Whitepaper: TokenWatcher-Cloud

Author: Diego Raúl Galmarini Version: 7.0 (Revised) Date: June 13, 2025 Project Repository: https://github.com/diegogalmarini/tokenwatcher-cloud

1. Executive Summary

TokenWatcher-Cloud is a software platform designed for the real-time monitoring of significant ERC-20 token transfers on Ethereum Virtual Machine (EVM) compatible networks, such as Ethereum, Polygon, and Arbitrum. This project stems from direct experience with the difficulty of manually tracking relevant on-chain movements, aiming to create an accessible and efficient tool that allows developers, analysts, and Web3 enthusiasts to gain this visibility without incurring the complexity and costs associated with deploying and maintaining their own infrastructure. The platform operates as a service that polls the blockchain at regular intervals, detects transfer events exceeding user-defined thresholds, and sends instant notifications. Management is centralized in a unified dashboard where users can create and manage their "watchers." One of its key features is the "Smart Threshold," which suggests and validates alert thresholds based on real-time market data to ensure notifications are always significant. Developed with Python, FastAPI for the backend, SQLAlchemy as the ORM, and deployed via Docker on the Render platform, this project not only offers a functional solution for token monitoring but also serves as a practical example of applying modern technologies in Web3 application development, on-chain data handling, and cloud service architecture.

Keywords: Blockchain Monitoring, ERC-20 Alerts, Web3, FastAPI, Python, PostgreSQL, Docker, Render, AWS S3, Open Source, Technical Portfolio, Cloud Architecture.

2. Introduction: The Challenge of On-Chain Visibility

The blockchain ecosystem, with its exponential growth, generates millions of daily transactions. This flood of data, while transparent, creates a challenge: how to detect truly important events in real time? As a developer immersed in Web3, I experienced firsthand the frustration of trying to manually track large token movements or the activity of specific contracts using block explorers – a tedious, error-prone, and unscalable task. For traders, analysts, dApp teams, and DAOs, the ability to react quickly to significant on-chain events is crucial. However, existing solutions often involve:

- **Information Overload:** Difficulty in filtering out the noise to find the signal.
- **Delays:** Manual information or data from basic tools does not arrive in time.
- **Technical Complexity:** The need to set up and maintain complex infrastructures (nodes, scalable databases, queues) acts as a barrier.
- **Public API Limits:** Dependence on services like Etherscan introduces bottlenecks due to rate-limiting.
- **Storage Costs:** Saving the complete history of on-chain events can become prohibitive.

TokenWatcher-Cloud was born directly from these challenges, with the goal of building an automated, efficient, and accessible solution. This project is not just a tool, but a demonstration of how to apply software engineering and cloud architecture principles to solve a real problem in the Web3 ecosystem in a pragmatic and sustainable way.

3. TokenWatcher-Cloud: The Detailed Solution

TokenWatcher-Cloud operates as a continuous backend service, focusing on ease of use (via its API) and operational efficiency.

3.1. Key Functionalities

- **Configurable "Watchers":** Allows defining monitoring rules (Watchers) for specific ERC-20 tokens, setting a name and a volume threshold that triggers an alert.
- **Intelligent On-Chain Polling:** A worker (`watcher.py`) periodically queries the blockchain (via Etherscan API) for new transfers (`Transfer` events) for active watchers. It optimizes queries by managing the starting block (`start_block`) for each watcher and handling the external API's pagination.
- **Processing and Filtering:** It analyzes the retrieved transfers, extracts the volume, and compares it against the defined threshold.
- **Logging of Significant Events (`TokenEvent`):** Transfers that exceed the threshold are recorded in the PostgreSQL database, storing key details (watcher ID, contract, volume, TxHash, block, timestamp).
- **Instant Notifications:** Upon logging a relevant event, a formatted alert is sent to Slack (using Block Kit) and/or Discord (using Embeds). The logic includes retries with exponential backoff for robustness against transient network errors or rate-limiting.
- **Management API (FastAPI):** A RESTful API allows for the programmatic management of Watchers (CRUD) and a way to query recent TokenEvents. It includes a `/health` endpoint for monitoring.

- **Unified Management Dashboard:** A modern and intuitive web user interface that allows users to register, log in, and fully manage their watchers (create, edit, pause/activate, delete) and view detected events in real time.
- **Admin Panel:** A private and secure section for the administrator that allows for complete management of the platform's users, including the ability to edit their watcher limits and activity status.
- **Smart Threshold:** A proactive system that, when creating or editing a watcher, queries real-time market data (price, volume, etc.) to suggest an optimal alert threshold and validate that user-configured thresholds are significant, protecting both the user from noise and the platform from abuse.
- **Efficient Data Management (Archiving and Purging):** To ensure long-term viability and control costs (especially on free tiers), an automated process (`cleanup_and_archive.py` executed as a Cron Job):
    - **Archives** events older than X days (configurable, e.g., 1 day) in JSON format to an AWS S3 bucket.
    - **Deletes** the already archived events from the main PostgreSQL database.
    - **Optimizes** the database (`VACUUM ANALYZE`) to maintain performance and recover space.
    - This active data management is fundamental to the service's sustainability.

3.2. Value Flow

1. **Configure:** A `Watcher` is defined via the API.
2. **Observe:** The system monitors the blockchain 24/7.
3. **Detect:** It identifies transfers that meet the `Watcher`'s criteria.
4. **Alert:** It sends an instant notification to the chosen channel.
5. **Log and Archive:** It saves the event for recent viewing and archives it to S3 for historical records, keeping the main database lightweight.

This cycle provides on-chain intelligence passively and in a timely manner.

4. Technical Architecture

The architecture of TokenWatcher-Cloud is based on modular components and proven technologies to achieve efficiency and maintainability.

4.1. Main Components

- **API Backend (FastAPI + Python):** The core of the application, it manages requests, interacts with the database, and exposes business logic. FastAPI was chosen for its performance, typing, and automatic documentation generation.
- **Polling Service (`watcher.py`):** An independent worker (executed as a Cron Job) that performs the intensive task of querying the blockchain (Etherscan API) and processing events.
- **Notifications Module (`notifier.py`):** Encapsulates the logic for formatting and sending alerts to different platforms (Slack, Discord), including error handling and retries.
- **Database (PostgreSQL on Render):** A relational store for active data (Watchers, recent TokenEvents). PostgreSQL was chosen for its robustness and advanced features.
- **Archiving/Cleanup Service (`cleanup_and_archive.py`):** A scheduled worker (Cron Job) responsible for the long-term data management strategy, interacting with PostgreSQL and AWS S3.
- **Historical Storage (AWS S3):** An S3 bucket for cost-effective storage of JSON files with historical events.
- **Containerization (Docker):** The API is packaged into a Docker image, ensuring a consistent environment and simplifying deployment.

4.2. Data Flow and Control Logic

Configuration -> Polling (Read Watchers -> Query Blockchain -> Filter -> Write Events to DB -> Notify) -> Archiving (Read Events from DB -> Write to S3 -> Delete Events from DB -> Optimize DB) -> API Access.

4.3. Key Design Decisions

- **Handling External Dependencies (Etherscan):** Retries with backoff and query optimization are implemented as an initial mitigation for rate limits. The architecture would allow for adding alternative sources (e.g., a proprietary node) in the future.
- **Data and Cost Sustainability:** The strategy of archiving to S3 and purging PostgreSQL is fundamental. It allows the service to operate with a small and inexpensive active database without losing the complete historical record. This design was key to operating within the limitations of free service plans on platforms like Render.
- **Simplified Deployment (Render + Docker):** Using Docker and a PaaS like Render greatly simplifies the deployment and management of the infrastructure (API, database, cron jobs), allowing the focus to remain on application development.
- **Secrets Management:** Environment variables managed by Render are used for all sensitive credentials and configurations, following security best practices.

5. Deployment and Operation on Render

The project is deployed on Render, utilizing its managed services for efficient operation.

5.1. Service Configuration

- **Web Service (`tokenwatcher-cloud`):** Serves the FastAPI API from a Docker container.
- **PostgreSQL Database (`tokenwatcher-db-v3`):** A managed database.
- **Cron Job (`Poll Watchers`):** Executes `watcher.py` periodically.
- **Cron Job (`Purge Old TokenEvents`):** Executes `cleanup_and_archive.py` daily.
- **Cron Job (`Keep-Alive Ping`):** Keeps the free-tier Web Service active.

5.2. Integration with GitHub and CI/CD

The repository is connected to Render for automatic deployments (Auto-Deploy) on every push to the main branch, providing a basic but effective CI/CD workflow.

6. TokenWatcher-Cloud as a Portfolio Project and Open Source Potential

This project is a centerpiece of the portfolio, designed to demonstrate specific technical competencies in modern software development.

6.1. Technical Skills Demonstrated

- **Backend and APIs:** Development with Python/FastAPI, RESTful design, validation with Pydantic.
- **Databases:** Relational modeling (SQLAlchemy), CRUD operations, optimization (`VACUUM`), long-term data management (PostgreSQL).
- **Web3 Interaction:** Consumption of blockchain APIs (Etherscan).
- **Cloud Architecture and DevOps:** Design of workers/cron jobs, containerization (Docker), deployment and management on a PaaS (Render), basic CI/CD (GitHub integration), secrets management, integration with cloud services (AWS S3).
- **Software Engineering:** Modular code, error handling and retries, logging, maintenance scripting.
- **Problem Solving:** Design of the archive/purge solution to overcome cost/storage limitations.

6.2. Open Source Availability

The source code is available on GitHub under a permissive license (e.g., MIT - *confirm or choose license*), inviting review, learning, and potential reuse or forking by the community.

7. Roadmap and Learnings

The evolution of TokenWatcher-Cloud is planned as an iterative development process, divided into key phases.

- **Phase 1 (Completed): Technical Validation and Sustainable Architecture**
    - **Objective:** To validate the core architecture, the on-chain data -> alert flow, and the viability of a sustainable deployment on Render.
    - **Achievements:** Implementation of workers for polling and archiving, integration with Etherscan and AWS S3, and successful deployment of the API and cron jobs.
    - **Key Learning:** The database archiving/purging strategy is essential for cost viability on free service plans.
- **Phase 2 (Completed): Dashboard Fundamentals and Advanced Management**
    - **Objective:** To build a functional user interface for watcher management and event visualization, and to implement advanced business logic.
    - **Achievements:**
        - Development of a complete dashboard with user authentication.
        - Implementation of an admin panel for user management.
        - Creation of the "Smart Threshold" functionality to suggest and validate thresholds.
        - Stabilization of the backend (resolving database and external API errors).

- **Phase 3 (In Progress): Content and Online Presence**
    - **Objective:** To improve the project's public presentation, the onboarding experience for new users, and the structure of marketing content.
    - **Next Steps:**
        - Refactor the homepage to improve its structure and content.
        - Create a footer with links to social media and resources.
        - Develop an "About Us" page.
        - Design an "onboarding" experience for new users in the dashboard.
    - **Long-Term Vision (Experimentation):** Investigate the integration of data analysis (trends, sentiment) as a technical exploration exercise.

8. Conclusion and Call to Action

TokenWatcher-Cloud is a functional and pragmatic demonstration of how to build an efficient and sustainable blockchain monitoring tool. It solves a real problem in the Web3 ecosystem by applying modern technologies and a well-thought-out cloud architecture. This project, deployed and operational, showcases the ability to take an idea from conception to production, managing technical, infrastructural, and cost-related aspects.

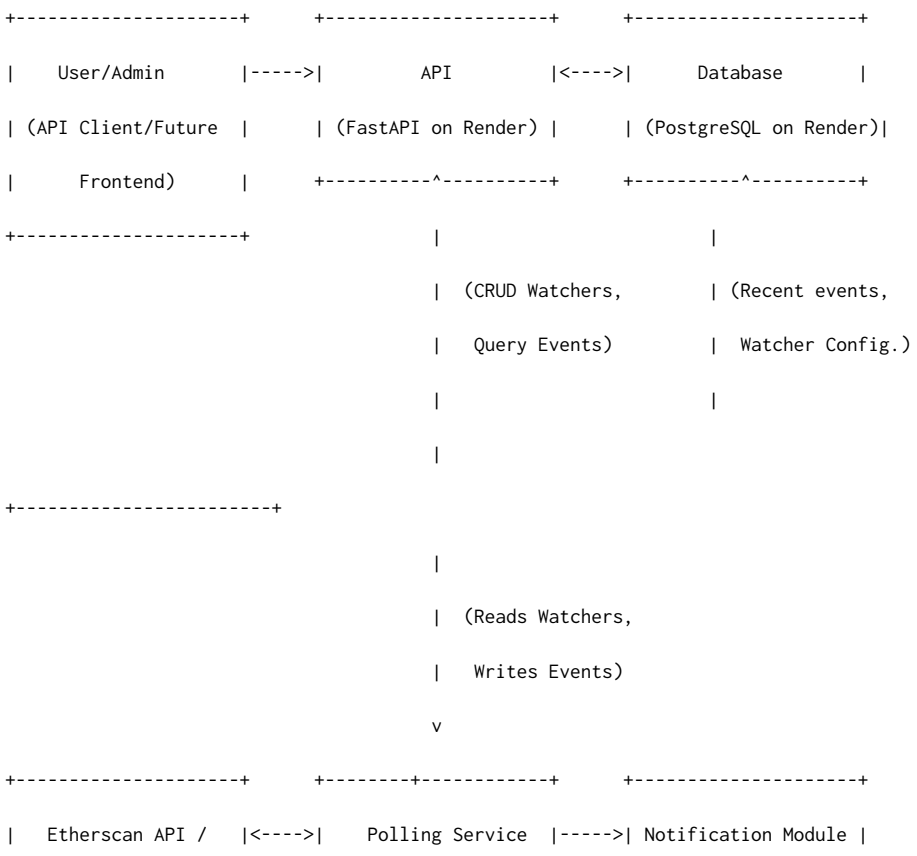
I invite you to explore the project further:

- **Review the source code on GitHub:** https://github.com/diegogalmarini/tokenwatcher-cloud
- **Test the deployed API:** https://tokenwatcher-cloud.onrender.com (refer to the code or the FastAPI documentation at `/docs` for endpoints)
- **Connect with me:** If you have questions, ideas, or want to discuss Web3 technology and backend development, feel free to contact me (link to LinkedIn/Twitter if desired).

This whitepaper documents the current state and vision behind TokenWatcher-Cloud, a tool built with a passion for technology and a desire to provide practical solutions to the blockchain space.

9. Technical Appendix

9.1. Simplified Architecture Diagram



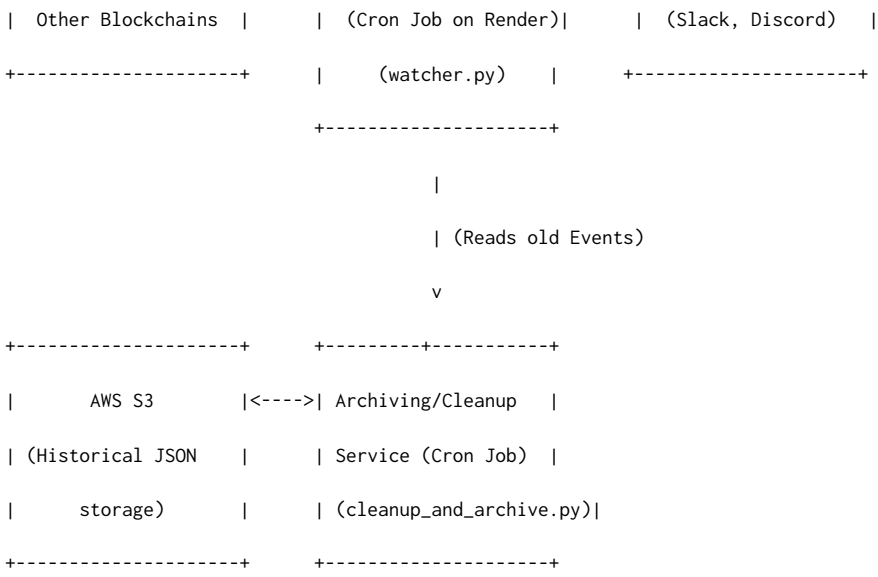


Diagram 1: Simplified architecture of TokenWatcher-Cloud and main data flow.

9.2. API Payload Examples

Create a Watcher (POST /watchers/):

Request Body:

JSON

```
{
  "name": "Monitor Large USDC Movements",
  "contract": "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
  "threshold": 100000.0
}
```

Response Body (example):

JSON

```
{
  "name": "Monitor Large USDC Movements",
  "contract": "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
  "threshold": 100000.0,
  "id": 1,
  "created_at": "2025-05-08T21:40:00.000Z",
  "updated_at": "2025-05-08T21:40:00.000Z"
}
```

Query Events for a Watcher (GET /events/{watcher_id}):

Response Body (example for `watcher_id=1`):

JSON

```
[
  {
    "watcher_id": 1,
    "contract": "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
    "volume": 150000.0,
    "tx_hash": "0xabcdef1234567890...",
    "block_number": 17000000,
    "id": 101,
    "timestamp": "2025-05-08T21:42:00.000Z"
  },
  {
    "watcher_id": 1,
    "contract": "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
    "volume": 250000.0,
    "tx_hash": "0x1234567890abcdef...",
    "block_number": 17000010,
    "id": 102,
    "timestamp": "2025-05-08T21:43:00.000Z"
  }
]
```

9.3. Notification Format (Conceptual Examples)

Slack Notification (using Block Kit):

Conceptual payload for Slack

JSON

```
{
  "blocks": [
    {
      "type": "header",
      "text": {
        "type": "plain_text",
```

```
        "text": ":rotating_light: TokenWatcher Alert: Monitor Large USDC Movements",
        "emoji": true
      }
    },
    {
      "type": "divider"
    },
    {
      "type": "section",
      "fields": [
        {"type": "mrkdwn", "text": "*Contract:*\n`0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48`"},
        {"type": "mrkdwn", "text": "*Volume:*\n150000.0 USDC"},
        {"type": "mrkdwn", "text": "*Block:*\n17000000"},
        {"type": "mrkdwn", "text": "*Date (UTC):*\n2025-05-08 21:42:00"},
        {"type": "mrkdwn", "text": "*Tx:*\n<https://etherscan.io/tx/0xabcdef1234567890...|View on Etherscan>"}
      ]
    }
  ]
}
```

Discord Notification (using Embeds):

Conceptual payload for Discord

JSON

```
{
  "embeds": [
    {
      "title": "🚨 TokenWatcher Alert: Monitor Large USDC Movements",
      "color": 14696747,
      "fields": [
        {"name": "Contract", "value": "`0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48`", "inline": false},
        {"name": "Volume", "value": "150000.0 USDC", "inline": true},
        {"name": "Block", "value": "17000000", "inline": true},
        {"name": "Date (UTC)", "value": "2025-05-08 21:42:00", "inline": false},
        {"name": "Tx", "value": "[View on Etherscan](https://etherscan.io/tx/0xabcdef1234567890...)", "inline": false}
```

```
      ],
      "footer": {"text": "TokenWatcher-Cloud"}
    }
  ]
}
```