LLM Observability with Opik

This document explains how Nilla leverages **Opik** for comprehensive LLM observability, enabling cost tracking, performance monitoring, debugging, and quality analysis of AI agent interactions.

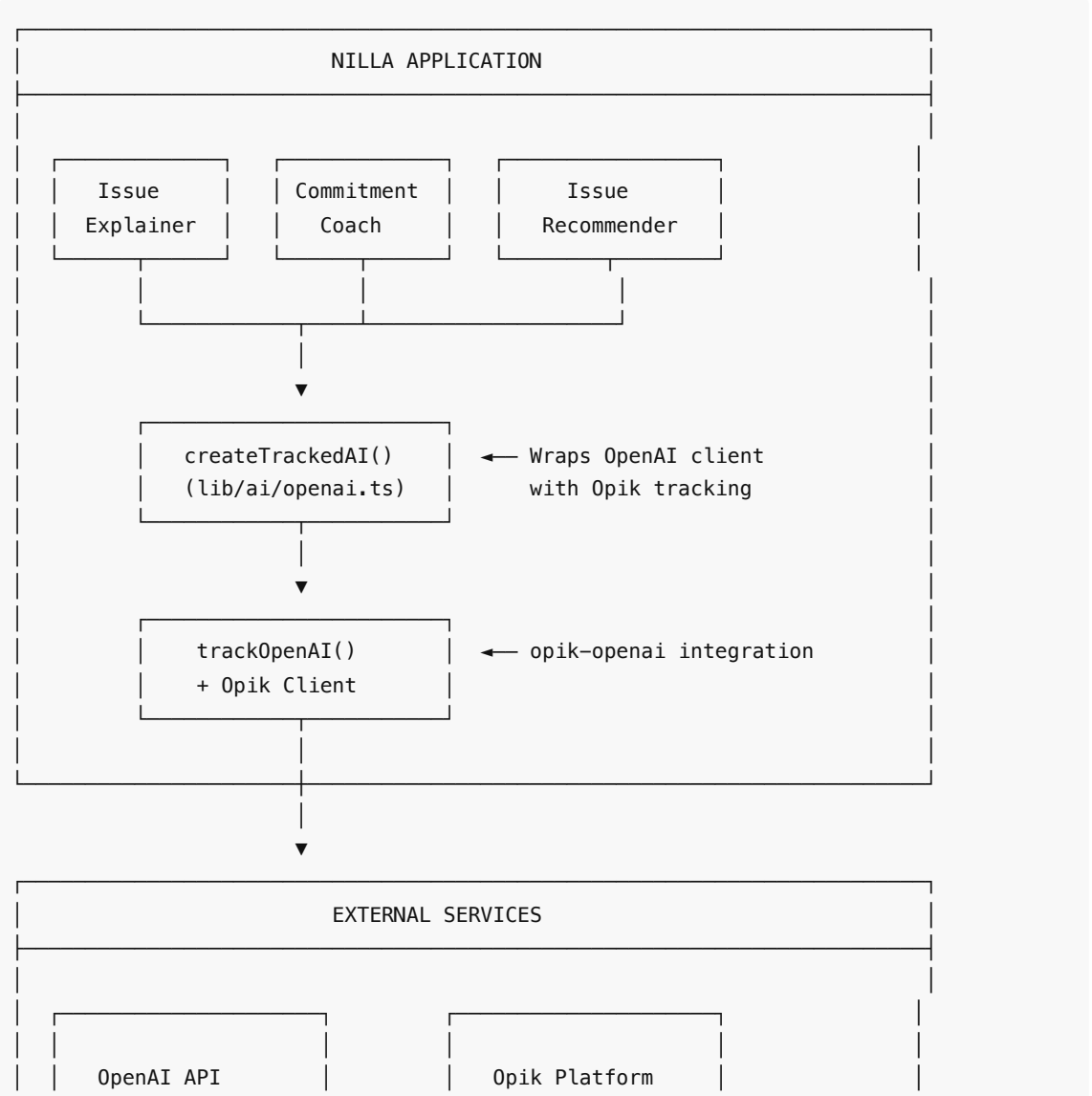
What is Opik?

Opik is an open-source LLM observability platform that provides:

- **Tracing**: Track every LLM call with full request/response logging
- **Cost Monitoring**: Calculate token usage and API costs per request
- **Latency Analysis**: Measure response times across different operations
- **Quality Evaluation**: Analyze output quality over time
- **Debugging**: Inspect prompts and responses for troubleshooting

---

Architecture Overview



```
│  │   (GPT-4o)          │         │   (Traces & Logs)   │              │
│  │                     │         │                     │              │
│  └─────────────────────┘         └─────────────────────┘              │
│                                                                       │
└───────────────────────────────────────────────────────────────────────┘
```

Implementation Details

1. Dependencies

Nilla uses two Opik packages:

```
{
  "dependencies": {
    "opik": "^1.10.2",
    "opik-openai": "^1.10.2"
  }
}
```

- **opik**: Core client for manual tracing and span management
- **opik-openai**: Automatic OpenAI integration wrapper

2. Client Initialization

The OpenAI client is wrapped with Opik tracking in `lib/ai/openai.ts`:

```
import OpenAI from "openai";
import { Opik, flushAll } from "opik";
import { trackOpenAI } from "opik-openai";

// Initialize the OpenAI client
const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
});

// Shared Opik client for manual tracing
export const opik = new Opik();

// Wrap with Opik tracking for observability
// Project name configured via OPIK_PROJECT_NAME env var
export const ai = trackOpenAI(openai, { client: opik });
```

3. Creating Tracked AI Instances

Each agent creates a tracked AI instance with a descriptive generation name:

```
export function createTrackedAI(generationName?: string) {
  return trackOpenAI(openai, {
    client: opik,
    generationName
```

```
  });
}
```

This allows filtering and grouping traces by agent type in the Opik dashboard.

4. Agent Integration Pattern

Every AI agent follows the same observability pattern:

```
// 1. Create a tracked client with agent-specific name
const trackedAI = createTrackedAI("issue-explainer-completion");

// 2. Make the LLM call (automatically traced)
const completion = await trackedAI.chat.completions.create({
  model: "gpt-4o",
  messages: [
    { role: "system", content: systemPrompt },
    { role: "user", content: userMessage },
  ],
  temperature: 0.5,
  max_tokens: 2048,
});

// 3. Ensure traces are sent before returning
await flushTraces();

return result;
```

What Gets Traced

For Each LLM Call

Data Point	Description
Timestamp	When the request was made
Generation Name	Agent identifier (e.g., "commitment-coach-completion")
Model	The LLM model used (gpt-4o)
Input Messages	Full system and user prompts
Output	Complete LLM response
Token Usage	Input tokens, output tokens, total tokens
Latency	Time from request to response
Temperature	Sampling temperature used
Max Tokens	Token limit configured

Agent-Specific Naming

Each agent uses a unique generation name for easy filtering:

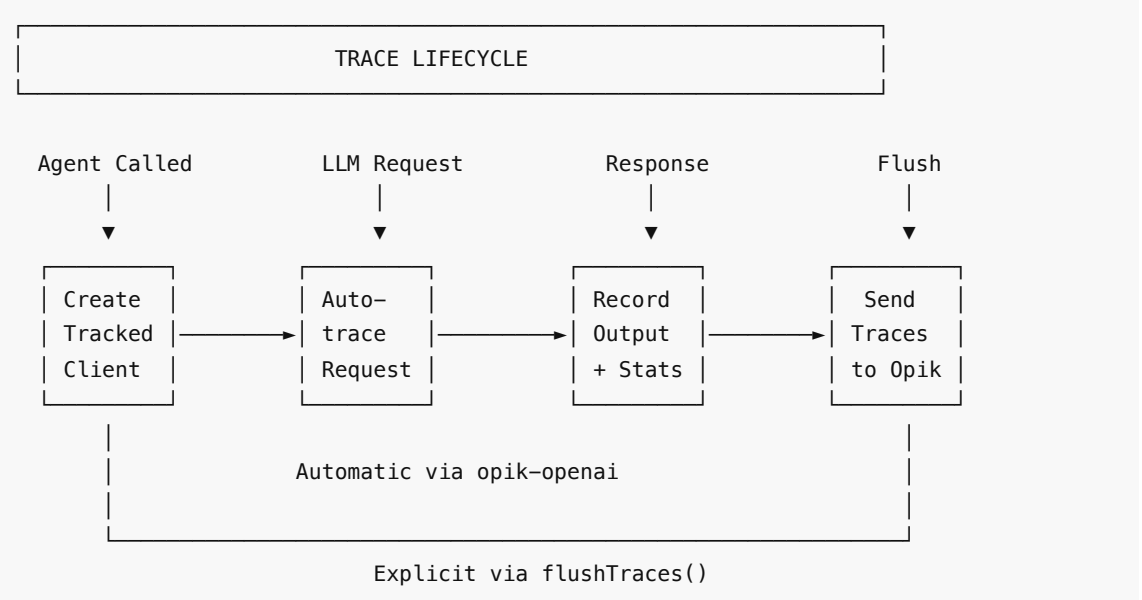
Agent	Generation Name
Issue Explainer	`issue-explainer-completion`
Commitment Coach	`commitment-coach-completion`
Issue Recommender	`recommend-issue-completion`

Environment Configuration

Opik is configured via environment variables:

```
# Opik Configuration
OPIK_URL_OVERRIDE=         # Custom Opik server URL (optional)
OPIK_PROJECT_NAME=         # Project name for grouping traces
OPIK_API_KEY=              # API key for authentication
OPIK_WORKSPACE=            # Workspace identifier
```

Trace Lifecycle



Why Flush Traces?

The `flushTraces()` call ensures all trace data is sent to Opik before the function returns. This is critical in serverless environments (like Next.js API routes) where the process may terminate immediately after the response is sent.

```typescript
export async function flushTraces(): Promise<void> {
  await flushAll();
}
```

Observability Benefits

1. Cost Tracking

Monitor API costs per agent type:

- Track token usage across all requests
- Identify expensive operations
- Set budgets and alerts
- Optimize prompt lengths

2. Performance Monitoring

Measure and improve response times:

- Track latency percentiles (p50, p95, p99)
- Identify slow requests
- Compare performance across agents
- Detect degradation over time

3. Debugging

Troubleshoot issues with full context:

- View exact prompts sent to the LLM
- Inspect raw responses before parsing
- Identify JSON parsing failures
- Trace user-reported issues

4. Quality Analysis

Evaluate output quality over time:

- Review generated explanations
- Compare coaching recommendations
- Validate recommendation logic
- Identify hallucination patterns

Next.js Configuration

Opik packages are excluded from webpack bundling to prevent conflicts:

```typescript
// next.config.ts
const nextConfig: NextConfig = {
  serverExternalPackages: ["openai", "opik", "opik-openai"],
};
```

This ensures the packages run correctly in Next.js server-side environments.

Dashboard Insights

With Opik properly configured, you can:

View All Traces

Browse every LLM call made by the application, filtered by:

- Time range
- Agent type (generation name)
- Model used
- Success/failure status

Analyze Costs

See token usage and estimated costs:

- Per request
- Per agent type
- Per time period
- Cumulative totals

Monitor Latency

Track performance metrics:

- Average response time
- Latency distribution
- Slowest requests
- Trends over time

Debug Failures

When things go wrong:

- View the exact prompt that caused issues
- See the raw LLM response
- Identify parsing errors
- Reproduce and fix problems

Best Practices Implemented

1. Descriptive Generation Names

Each agent uses a clear, consistent naming pattern that makes filtering and analysis straightforward.

2. Explicit Trace Flushing

Every agent calls `flushTraces()` to ensure data is captured in serverless environments.

3. Shared Client Instance

A single Opik client is shared across all agents for efficient connection pooling.

4. Environment-Based Configuration

All Opik settings are configurable via environment variables, enabling different configurations for development, staging, and production.

5. Graceful Degradation

If Opik is unavailable, the application continues to function—observability is additive, not blocking.

Summary

Opik integration in Nilla provides comprehensive visibility into AI agent behavior:

Capability	Implementation
Automatic Tracing	`trackOpenAI()` wrapper
Agent Identification	Unique generation names
Reliable Capture	Explicit `flushTraces()` calls
Cost Visibility	Token usage tracking
Performance Insights	Latency monitoring
Debugging Support	Full prompt/response logging

This observability layer is essential for operating AI features in production—enabling cost control, performance optimization, and rapid debugging when issues arise.