Remote API State Monitoring

Overview

- Introduction
- Concept of Remote State Monitoring & Event-Driven Design
- Comparison of popular libraries or services
- Sample setup illustrating how to detect and respond to API changes
- Recommendations on open-source vs. cloud-based tools
- Proposed solution for our project

01

Introduction

Why API State Monitoring?

- APIs are fundamental integration points in modern distributed systems, providing access to external data and functionality.

- **The Core Problem: Tracking External State Changes**

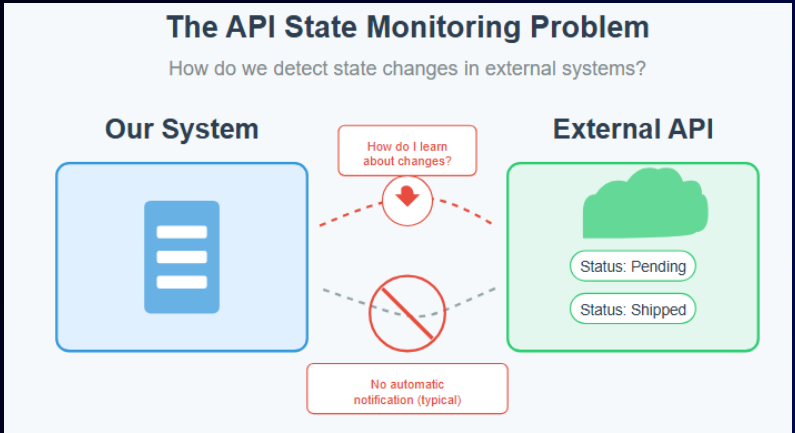
The state or data exposed by external APIs changes over time.

- Examples:

  Order status updates (PENDING -> SHIPPED)
  payment confirmations (RECEIVED)
  sensor readings (VALUE_X)
  task completion status (COMPLETED).

Challenge:

Most standard APIs (esp. REST APIs) do not proactively notify consumers about these state changes. They typically require the consumer to actively query for the current state.



- **Objective**: To implement mechanisms that reliably and efficiently detect these external API state changes to enable timely reactions.

Monitoring Goal: Detect -> Notify / React

- Definition: Remote API State Monitoring

The process of systematically observing an external API to identify significant changes in its state or the data it provides.

Detect: Identify when a relevant state change has occurred in the remote API.

Notify/React: Trigger subsequent actions based on the detected change:

Notify: Send alerts (Email, Message…).

React: Initiate business processes (e.g., trigger data synchronization via "Stay in Sync", update internal state…..).

02

Concept of Remote State Monitoring & Event-Driven Design

Understanding "Remote State" in API Monitoring

- **Definition:** Refers to the data or status maintained by an external system, accessible only through its API.

- **Contrast with Local State**: Unlike internal application state (directly accessible, controllable),

remote state is:

- Owned and managed by a third party.
- Changes without direct internal notification
- Its current value can

only be ascertained by querying the API (Pull) or receiving a notification (Push)

Event-Driven Architecture (EDA)

EDA Core Idea: Systems communicate and react based on the occurrence of significant events (e.g., "Order Shipped", "Inventory Low")

➢ **Benefits of EDA**:

Decoupling: Producers of events don't need to know about consumers, and vice-versa.

Flexibility/Scalability: Easier to add/remove/scale event consumers independently.

Responsiveness/Real-time: Enables systems to react quickly to changes.

Connection to API Monitoring:

A detected relevant change in an API's remote state is an event for a system.

Goal: To build a monitoring system that generates these events (if API is pull-based) or consumes them (if API is push-based), allowing the rest of a system to react in an event-driven manner.

03

Comparison of popular libraries or services

Pull vs Push

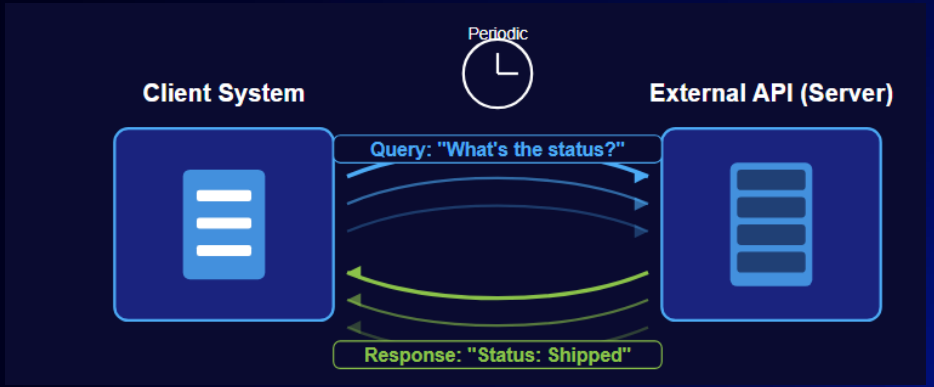
Pull

- **Polling:**

Core Principle: Active & Periodic Querying

A system (Client) actively queries the external API (Server) at regular, predefined intervals (e.g., every hour, every 5 minutes)

The goal is to fetch the current snapshot of the required data for ingestion or synchronization



Implementation Tools & Services (Examples):

A. Custom Scripts + System Schedulers:
Python script scheduled with (e.g. cron).

B. Serverless Functions + Cloud Schedulers:
AWS Lambda/Azure Function triggered by a timer.
Run code snippets on demand without managing servers. Cloud provider handles infrastructure, scaling, and execution based on triggers (like timers). Pay only for what you use (often with a free tier).

C. Workflow Orchestration Tools (e.g., Apache Airflow):
An Airflow DAG with a schedule_interval and a standard operator (e.g., PythonOperator) performing the fetch and write.

Push

Core Principle: Proactive Notification by API
The external API (Server) actively sends a notification (an Event) to a system (Client) when a relevant state change occurs.
This is the more efficient and lower-latency approach, BUT it requires explicit support from the API provider.

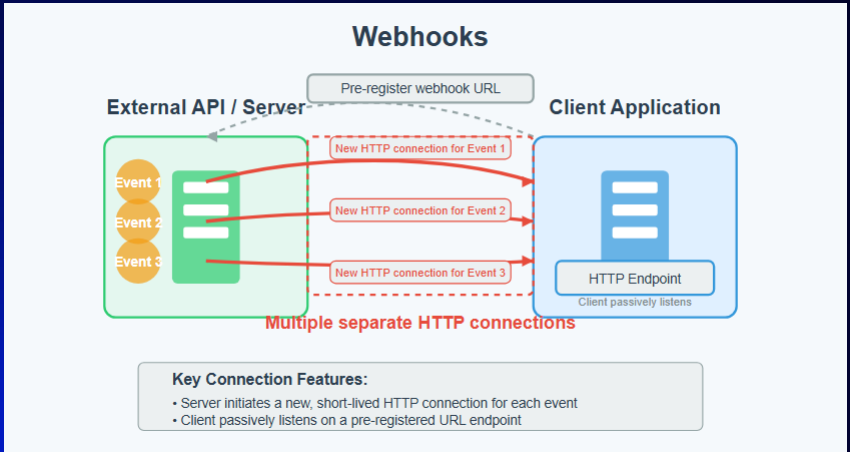
Enables: Event-Driven Architecture (EDA)
Leads to more flexible and faster systems because components react to events instead of calling each other directly

Webhooks

Mechanism: Server initiates a short-lived HTTP POST request to a specific Client URL for each event.

Client Responsibility: Must host an HTTP endpoint to passively listen for and receive these incoming requests.

Connection: Stateless (new request per event)

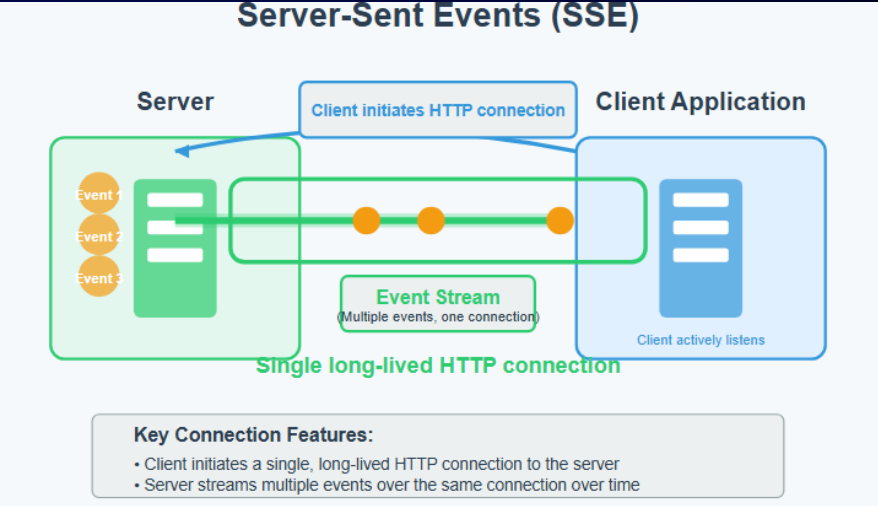


Best For: Discrete events, simple notifications

Server-Sent Events (SSE)

- **Mechanism**: Client initiates a single, long-lived HTTP connection. Server then streams events over this existing connection
- **Client Responsibility**: Must initiate and maintain the connection and actively listen for incoming event messages on the stream.

Connection: Stateful (persistent connection)

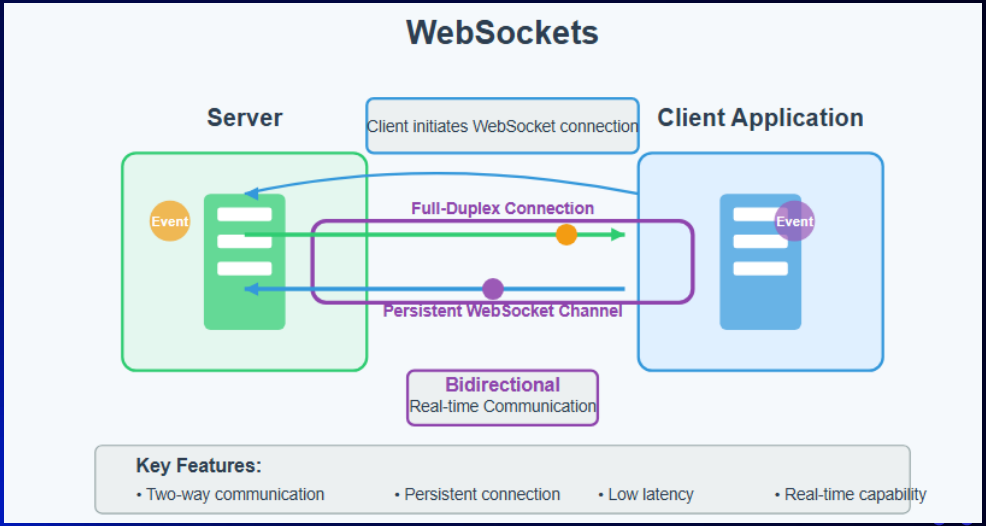


Best For: Continuous or frequent updates (feeds, statuses)

WebSockets

Mechanism: Client initiates a persistent connection. Both sides can send messages anytime.

Client Responsibility: Maintain connection, handle incoming messages, potentially send outgoing messages.

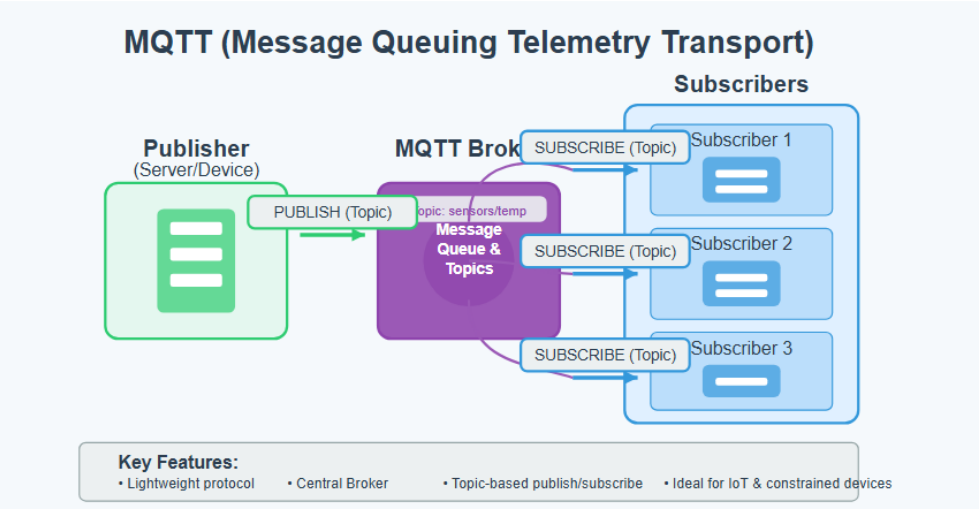


Best For: Real-time, low-latency, bidirectional communication needs

Message Queuing Telemetry Transport (MQTT)

Mechanism: Publish/Subscribe protocol via a central Broker. Clients subscribe to Topics.

Client Responsibility: Use MQTT client library to connect to Broker & subscribe.



Best For: IoT, resource-constrained environments

04

Sample setup illustrating how to detect and respond to API changes

Sample Setup: WebSocket Client

```
SETUP RabbitMQ:
        CONNECT to RabbitMQ
        CREATE Exchange 'websocket_events'
        CREATE Queue 'order.*'
        BIND Queue to Exchange with pattern 'order.*'

CONNECT to WebSocket Server
SEND test message {"status": "SHIPPED", "order_id": 123}

LOOP:
        WAIT for message (max 60 seconds)

        IF no message in 60s:
                TEST connection with ping/pong
                IF connection dead: EXIT LOOP
                ELSE: CONTINUE LOOP

        IF message received: TRY parse as JSON

        IF not JSON:
                IGNORE and CONTINUE LOOP

        IF JSON with status:
                IF status is "SHIPPED":
                CREATE event {timestamp, original_message}
                FORWARD to RabbitMQ (routing: 'order.*')
                ELSE: LOG other status

CATCH WebSocket Connection Error: LOG "Connection failed or closed unexpectedly„

CATCH Connection Refused: LOG "Server not reachable"

CATCH Any Other Error: LOG "Unexpected error occurred"
```

05

Recommendations on open-source vs. cloud-based tools

Open-Source Tools (Self-Hosted)

Custom Scripts – (Self implemented monitoring)

Free Tools: (e.g. Apache Airflow, Dagster, Perfect)

Self-hosted databases for state management (PostgreSQL, MariaDB)

Advantages & Disadvantages

- **Advantages:**

Full Control & Customization: Tailor the solution precisely to your needs.
No Licensing Fees: Software itself is typically free.
No Vendor Lock-in: Greater portability across environments.
Community Support: Often large communities for established projects

- **Disadvantages:**

High Operational Overhead: Responsibility for setup, maintenance, updates and scaling lies with your team.

High setup time : Initial setup and building robust operational practices can be time-consuming.

Cloud-Based Tools

Serverless Functions (AWS Lambda, Azure Functions)

Managed Databases (RDS, Azure SQL, DynamoDB, Cosmos DB)

Cloud-native Monitoring (CloudWatch, Azure Monitor)

- **Advantages:**

Low Operational Overhead: Provider manages infrastructure, updates, and often scaling/availability.
Faster to Setup up: Quick to get started and deploy.
Scalability & Reliability: Often built-in and managed by the provider.

- Disadvantages:

Potential Costs: Can become expensive at scale or with premium features.
Vendor Lock-in: Can be difficult to migrate away from a specific provider's services.
Less Control/Customization: Limited to the features and configurations offered by the service.

06

Proposed solution for our project

Serverless Functions ❌

- **Not Designed for Continuous, Long-Running Execution & Associated Costs:**

Serverless functions are fundamentally designed for **short-lived, event-triggered executions**, not to run continuously like a traditional long-running process.

They have **maximum execution time limits**
Even simulating high-frequency polling by scheduling **very frequent, short invocations** (e.g., every second) leads to a massive number of invocations, quickly exceeding free tiers and resulting in **significant operational costs.**

- **"Blind Spots" & Detection Latency with Scheduled Intervals:**

Because functions are triggered, execute, and then stop, polling is not truly continuous.

The time between these scheduled invocations creates "blind spots" where API state changes can be missed entirely or detected with a significant delay.

This delay and potential for missed events negatively impact the accuracy of timely change detection, especially for stateful conditions like sliding windows that require a comprehensive history of events.

Tools like airflow ❌

- **Not Designed for Continuous, Low-Latency Polling Tasks:**

Airflow tasks are designed to complete, not run indefinitely in a tight, high-frequency polling loop.

Monitoring the *internal state* of a "continuously polling" custom logic *within* a single, long-running Airflow task is not what Airflow's UI is primarily built for.

- **"Blind Spots": Missed API Changes Between DAG Runs**

If a Sensor succeeds and triggers downstream processing, **that specific DAG run stops actively polling.** New API changes occurring during this processing time (or until the DAG is rescheduled) also fall into a "blind spot"

- **Potential "Overkill" for Simple, High-Frequency Polling:**

The entire Airflow infrastructure can be **disproportionately resource-intensive and complex** if the *sole* purpose is to frequently poll a API

DAG = Directed Acyclic Graph

DAG = Directed Acyclic Graph

A DAG defines a collection of tasks and their dependencies, representing your entire data pipeline or workflow.

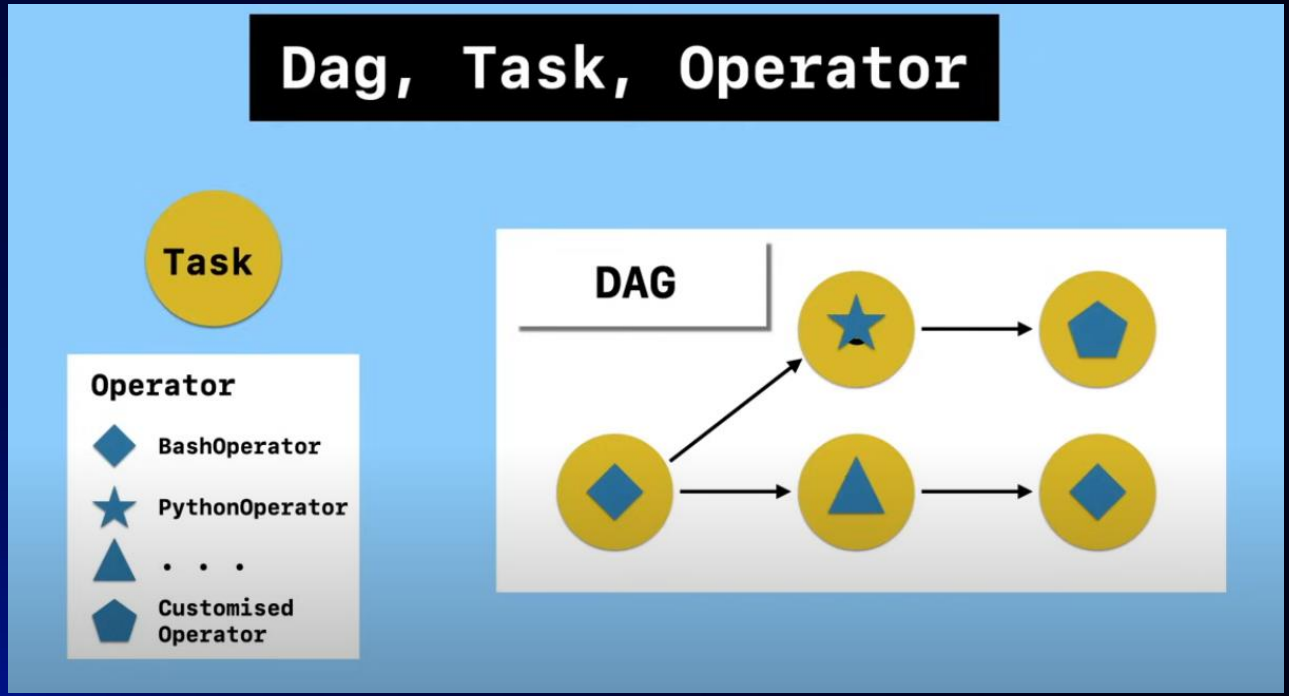
- **Key Characteristics:**

Tasks: Individual work units (e.g., poll API, transform data). Implemented by Operators.

Directed: Tasks run in a specific order; arrows show the flow.

Acyclic: No circular dependencies; the workflow doesn't loop back on itself.

DAG Visualisation



Custom Polling Application with Prometheus & Grafana ✅

Monitoring Stack for This Solution:

- **Custom Application exposes Metrics:**
The polling app provides an HTTP endpoint (/metrics) with internal performance data (poll counts, durations, errors, changes detected).

- **Prometheus collects Metrics:**
Prometheus server regularly "scrapes" these metrics from the polling app.

- **Grafana visualizes Data:**
Dashboards in Grafana display metrics from Prometheus providing visibility into the polling process.

- **PostgreSQL for State Management:**
The custom polling application utilizes our PostgreSQLdatabase to store and compare the "last known state" of APIs, enabling the detection of changes for conditional triggers.

Sources & Refernces

https://aws.amazon.com/de/event-driven-architecture/
https://www.databricks.com/resources/ebook/understanding-etl?scid=7018Y000001Fi0wQAC&utm_medium=paid+search&utm_source=google&utm_campaign=15638819267&utm_adgroup=150401664887&utm_content=ebook&utm_offer=understanding-etl&utm_ad=679973922670&utm_term=data%20transformation%20tool&gad_source=1&gclid=CjwKCAjwtdi_BhACEiwA97y8BDXO7iFJiFTkIgX9hfGFoicEayyzq_shvNtTvBTndqcQ122nUeRGlBoC8m4QAvD_BwE
https://en.wikipedia.org/wiki/Polling_(computer_science)
https://www.enterpriseintegrationpatterns.com/patterns/conversation/Polling.html
https://docs.github.com/en/webhooks/about-webhooks
https://zapier.com/blog/what-are-webhooks/
https://hookdeck.com/webhooks/guides/when-to-use-webhooks
https://ably.com/blog/websockets-vs-sse
https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events
https://mqtt.org/faq/
https://websockets.spec.whatwg.org/
https://www.svix.com/resources/faq/mqtt-vs-websocket/
https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/sensors.html
https://www.youtube.com/watch?v=K9AnJ9_ZAXE&t=1244s
https://man7.org/linux/man-pages/man8/cron.8.html
https://www.advsyscon.com/blog/python-job-scheduling/
https://www.serverless.com/framework/docs/providers/aws/guide/functions
https://dev.to/aws-builders/how-to-trigger-a-lambda-on-a-timer-via-aws-console-serverless-framework-2n6i
https://azure.microsoft.com/en-us/products/functions
https://www.moesif.com/blog/engineering/serverless/A-Primer-On-Serverless-Computing-AWS-Lambda-vs-Google-Cloud-Functions-vs-Azure-Functions/
https://www.rabbitmq.com/tutorials
https://github.com/n1ckdm/websocket-broadcasting/blob/master/backend/app/notifier.py
https://superfastpython.com/asyncio-websocket-clients/
https://websocket.org/tools/websocket-echo-server
https://prometheus.io/docs/introduction/overview/
https://grafana.com/

Pictures: https://claude.ai