# Current Monitoring and Analytics Setup for Setarcos App

This document outlines the existing monitoring and analytics infrastructure implemented in the Setarcos philosophy application, based on the technical blueprint and technology stack documentation.

## 1. Analytics Infrastructure

### 1.1 PostHog Analytics Platform

**Implementation Status**: Fully implemented **Repository Location**: setarcos-app (frontend) and setarcos-api (backend)

PostHog serves as the primary analytics platform for the Setarcos app, providing comprehensive user behavior tracking and analytics capabilities:

#### Key Features Implemented:

* **Event Tracking**: Capturing user interactions across all app features
* **User Identification**: Associating events with specific user profiles
* **Funnel Analysis**: Tracking user journeys through key app flows
* **Retention Analysis**: Measuring user engagement over time
* **Dashboard Capabilities**: Visualizing key metrics and trends

#### Integration Points:

* **Frontend**: React Native integration via PostHog client
* **Backend**: API-based event tracking for server-side events
* **Cross-Platform**: Consistent event naming and properties across platforms

#### Key Metrics Tracked:

* **Feature Usage**: Engagement with Ask, Quest, Explore, Journal, and Forum features
* **User Progression**: XP accumulation, level progression, and badge achievements
* **Content Engagement**: Time spent with AI responses, concept exploration depth
* **Retention Metrics**: Daily and weekly active users, session frequency and duration
* **Conversion Metrics**: Free-to-paid conversion rates, feature unlock patterns

#### Implementation Details:

// Frontend implementation (React Native)

import PostHog from 'posthog-react-native';

// Initialize in App.tsx

PostHog.setup('phc\_YOUR\_API\_KEY', {

host: 'https://app.posthog.com',

captureApplicationLifecycleEvents: true,

captureDeepLinks: true,

recordScreenViews: true,

});

// Track events

function trackEvent(eventName, properties) {

PostHog.capture(eventName, properties);

}

// Identify users

function identifyUser(userId, userProperties) {

PostHog.identify(userId, userProperties);

}

# Backend implementation (FastAPI)

from posthog import Posthog

posthog = Posthog(project\_api\_key='phc\_YOUR\_API\_KEY', host='https://app.posthog.com')

def track\_event(user\_id, event\_name, properties=None):

"""Track an event in PostHog."""

if properties is None:

properties = {}

posthog.capture(

distinct\_id=user\_id,

event=event\_name,

properties=properties

)

### 1.2 Custom Analytics Services

**Implementation Status**: Partially implemented **Repository Location**: setarcos-api/src/services/analytics\_service.py

In addition to PostHog, the app implements custom analytics services for specialized tracking needs:

#### AI Response Quality Analytics:

* Tracking response quality metrics across different AI models
* Correlating user ratings with model performance
* Analyzing tone performance by philosophical area

#### XP and Gamification Analytics:

* Tracking XP distribution across features
* Analyzing progression rates through levels
* Identifying most/least earned badges and achievements

#### Implementation Details:

# src/services/analytics\_service.py

class AIResponseAnalytics:

def track\_response\_quality(self, response\_id, quality\_metrics):

"""Track quality metrics for an AI response."""

# Store in database

self.db.store\_quality\_metrics(response\_id, quality\_metrics)

# Track in PostHog

self.posthog.capture(

distinct\_id=quality\_metrics['user\_id'],

event='ai\_response\_quality',

properties={

'response\_id': response\_id,

'model': quality\_metrics['model'],

'tone': quality\_metrics['tone'],

'relevance\_score': quality\_metrics['relevance\_score'],

'philosophical\_depth': quality\_metrics['philosophical\_depth'],

'tone\_consistency': quality\_metrics['tone\_consistency'],

'concept\_accuracy': quality\_metrics['concept\_accuracy'],

'coherence\_score': quality\_metrics['coherence\_score'],

'engagement\_potential': quality\_metrics['engagement\_potential']

}

)

## 2. Error Tracking and Monitoring

### 2.1 Sentry Error Tracking

**Implementation Status**: Fully implemented **Repository Location**: setarcos-app (frontend) and setarcos-api (backend)

Sentry provides comprehensive error tracking and monitoring capabilities:

#### Key Features Implemented:

* **Native Crash Reporting**: For both iOS and Android platforms
* **Performance Monitoring**: Tracking app performance metrics
* **Session Replay**: Capturing user sessions for debugging
* **User Feedback Collection**: Gathering user-reported issues
* **Release Health Tracking**: Monitoring stability of app releases

#### Integration Points:

* **Frontend**: React Native integration via @sentry/react-native
* **Backend**: Python integration via sentry-sdk
* **Release Tracking**: Integration with CI/CD pipeline for release tracking

#### Implementation Details:

// Frontend implementation (React Native)

import \* as Sentry from '@sentry/react-native';

// Initialize in App.tsx

Sentry.init({

dsn: 'https://YOUR\_DSN@o123456.ingest.sentry.io/123456',

enableAutoSessionTracking: true,

sessionTrackingIntervalMillis: 10000,

attachStacktrace: true,

debug: \_\_DEV\_\_,

enableNative: true,

integrations: [

new Sentry.ReactNativeTracing({

routingInstrumentation: new Sentry.ReactNavigationInstrumentation(),

tracingOrigins: ['localhost', 'api.setarcos.com', /^\//],

}),

],

tracesSampleRate: 1.0,

});

// Capture exceptions

try {

// Code that might throw

} catch (error) {

Sentry.captureException(error);

}

// Set user context

Sentry.setUser({

id: 'user-id',

email: 'user@example.com',

username: 'username',

subscription: 'premium'

});

# Backend implementation (FastAPI)

import sentry\_sdk

from sentry\_sdk.integrations.asgi import SentryAsgiMiddleware

from sentry\_sdk.integrations.sqlalchemy import SqlalchemyIntegration

from sentry\_sdk.integrations.redis import RedisIntegration

from sentry\_sdk.integrations.celery import CeleryIntegration

sentry\_sdk.init(

dsn="https://YOUR\_DSN@o123456.ingest.sentry.io/123456",

traces\_sample\_rate=0.2,

environment=os.getenv("ENVIRONMENT", "development"),

release=os.getenv("RELEASE", "unknown"),

integrations=[

SqlalchemyIntegration(),

RedisIntegration(),

CeleryIntegration(),

],

)

# Wrap FastAPI app

app = SentryAsgiMiddleware(app)

### 2.2 Comprehensive Logging System

**Implementation Status**: Implemented **Repository Location**: setarcos-api/src/utils/logging.py

The app implements a structured logging system for tracking application events and errors:

#### Key Features:

* **Structured JSON Logging**: Consistent, parseable log format
* **Log Levels**: DEBUG, INFO, WARNING, ERROR, CRITICAL
* **Context Enrichment**: Adding user, request, and feature context
* **Sensitive Data Filtering**: Preventing PII from being logged

#### Implementation Details:

# src/utils/logging.py

import logging

import json

from datetime import datetime

class StructuredLogger:

def \_\_init\_\_(self, name):

self.logger = logging.getLogger(name)

self.logger.setLevel(logging.INFO)

# Add handler if not already added

if not self.logger.handlers:

handler = logging.StreamHandler()

formatter = logging.Formatter('%(message)s')

handler.setFormatter(formatter)

self.logger.addHandler(handler)

def \_format\_log(self, level, message, \*\*kwargs):

log\_data = {

'timestamp': datetime.utcnow().isoformat(),

'level': level,

'message': message,

\*\*kwargs

}

# Filter sensitive data

if 'user' in log\_data and 'password' in log\_data['user']:

del log\_data['user']['password']

return json.dumps(log\_data)

def info(self, message, \*\*kwargs):

self.logger.info(self.\_format\_log('INFO', message, \*\*kwargs))

def error(self, message, \*\*kwargs):

self.logger.error(self.\_format\_log('ERROR', message, \*\*kwargs))

def warning(self, message, \*\*kwargs):

self.logger.warning(self.\_format\_log('WARNING', message, \*\*kwargs))

def debug(self, message, \*\*kwargs):

self.logger.debug(self.\_format\_log('DEBUG', message, \*\*kwargs))

def critical(self, message, \*\*kwargs):

self.logger.critical(self.\_format\_log('CRITICAL', message, \*\*kwargs))

## 3. Performance Monitoring

### 3.1 API Performance Monitoring

**Implementation Status**: Implemented **Repository Location**: setarcos-api/src/middleware/performance.py

The app tracks API performance metrics to ensure optimal response times:

#### Key Metrics Tracked:

* **Request Duration**: Time taken to process API requests
* **Database Query Performance**: Tracking slow queries
* **Endpoint Performance**: Response times by endpoint
* **Error Rates**: Percentage of failed requests

#### Implementation Details:

# src/middleware/performance.py

from fastapi import Request

import time

from src.utils.logging import StructuredLogger

logger = StructuredLogger("api\_performance")

async def performance\_middleware(request: Request, call\_next):

start\_time = time.time()

# Process request

try:

response = await call\_next(request)

status\_code = response.status\_code

error = None

except Exception as e:

status\_code = 500

error = str(e)

raise e

finally:

process\_time = time.time() - start\_time

# Log performance data

logger.info(

"API Request",

path=request.url.path,

method=request.method,

status\_code=status\_code,

duration\_ms=round(process\_time \* 1000, 2),

error=error

)

# Track in PostHog for longer-term analysis

if hasattr(request.state, "user") and request.state.user:

from src.services.analytics\_service import track\_event

track\_event(

request.state.user.id,

"api\_request",

{

"path": request.url.path,

"method": request.method,

"status\_code": status\_code,

"duration\_ms": round(process\_time \* 1000, 2),

"error": error is not None

}

)

return response

### 3.2 AI Model Performance Tracking

**Implementation Status**: Implemented **Repository Location**: setarcos-api/src/services/ai/performance\_monitor.py

The app tracks performance metrics for AI model interactions:

#### Key Metrics Tracked:

* **Response Times**: Time taken for AI models to generate responses
* **Token Usage**: Number of tokens consumed per request
* **Error Rates**: Percentage of failed AI requests
* **Cost Efficiency**: Cost per successful response

#### Implementation Details:

# src/services/ai/performance\_monitor.py

class AIPerformanceMonitor:

def \_\_init\_\_(self):

self.logger = StructuredLogger("ai\_performance")

self.db = get\_db()

async def track\_request(self, model, tone, start\_time, end\_time, token\_count, success, error=None):

duration\_ms = round((end\_time - start\_time) \* 1000, 2)

# Log performance data

self.logger.info(

"AI Request",

model=model,

tone=tone,

duration\_ms=duration\_ms,

token\_count=token\_count,

success=success,

error=error

)

# Store in database for analysis

await self.db.ai\_performance.insert\_one({

"model": model,

"tone": tone,

"duration\_ms": duration\_ms,

"token\_count": token\_count,

"success": success,

"error": error,

"timestamp": datetime.utcnow()

})

# Update rolling averages

await self.update\_model\_averages(model, duration\_ms, token\_count, success)

async def update\_model\_averages(self, model, duration\_ms, token\_count, success):

# Update 24-hour rolling averages

# Implementation details omitted for brevity

pass

### 3.3 Mobile App Performance Monitoring

**Implementation Status**: Partially implemented **Repository Location**: setarcos-app/src/utils/performance.ts

The app tracks frontend performance metrics to ensure a smooth user experience:

#### Key Metrics Tracked:

* **Screen Render Times**: Time taken to render key screens
* **Animation Performance**: Frame rates during animations
* **Memory Usage**: App memory consumption
* **Network Request Performance**: API call durations

#### Implementation Details:

// src/utils/performance.ts

import { PerformanceObserver } from 'react-native-performance';

import \* as Sentry from '@sentry/react-native';

export class PerformanceMonitor {

private observer: PerformanceObserver;

constructor() {

this.observer = new PerformanceObserver((list) => {

const entries = list.getEntries();

entries.forEach(entry => this.processEntry(entry));

});

this.observer.observe({ entryTypes: ['measure', 'resource'] });

}

private processEntry(entry: any) {

// Track in Sentry Performance

Sentry.addBreadcrumb({

category: 'performance',

message: `Performance entry: ${entry.name}`,

data: {

entryType: entry.entryType,

startTime: entry.startTime,

duration: entry.duration,

},

level: 'info',

});

// Track in PostHog for analysis

if (entry.duration > 100) { // Only track slower operations

PostHog.capture('performance\_metric', {

name: entry.name,

entryType: entry.entryType,

startTime: entry.startTime,

duration: entry.duration,

});

}

}

// Measure screen render time

measureScreenRender(screenName: string) {

const startMark = `${screenName}\_start`;

const endMark = `${screenName}\_end`;

performance.mark(startMark);

return () => {

performance.mark(endMark);

performance.measure(

`screen\_render\_${screenName}`,

startMark,

endMark

);

};

}

}

## 4. Alerting System

**Implementation Status**: Basic implementation **Repository Location**: setarcos-api/src/services/alerting\_service.py

The app has a basic alerting system for critical issues:

#### Alert Types:

* **Error Rate Alerts**: Triggered when error rates exceed thresholds
* **Performance Alerts**: Triggered when API response times degrade
* **Security Alerts**: Triggered for suspicious authentication activities
* **Resource Usage Alerts**: Triggered when approaching resource limits

#### Notification Channels:

* **Email**: Alerts sent to engineering team
* **Slack**: Critical alerts posted to dedicated Slack channel

#### Implementation Details:

# src/services/alerting\_service.py

class AlertingService:

def \_\_init\_\_(self):

self.logger = StructuredLogger("alerting")

self.thresholds = {

"error\_rate": 0.05, # 5% error rate

"api\_latency": 500, # 500ms

"auth\_failures": 10, # 10 failures in 5 minutes

}

async def check\_error\_rates(self, window\_minutes=5):

"""Check if error rates exceed thresholds."""

# Implementation details omitted for brevity

pass

async def check\_api\_performance(self, window\_minutes=5):

"""Check if API performance is degraded."""

# Implementation details omitted for brevity

pass

async def send\_alert(self, alert\_type, message, severity="warning", data=None):

"""Send an alert through configured channels."""

# Log the alert

self.logger.warning(

f"Alert: {message}",

alert\_type=alert\_type,

severity=severity,

data=data

)

# Send email for high severity

if severity in ["critical", "high"]:

await self.send\_email\_alert(alert\_type, message, data)

# Send to Slack for all severities

await self.send\_slack\_alert(alert\_type, message, severity, data)

## 5. Integration Points and Data Flow

### 5.1 Analytics Data Flow

The current analytics implementation follows this data flow:

1. **Event Generation**:
   * Frontend events captured via PostHog React Native SDK
   * Backend events captured via PostHog Python SDK
   * Custom events generated by application services
2. **Data Processing**:
   * Events stored in PostHog
   * Custom analytics processed by internal services
   * Performance metrics stored in database
3. **Visualization and Reporting**:
   * PostHog dashboards for key metrics
   * Custom internal dashboards for specialized metrics
   * Scheduled reports via email

### 5.2 Monitoring Data Flow

The current monitoring implementation follows this data flow:

1. **Metric Collection**:
   * Error tracking via Sentry
   * Performance metrics via custom middleware
   * Log generation via structured logging system
2. **Data Processing**:
   * Errors aggregated in Sentry
   * Performance metrics stored in database
   * Logs processed and stored
3. **Alerting and Visualization**:
   * Sentry alerts for errors
   * Custom alerts for performance issues
   * Sentry dashboards for error trends
   * Custom dashboards for performance metrics

## 6. Current Limitations and Gaps

### 6.1 Analytics Limitations

* **Limited Real-Time Analytics**: Current implementation focuses on batch processing
* **Incomplete User Journey Tracking**: Some cross-feature journeys not fully tracked
* **Manual Correlation**: Limited automation in correlating events across systems

### 6.2 Monitoring Limitations

* **Basic Alerting System**: Limited sophistication in alert rules and thresholds
* **Incomplete Infrastructure Monitoring**: Focus on application monitoring with limited infrastructure visibility
* **Manual Incident Response**: No automated incident response procedures

### 6.3 DevOps Integration Gaps

* **Limited CI/CD Integration**: Monitoring and analytics not fully integrated with CI/CD pipeline
* **Manual Performance Testing**: No automated performance testing in deployment process
* **Limited Canary Deployments**: No sophisticated canary or blue-green deployment strategy

## 7. Summary

The Setarcos app currently implements a solid foundation for monitoring and analytics:

* **PostHog**: Comprehensive analytics platform for user behavior tracking
* **Sentry**: Error tracking and performance monitoring
* **Custom Services**: Specialized analytics for AI performance and user engagement
* **Structured Logging**: Consistent logging across the application
* **Basic Alerting**: Email and Slack alerts for critical issues

This foundation provides good visibility into application performance and user behavior but has opportunities for enhancement to reach industry-standard robustness, particularly in the areas of real-time analytics, infrastructure monitoring, and automated incident response.