
- Average File Size p95 (time series)

Use Case: Monitor PDF processing performance and volume

3. Business Metrics

File: `monitoring/grafana/dashboards/business-metrics.json`

Panels:
- Total Users (stat)
- Active Sessions (stat)
- Login Attempts (stacked area)
- Signup Attempts (stacked area)
- Subscription Events (stacked area)
- Users by Subscription Type (pie chart)
- API Usage by Tier (time series)

Use Case: Business KPIs and user engagement

Adding Custom Metrics

Step 1: Define the Metric

Edit `lib/monitoring/metrics.ts`:

```
import { Counter } from 'prom-client';

export const myCustomMetric = new Counter({
  name: 'propdf_my_custom_metric_total',
  help: 'Description of my custom metric',
  labelNames: ['label1', 'label2'],
  registers: [register],
});
```

Step 2: Instrument Your Code

```
import { myCustomMetric } from '@/lib/monitoring/metrics';

// Increment counter
myCustomMetric.inc({ label1: 'value1', label2: 'value2' });

// Increment by specific amount
myCustomMetric.inc({ label1: 'value1', label2: 'value2' }, 5);
```

Step 3: Query in Grafana

```
sum(rate(propdf_my_custom_metric_total[5m])) by (label1)
```

Alerting

Grafana Cloud Alerts

1. In Grafana Cloud, go to **Alerting** → **Alert rules**
2. Click **New alert rule**
3. Configure alert:

Example: High Error Rate Alert

```
Name: High Error Rate
Query:
  A: sum(rate(propdf_http_requests_total{status_code=~"5.."}[5m]))
Condition:
  WHEN A IS ABOVE 10
For: 5m
Annotations:
  summary: High error rate detected
  description: Error rate is {{ $value }} errors/sec
Notifications:
  - Email: ops@propdf.com
  - Slack: #alerts
```

Common Alert Rules

1. High Response Time

```
histogram_quantile(0.95,
  sum(rate(propdf_http_request_duration_seconds_bucket[5m])) by (le)
) > 2
```

2. Low Success Rate

```
sum(rate(propdf_conversions_total{status="success"}[5m]))
/
sum(rate(propdf_conversions_total[5m])) < 0.95
```

3. Database Issues

```
sum(rate(propdf_db_queries_total{status="error"}[5m])) > 1
```

4. High Memory Usage

```
propdf_nodejs_heap_size_used_bytes
/
propdf_nodejs_heap_size_total_bytes > 0.9
```

Troubleshooting

Metrics Endpoint Returns 500

Check:

1. Application logs for errors

2. Prometheus client library is installed
3. Metrics registry is properly initialized

```
# Check logs
yarn dev
# Look for metrics-related errors
```

Prometheus Can’t Scrape Metrics

Check:

1. Application is running on correct port
2. Firewall allows connections
3. Prometheus configuration has correct target

```
# Test connectivity
curl http://localhost:3000/api/metrics

# Check Prometheus targets
open http://localhost:9090/targets
```

Dashboards Show No Data

Check:

1. Prometheus is receiving metrics
2. Grafana datasource is configured correctly
3. Time range is appropriate
4. Queries are correct

```
# Test query in Grafana Explore
sum(propdf_http_requests_total)
```

High Cardinality Warning

Issue: Too many unique label combinations

Solution:

- Avoid dynamic labels (e.g., user IDs, timestamps)
- Use label values from a fixed set
- Aggregate before labeling

```
// ❌ Bad: High cardinality
myMetric.inc({ userId: req.userId }); // Unique per user!

// ✅ Good: Low cardinality
myMetric.inc({ userType: req.user.subscriptionType }); // Fixed set: free, premium
```

Memory Leak in Metrics

Issue: Metrics consuming too much memory

Solution:

1. Use histograms instead of individual metrics
2. Set appropriate bucket sizes

3. Limit label cardinality
4. Reset metrics periodically if needed

Best Practices

1. Naming Conventions

- Use `propdf_` prefix for all custom metrics
- Use snake_case for metric names
- Suffix with unit: `_seconds`, `_bytes`, `_total`
- Use descriptive names: `http_request_duration` not `req_time`

2. Label Guidelines

- Keep labels to 5-10 per metric
- Use consistent label names across metrics
- Avoid high-cardinality labels (user IDs, timestamps)
- Document label meanings

3. Metric Types

- **Counter:** Things that only increase (requests, errors)
- **Gauge:** Things that go up and down (connections, memory)
- **Histogram:** Distribution of values (duration, size)
- **Summary:** Like histogram but with quantiles

4. Performance

- Don’t instrument every line of code
- Focus on business-critical paths
- Use sampling for high-frequency events
- Batch metric updates when possible

5. Security

- Protect `/api/metrics` endpoint in production
- Use authentication for Prometheus scraping
- Don’t expose sensitive data in labels
- Sanitize user input before labeling

Resources

- **Prometheus Documentation:** https://prometheus.io/docs/
- **Grafana Documentation:** https://grafana.com/docs/
- **prom-client (Node.js):** https://github.com/siimon/prom-client
- **PromQL Tutorial:** https://prometheus.io/docs/prometheus/latest/querying/basics/
- **Grafana Dashboards:** https://grafana.com/grafana/dashboards/

Support

For monitoring issues:

1. Check this guide

2. Review Prometheus logs: `docker logs pro-pdf-prometheus`
3. Check Grafana Cloud status: https://status.grafana.com/
4. Open an issue in the repository

Last Updated: December 2025
Version: 1.0.0