Friction Log

Building the AIOps Orchestrator wasn't just about writing code, it was a journey full of technical challenges, unexpected bugs, and valuable learning moments. Here's a look at some of the key frictions I encountered along the way, and how I worked through them.

1. Installation & Environment Setup

One of the earliest hurdles was keeping the development environment consistent across systems, Windows, Linux, and cloud instances all behaved slightly differently. I ran into dependency mismatches, version conflicts, and environment variable issues.

Solution:
I consolidated all dependencies into a single pyproject.toml and locked them with uv.lock for consistent builds. I also introduced a .env.example file so others could configure their setup easily without exposing secrets. Finally, I used isolated virtual environments to ensure everything ran smoothly across platforms.

2. SDK & Library Limitations

Some third-party monitoring SDKs I explored didn't fully support async operations or advanced metrics. This made concurrent data collection slower and less scalable.

Approach:
Instead of depending too heavily on those SDKs, I built custom monitoring modules inside the monitors/ directory. Leveraging Python's asyncio allowed me to run non-blocking data polling efficiently. I also wrapped external SDKs in modular adapters, making it easier to swap or update them later without touching the core logic.

3. Authentication & Configuration Management

Early in the project, I had sensitive credentials stored directly inside config.py. It worked temporarily but wasn't safe or scalable, a big lesson learned!

Fix:

I moved all secrets into a .env file (which is ignored by Git) and added a public .env.example template for contributors. Configuration loading was then centralized inside storage.py for cleaner management and better security.

4. Context and State Management

As the orchestrator evolved, managing the "state" of each workflow became tricky. Some tasks re-triggered unnecessarily, while others lost context between monitoring cycles.

Solution:

I built a state management layer within storage.py to persist context between runs. Then I improved communication between modules by standardizing data models in models.py. Adding sanity checks and timestamps ensured each orchestration ran predictably and only when needed.

5. Debugging Reasoning Logic

The reasoning engine, the core of automation, was the most interesting challenge. Initially, it made inconsistent or repetitive decisions when multiple events fired simultaneously.

Improvement:

I implemented rule-based prioritization and context caching, then started logging every reasoning path for transparency. Through several iterations and test runs, the engine became much more stable and predictable.

Outcome

Working through these challenges transformed the AIOps Orchestrator into a stable, modular, and intelligent system, one that can operate autonomously and adapt to real-world AIOps scenarios.

Beyond the technical side, this project reinforced the importance of structure, modularity, and secure practices in AI-driven automation systems. Every friction point turned into a valuable learning milestone.