Project Report – Real-Time Big Data Processing 2024/2025

Real-Time Analysis of Public GitHub Activity

Roland Bernard, 19598, rolbernard@unibz.it

February 2026

Code Repository	https://gitlab.inf.unibz.it/Roland.Bernard/rtbdp-project, or https://github.com/rolandbernard/rtbdp-project
Commit	58f75794 (latest commit in the repository)
Live Demo Link	https://rtgh.rolandb.com/

Abstract

The open-source software ecosystem generates large volumes of data. Much of this activity takes place on GitHub. While native and third-party analytics tools exist, they predominantly focus on historical data or static snapshots. This project addresses that gap by developing a real-time big data processing application for analyzing public GitHub activity.

The implemented system utilizes a pipeline orchestrated via Docker Compose, leveraging Apache Kafka for messaging and data ingestion from the GitHub Events API. Processing is handled by Apache Flink, which performs complex stream processing tasks including data enrichment, windowed aggregations, and leaderboard generation.

The final application features an interactive single-page application (SPA) frontend built with React and TypeScript, which visualizes data through live event feeds, dynamic charts, and real-time leaderboards. The system successfully processes events at a rate of 70-120 per second, handling challenges such as out-of-order event delivery and real-time ranking updates.

1 Application Domain

1.1 Context and Motivation

Open-source software development has grown into a continuously evolving global ecosystem. GitHub [9] represents the central hub for much of this activity, hosting millions of projects and developers collaborating on open-source projects. As such, the platform generates a large volume of data every second, some of which is publicly available through the GitHub API [10].

While GitHub provides some native analytics, and various third-party dashboards exist, the majority focus on historical data analysis or static snapshots of specific repositories. There is a lack of tools that provide immediate and global insights into the ecosystem as it changes. Developers, researchers, and companies could benefit from monitoring trends in near real time, detecting emerging projects as they start to gain traction, and observing global development activity distribution.

1.2 Use Cases

The system designed for this project is intended to support several key use cases for analyzing the landscape of open-source development on GitHub:

Monitoring Project Popularity By tracking new star events in real time, the system allows users to gauge the immediate community reaction to new releases or announcements.

Discovering Emerging Repositories The system identifies repositories that are rapidly gaining traction, i.e., trending repositories, within short time windows, highlighting potential viral projects.

Observing Development Activity Users can observe the volume of commits, issues, and pull requests to identify active maintenance periods or crunch times in major projects.

Tracking User Contributions The dashboard enables the tracking of highly active users, providing a leaderboard of the most productive contributors across the platform.

1.3 Project Objectives

The goal of this project is to develop a real-time application that addresses these use cases. To this end, the project will continuously acquire public GitHub events from the official API. These raw events will then be enriched with real-time statistics and human-readable descriptions generated by the system. Further, continuously updated leaderboards for active users and

repositories will be generated based on sliding time window statistics. Moreover, The project aims to provide a responsive and interactive web interface allowing users to visualize this information. The frontend should allow filtering events and drilling down into specific repository or user statistics.

The project achieves these goals by implementing a fully distributed pipeline using technologies such as Apache Kafka [2, 13] for messaging, Apache Flink [6, 1] for stream processing, and PostgreSQL [18] for storage. It is a focus of the project to ensure low-latency processing as much as possible to maintain the real-time nature of the dashboard. Further, fault tolerance guarantees and recovery capabilities to handle potential failures in the ingestion or processing layers are considered. Finally, the project provides a reproducible deployment environment using Docker and Docker Compose.

2 Description of the Data Sources

2.1 GitHub Events API

The primary source of data for this project is the official GitHub Events API [11], specifically the `api.github.com/events` HTTP endpoint. This endpoint provides the most recent events from public activity across the entire GitHub platform. The data can be retrieved via periodic polling of the HTTP endpoint to effectively get a high-volume stream of events. Each event is delivered as a JSON object containing detailed information about the action performed.

However, this data source has some limitations relevant to the project. Firstly, the endpoint only offers the 300 most recent events. This means that it is not possible to get older events. Additionally, a maximum of 100 events can be retrieved with a single request, necessitating a mechanism to request three pages at once to capture the full 300-event buffer. Further, the events are not updated continuously, instead empirical testing indicates that all 300 events are refreshed together every second. Finally, the API has strict rate limits associated with unauthenticated requests, requiring the system to use personal access tokens.

2.2 Dummy API and GHArchive

To facilitate offline development and testing without exhausting API rate limits, and without needing any real access token at all, a dummy API server was developed. This component serves historical data sourced from GHArchive [12]. Note that while GHArchive does not require any access token, it is not suitable as the main data source because it is two to three hours delayed and does not include all events. For testing, the dummy API server allows for a configurable initial delay and speed-up factors, enabling the simulation of either high-velocity traffic patterns for stress-testing the Flink pipeline, or low-velocity traffic for easier debugging of potential issues, all without relying on the live GitHub API.

2.3 Data Characteristics


```
{
  "id": "8251545586",
  "type": "PushEvent",
  "actor": {
    "id": 121951544,
    "login": "movieflixgr",
    "display_login": "movieflixgr",
    "gravatar_id": "",
    "url": "https://api.github.com/users/[..]",
    "avatar_url": "https://avatars.github[..]"
  },
  "repo": {
    "id": 1126450259,
    "name": "movieflixgr/Subtitles",
    "url": "https://api.github.com/repos/[..]"
  },
  "payload": {
    "repository_id": 1126450259,
    "push_id": 30546526697,
    "ref": "refs/heads/main",
    "head": "d5e986993aa47729cb18a82ac9d1[..]",
    "before": "6bf91f2d0600084cc7218c1934[..]"
  },
  "public": true,
  "created_at": "2026-02-08T20:09:50Z"
}
```


Figure 1: An example event extracted from the GitHub Events API. It shows the most common type of event, corresponding to the push of a commit to a repository. Some values have been truncated for brevity, indicated with `[..]`.

Variety The data is diverse, with the API returning JSON objects representing a wide array of event types. This includes for example events for commit pushes, opening, closing, and merging pull requests or issues, users starring or forking repositories, and comments created on issues, pull requests, or commits. Figure 1

shows an example of an event representing the push of commits to a repository. This is the most common, but also one of the simplest event types. Each event type possesses a unique nested JSON structure containing detailed information about the actor, the repository, and the specific action payload, which can be significantly more complex than the example shown in Figure 1.

Volume and Velocity Even considering only public events, the volume of events on GitHub is still significant. Observations during the project indicate a rate of 70-120 events per second, depending on the time of day. This accumulates to millions of events per day, translating to roughly 500-600 MiB of data per hour in raw events. While the project implementation defaults to a polling interval of 2.25 seconds to respect rate limits, the architecture is capable of handling much higher frequencies; the live demo, for instance, operates with a polling interval of 750ms. Generally there is little benefit to reducing the polling interval below one second, since the events on the GitHub Events API are only updated once a second.

Veracity and Delay A critical characteristic of this data source is the inherent latency. Events retrieved from the standard `api.github.com/events` endpoint are typically about 5 minutes behind real time. Furthermore, observation has shown that while roughly 95% of events arrive in chronological order, out-of-order delivery is possible. Analysis suggests that 99.8% of events are less than 10 seconds out of order, and 99.9% are within a 5-minute window. However, extreme outliers exist where events may be delayed by several days. This project handles the latter case by dropping events that are too far delayed, motivated by the fact that they represent only a small fraction of all events.

3 Technologies and Overall Architecture

3.1 Technology Stack

Backend Components The backend services, i.e., the producer, the frontend server, and the Flink based processor, are implemented in Java [17]. Apache Maven [3] is used for dependency management and as a build system. Both the producer and the frontend server rely on RxJava [22] for reactive stream processing and managing asynchronous operations. In all backend components the Argparse4j library [28] is used to handle command-line configuration, while Jackson [24] is utilized for high-performance JSON parsing. Logging is standardized by using SLF4J [20] and Logback [19], with JUnit [5] used for unit testing to ensure code reliability.

Frontend Client The user interface of the project is implemented as a Single Page Application (SPA) built with TypeScript [16] and React [14]. Vite [29] is used to serve as the build toolchain. React Router [25] is used to handle client-side routing. Styling is handled via Tailwind CSS [26], and data visualization is powered by the React based Recharts [21] library. Communication with the backend utilizes RxJS [23] to manage WebSocket streams and simple stream operations.

Stream Processing and Storage Further, the project makes use of a number of different technologies for the storage, processing, and streaming transmission of data. Firstly, Apache Kafka [2, 13] is used as the central system for data ingestion and messaging in the project. Next, Kafka Flink [1, 6] serves as the core stream processing engine in the project. Flink has been configured to utilize the RocksDB [15] state backend to manage large state sizes and enable fault tolerance. Flink is further configured for high availability, by using Apache ZooKeeper [4]. To provide persistent storage for processed data, the project uses PostgreSQL [18], a popular open-source database management system. The project also uses the TimescaleDB extension [27], providing facilities for managing time-series data, especially partitioning and retention. Finally, the project makes use of Docker [8] and Docker Compose [7] to orchestrate all the services.

3.2 System Architecture and Data Flow

The project is built on a variation of the provided reference architecture. Figure 2 shows an overview of the overall system architecture with all important components. The system follows a modular architecture, orchestrated via Docker Compose. The workflow is divided into four distinct stages, data ingestion, stream processing, data serving, and display of the results in the frontend.

3.2.1 Data Ingestion

A custom Kafka Producer is responsible for interfacing with the GitHub Events API. It polls the GitHub Events API at a configurable interval and depth. The code is implemented to respect GitHub's rate limiting, automatically pausing if the limit is hit. To capture all events, the polling retrieves overlapping windows

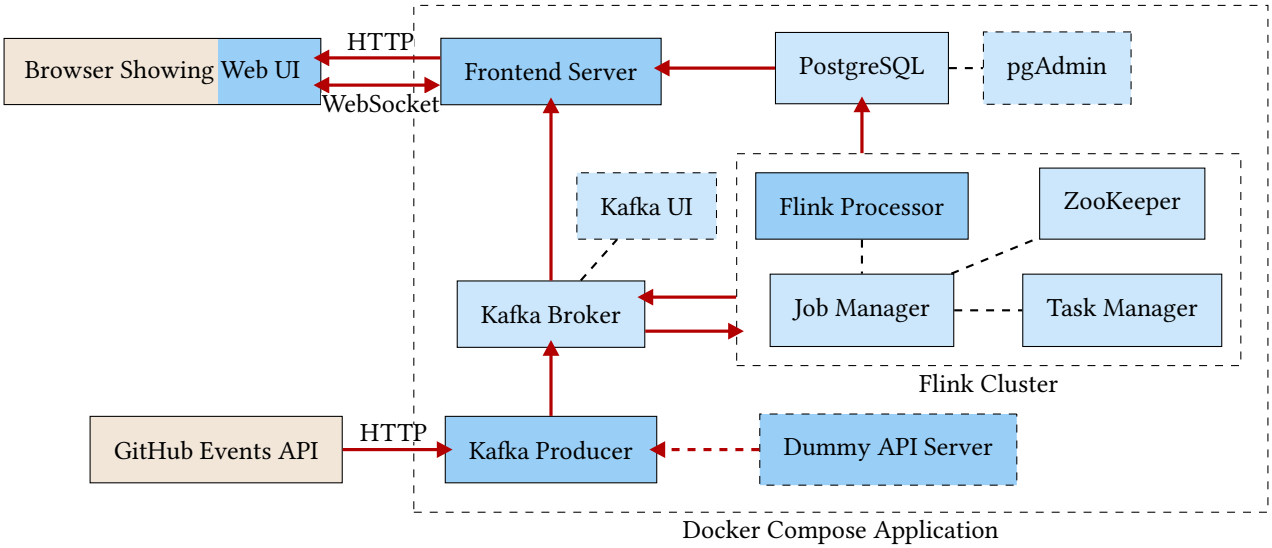

Figure 2: Implemented application architecture. Blue indicates parts of the architecture controlled by the project, while the brown indicates external components. Darker blue represents components implemented as part of the project, while light blue components represent the use of existing software that has merely been configured.

of events with subsequent polls. Therefore, after receiving the list of events from the GitHub API, this component will also handle deduplication of events. For this task, the producer maintains a cache of the most recent event IDs. The producer then publishes the deduplicated raw GitHub events as JSON serialized strings to a dedicated Kafka topic, called `raw_events`.

During real deployment of the application, it is intended that the producer polls from the official GitHub API. However, for testing and development purposes a dummy API server has been developed. It offers the same `/events` endpoint, but does not require a real access token and has configurable throughput. The producer interacts with it in the same way as the official API.

3.2.2 Stream Processing

The core processing on the events is performed in a custom Apache Flink job. It is used to process the stream of events to extract relevant information and compute various aggregations to derive meaningful analytics. It consumes raw events from the `raw_events` Kafka topic, performs several complex processing operations. The transformations performed are organized into a series of modular pipelines, each responsible for feeding one table in PostgreSQL and one topic in Kafka. The following subsections discuss the main processing steps performed in the Flink based processor.

Preprocessing and Time Management The first step in the processing pipeline is to parse the raw JSON events. The job operates in event time by assigning as timestamp the `created_at` timestamp extracted from the JSON payload. A watermark strategy tolerating up to 10 seconds of out-of-orderness is applied. Late events beyond this window are handled via specific allowed lateness strategies depending on the window type of different later operations.

Events are then passed on to subsequent steps, one of which parses the event payloads and generates human-readable descriptions, e.g., `"Roland pushed 3 commits to repo"`. It then takes this description, together with the event kind, the repository ID, and the user ID, as an output. These processed events are what is used in the frontend client to implement the live events stream.

User and Repository Information Not all events involving a specific user or repository contain all information about the concerning entry. For example, in some cases we only get information about the user ID, and are missing the rest of the user information. To improve the user interface, the processor therefore contains some transformations for extracting and aggregating user and repository details. For each event, we extract from the payload all the user and repository details.

Since events often contain only partial information about an entity, these extracted details have to be merged on a per-field basis, as compared to the full row, as is done with the other computation results generated by the Flink job. To handle this, these pipelines

assign per-field sequence numbers to each update. The downstream database sink uses these sequence numbers to ensure that only that part of a row is updated for which incoming data is strictly newer than what is currently stored. A similar merging is applied on the frontend client using the real-time updates it receives.

Windowed Aggregation For historical data display, primarily for the charts in the frontend UI, the processor job performs windowed aggregation. It uses standard Flink Tumbling Windows of 10 seconds and 5 minutes duration to build long-term history of event counts. Event counts are computed for multiple different groupings. First, event count aggregations are computed on a per-type basis. Further, they are also computed per-user and per-repository. Finally, in addition to the total event count, for repositories the job also computes the count of new stars, which is used as an input to the trending score computation.

For real-time counters, standard sliding windows as provided by Flink proved too inefficient. Instead, the system uses a custom operator. This operator maintains circular buckets to aggregate counts efficiently across multiple overlapping window sizes of 5 minutes, 1 hour, 6 hours, and 24 hours simultaneously. This allows the dashboard to show the events in the last 24 hours updated every second without the overhead of maintaining millions of window objects.

Ranking and Trending To generate the user and repository rankings, the process consumes the output of the real-time counters. A custom operator is used to compute and maintain the complete ranking for each category. However, outputting the complete ranking, or even just the rows that have shifted, for each time a live counter changes, would be prohibitively expensive. Instead, the operator simply outputs an update including only the row for which the count changed. However, crucially, the updates produced by the operator include both the new and the old row number, allowing the frontend client to locally reconstruct exactly how the ranking changed.

For the global lists of trending repositories, the number of new stars is used. A custom operator calculates a composite score by combining new star counters across different time horizons, allowing the system to flag repositories experiencing a sudden surge in interest. The trending score is computed as

$$t_{\text{score}} = 10s_{5\text{m}} + 5s_{1\text{h}} + 2s_{6\text{h}} + s_{24\text{h}},$$

where s_t is the number of new stars the repository has received in the latest time window of length t. This is a relatively simplistic model, but works reasonably well to identify trending repositories. By assigning a higher weight to more recently obtained stars, repositories that are experiencing immediate popularity are weighted more highly, but long term popularity is still taken into account.

Output of Results After performing the processing steps to compute the different analytics, the results are first written to PostgreSQL tables. Row updates are modeled as upserts, i.e., they represent an insert or an update of a single row. During this process each row is assigned a sequence number. This number is used by the client to know whether a row it receives from the frontend server is newer or older than the one it may already have. After writing the result to PostgreSQL, the row updates are sent to dedicated Kafka topics. Note that results are sent to Kafka only after they have been committed into PostgreSQL. This sequence ensures data consistency. By committing to PostgreSQL before Kafka, the system prevents a race condition where a frontend snapshot read from the database might miss updates already broadcast over the Kafka stream. After outputting the results to PostgreSQL and Kafka, they are ready to be consumed by the frontend.

3.2.3 Data Serving

A Java-based server acts as a bridge between the backend and the frontend client. It serves static assets and hosts a WebSocket endpoint. The processed data from Flink, that has been published to PostgreSQL tables and Kafka topics, is made available to the frontend client via a WebSocket API. It combines data queried from Postgres with real-time streams consumed from Kafka. The way this works is that the frontend client can request either reads from the table, or subscribe to a specific table with filters. In case of a table read, the frontend server connects to PostgreSQL and then forwards the results to the client. For real-time updates, the frontend server connects to all Kafka topics when starting the server, avoiding having to open a separate Kafka consumer for each client. Then, for each event it receives from Kafka, finds all clients that currently have a subscription matching the event, forwarding the event to those clients.

3.2.4 Frontend

The user interface is implemented as a custom single-page application using React and React Router, served by the frontend server. It offers an interactive dashboard with dynamic charts that update in real time

as new data arrives. To obtain the real-time data, it connects via WebSocket to the frontend server. Upon connection, it subscribes to live updates, and then requests an initial snapshot from the backend, which is sourced from the DB. Doing it in this order ensures that no events are missed, and the sequence numbers are used to ensure new rows are not overwritten with old ones. As new messages arrive via WebSocket, the charts and counters increment in real time.

3.2.5 Storage

The database serves two purposes. Firstly, it is required for providing initial state for the dashboard, without having to wait for all values to update or replaying old events from Kafka. Second, it enables the retrieval of historical results. The database contains a table for each of the aspects computed by the Flink process, such as the events stream, the user and repository details, as well as the live and historical aggregations of event counts. TimescaleDB hypertables are used for some tables for automatic partitioning and using a retention policy to delete old data.

For the rankings, virtual views are used to compute row numbers and ranks based on the live scores. These are virtual views instead of proper tables because the high frequency of updates causes the rankings to change frequently. Since the change of a single row in the live scores could cause a cascade of row changes, using a materialized table would lead to unsustainable update volumes. These are two views to ensure low-latency queries for the frontend. One that is faster when retrieving a range of rows, and one that is faster for querying a very small number of rows. The frontend server dynamically switches between these for best performance.

3.2.6 Messaging

The final component of the architecture is formed by Apache Kafka. Kafka serves to decouple the ingestion from the processing, as well as the processing from the serving of the transformation results in the frontend. In addition to a topic used for the stream of raw events generated by the processor, the architecture utilizes specific topics for each of the outputs generated by the Flink processor. The Flink processor publishes processed results to these output topics for the frontend to consume in real time.

3.3 Scalability and Reliability

The system is designed for both resilience and scalability and efficiency.

Fault Tolerance Flink is configured to take incremental checkpoints of the RocksDB backend. In the event of a TaskManager failure, the job restarts from the last checkpoint, ensuring at-least-once processing semantics of the pipeline. At-least-once semantics are acceptable for this project, since all the outputs of the Flink processor are handled as upserts to the tables. Additionally, Flink has also been configured for high availability with the help of a ZooKeeper container. This means that the system is able to recover from JobManager failure, allowing jobs to restart from the last checkpoint using the metadata stored in ZooKeeper.

Further, if the processing layer goes offline for maintenance or unexpected failure, Kafka retains the message log. Upon restart, Flink processes the backlog to catch up. If however the producer goes down, events will be lost. This is unavoidable because the GitHub Events API does not offer any way to acquire older events than the last 300. One way to mitigate the issue would be by running multiple producers in parallel. This would cause duplicate events to be written to Kafka, but this would be acceptable since there is an additional deduplication step in the Flink processor.

Handling Out-of-Order Data As discussed in Section 3.2.2, a watermark strategy tolerating up to 10 seconds of out-of-orderness is applied in the Flink processor. For the historical windowed aggregations, a specific lateness, 50 minutes for 5-minute windows and 100 seconds for the 10-second window, is additionally allowed for late-arriving events. This will cause an update to previously emitted window results, ensuring eventual consistency. A similar approach is taken for the live window aggregations. Here late events are used only to update the most recently emitted window, and any window emitted in the future.

For row updates in the frontend client, simply applying the latest arriving event is incorrect because the results of the snapshot replay may arrive after the real-time updates from the subscription. To solve this issue, the system assigns a sequence number at the ingestion point before the database sink. The client then only considers the latest version of each row by dropping events that arrive late with an earlier sequence number.

Scalability To improve scalability of the application, the Kafka topics are partitioned, allowing the Flink job to scale out its parallelism. Furthermore, most

operators in the Flink processing job, such as parsing, windowing, mapping are fully parallel.

However, there are some limitations to the parallelism in the Flink processor. Global ranking requires gathering data to a single node to compare with all other values, which represents a theoretical bottleneck. Though sufficient for the current volume of GitHub data, this is an aspect of the architecture that is currently not scalable. Note that parallelization is still possible between the different rankings, i.e., ranking per-user and per-repository activity can be computed independently in parallel. Similarly, the rankings for different time window lengths can be computed in parallel.

4 Functionalities

The user interface is designed to be intuitive while displaying high-density information. This section will go through the key functionalities provided to user, illustrated with screenshots of the application.

4.1 Live Dashboard

Figure 3: Example screenshot of the main dashboard of the application. At the top, right below the header, are the per-kind event counter. Below to the right is the filterable stream of live events and on the left are the user and repository rankings.

Users start at the home page, which represents the main dashboard for the application. It shows a number of different components to provide an overarching view. An example screenshot depicting the user interface of the dashboard can be seen in Figure 3.

Live Event Stream A scrolling feed of the latest events is shown on the bottom right of the dashboard. Each event is parsed into a human-readable description in the Flink processor, and these descriptions are displayed here. Users can filter this stream by event kind, user, or repository, using the input fields right above the live feed.

Real-Time Event Counters Dynamic counters display the total volume of specific event types, e.g., commits, opened or closed issues, or new forks, over sliding windows at the top of the dashboard. The duration of the sliding window is configurable to either 5 minutes, 1 hour, 6 hours, or one day. In all cases they are updated once a second. Next to the counters the UI also displays a chart showing the evolution of the volume over the selected time window.

4.2 Activity Leaderboards

Figure 4: Example screenshot of the leaderboards on the main dashboard. On the left are the user rankings, while on the right we see the repository rankings.

Still on the main dashboard, the application also maintains continuously updated rankings over users and repositories as shown in Figure 4. Users can be ranked by the number of events they generated. Repositories can also be ranked by the volume of activity, but they can additionally be ranked also by either the trending score or the raw number of stars. All rankings, except for the trending score that already combines all window lengths, it is possible for the user to select the length of sliding window for which the ranking is to be shown. Like for the per-kind counter, also the ranking shows a chart to illustrate the evolution over the selected window.

The trending score highlights repositories rapidly gaining popularity. This is calculated using a linear combination of new stars over multiple time windows, allowing the system to detect viral trends faster than simple daily counts, while being less noisy than simply using the 5-minute counts.

4.3 Detailed Pages

Beyond the main dashboard, specific views allow for deeper analysis. These can be navigated through from the main dashboard either through the search functionality, clicking the real-time event counter cards, or by using one of the links in the leaderboards or event stream.

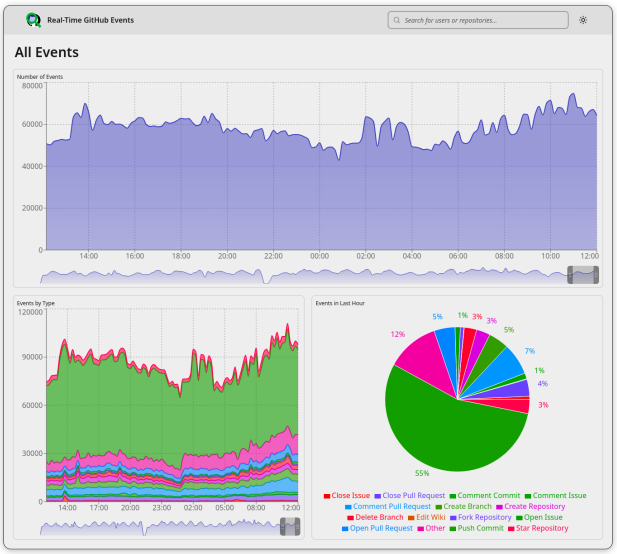

Figure 5: Example screenshot of the event page. Historical counts for the selected event kind at the top, with distribution of events over time or the last hour at the bottom.

Events Page Firstly, there is a page dedicated for each event kind. An example of such a page is shown in Figure 5. This page provides insights into specific types of events and the distribution among event types. At the top of the page, it includes an area chart showing how the volume for the selected event kind changed over time. Additionally, the page includes a pie chart showing the distribution of event types over the last hour and stacked area charts for historical trends at the bottom of the page. A user may click on one of the slices of the pie, or areas of the chart, to quickly switch to the dedicated view for that event type. This will affect the area chart shown at the top.

Repository Page A page dedicated to each individual repository is also provided as depicted in Figure 6.

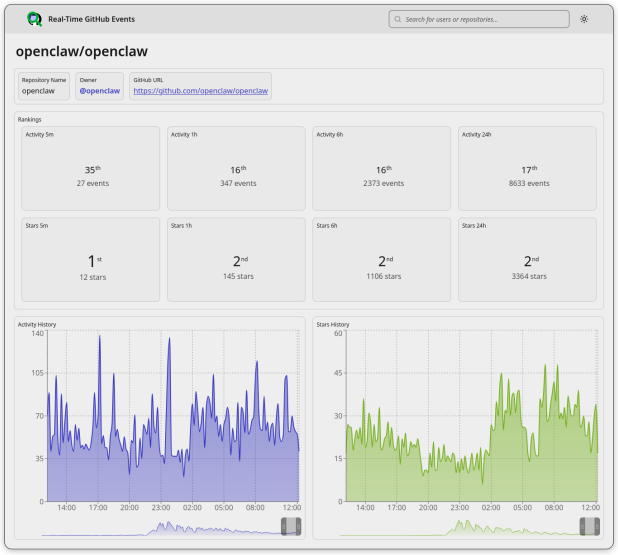

Figure 6: Example screenshot of the dedicated repository page. General repository information is shown at the top. Below that the rankings by different time windows and metrics and at the very bottom the historical activity and new stars.

The page includes, in addition to some general information about the repository such as a link to GitHub, a description, or a link to the owner, also per-repository statistics. It shows the rank of the repository in the global activity and new stars leaderboards for all available time windows. Further, included is also a historical chart of event counts and one of new stars.

User Page Similar to the repository page, details for a specific user's activity history and their global ranking based on contribution volume are shown in a dedicated page. Figure 7 shows an example screenshot of that page. Again, it includes in addition to the computed statistics also some general information about the user, such as the avatar and whether the user is a bot or a regular user account.

4.4 Search Functionality

A global search bar is provided at the top of all pages as shown in Figure 8. The search bar allows the user of the application to quickly search for specific users and repositories based on user or repository name. This enables the application to be used also for analyzing individual user and repositories, which would otherwise be very hard to find if they are not at the top of the leaderboards. Clicking on one of the search results navigates the user directly to the relevant detail page described in Section 4.3.

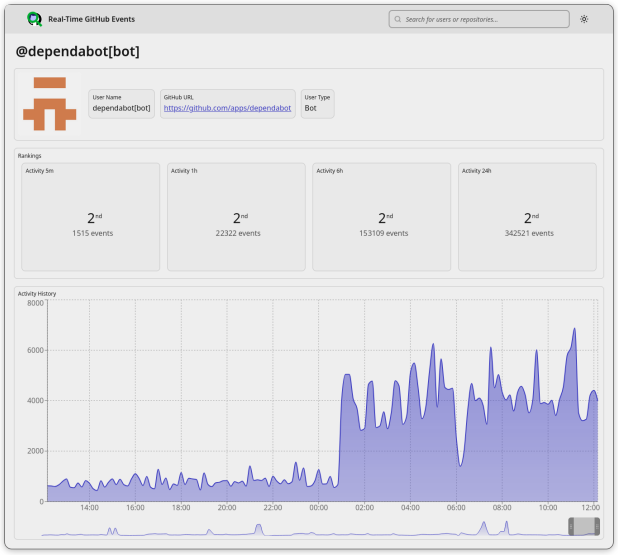

Figure 7: Example screenshot of the dedicated user page. General user information is show at the top. Below that the rankings by different time windows and at the bottom the historical activity.

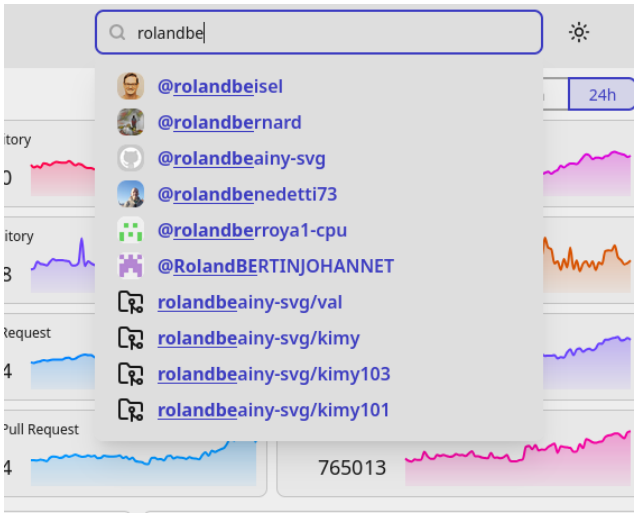

Figure 8: Example screenshot of the search bar. Shows results for the search of "`rolandbe`", including some found users and some repositories.

5 Lessons Learned

This section concludes the report by discussing what has worked well in the project, what were some of the main challenges, and what future improvements could be made to the system.

5.1 Successes

Overall, the architecture proved robust for the defined use case. The combination of Kafka and Flink handled the throughput of the GitHub API relatively well, even on somewhat older hardware. The use of Docker Compose significantly simplified the deployment process, by making the complex distributed system reproducible on different machines. The integration of TimescaleDB allowed for efficient storage and querying of historical time-series data without the operational overhead of managing a separate time-series database. The resulting dashboard provides a window into the development activity of the open-source world.

5.2 Challenges

Even with these successes, the project still faced some challenges.

Real-Time Ranking Generating a global leaderboard that updates in real time was non-trivial. The main difficulty arises from the fact that a large number of ranks can change quickly. For example, if a repository jumps up in the ranking, all rows below may also be moved. Achieving this efficiently required tight coordination between the ranking process in Flink, the view based optimizations in the database, and the ability of the frontend to handle ranking updates in a specialized way.

High-Frequency Updates for Long Windows Standard Flink windowing proved inefficient for maintaining sliding windows that are very long, e.g., 24 hours, but are updated very frequently, e.g., every 1 second. This necessitated the implementation of a custom `KeyedProcessFunction` to optimize state management and triggering.

API Changes During the development lifecycle, specifically around October, GitHub modified their API payload, removing certain details. This caused the processing to crash, requiring refactoring of some parts of the parsing and transformation logic in the Flink processor.

Debugging Flink Gaining sufficient visibility into the internal state of Flink operators during streaming execution was difficult. This made it difficult to diagnose logic errors in the complex windowing functions and ranking logic. Further, there was some difficulty managing state growth for certain Flink operators. This proved especially hard to debug due to Flink's limited facilities for inspecting state.

5.3 Potential Improvements

The project also features multiple potential avenues for future improvements. Aside from adding additional charts using the existing analysis results, the following may be promising additions.

Enhanced Trending Algorithm The current trending score is a linear combination of star counts and as such very simplistic. A more sophisticated model could incorporate the acceleration of stars or social signals such as created issues, pull requests, or comments. This could help to identify trending repositories earlier.

Data Enrichment Integrating additional data sources, such as tracking mentions of repositories on social media such as X or Reddit, would also provide a more comprehensive view of the trending status of repositories. This information could be useful both to identify which repositories are trending but also possibly give insights into why they are trending, something that is currently obscured in the system.

Sentiment Analysis Finally, analyzing the text content of certain event types, such as commit messages, issue descriptions, and pull request comments could provide qualitative insights into the sentiment of different developer communities.

References

[1] Apache Software Foundation. *Apache Flink Documentation*. https://nightlies.apache.org/flink/flink-docs-stable/. Accessed: 2026-01-07. 2026.

[2] Apache Software Foundation. *Apache Kafka Documentation*. https://kafka.apache.org/documentation/. Accessed: 2026-01-07. 2026.

[3] Apache Software Foundation. *Apache Maven Documentation*. https://maven.apache.org/guides/. Accessed: 2026-01-07. 2026.

[4] Apache Software Foundation. *Apache ZooKeeper Documentation*. https://zookeeper.apache.org/doc/current/. Accessed: 2026-01-07. 2026.

[5] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, and Christian Stein. *JUnit 5 User Guide*. https://docs.junit.org/5.13.4/user-guide/. Accessed: 2026-01-07. 2025.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache flink: Stream and batch processing in a single engine". In: *The Bulletin of the Technical Committee on Data Engineering* 38.4 (2015).

[7] Docker, Inc. *Docker Compose Documentation*. https://docs.docker.com/compose/. Accessed: 2026-01-07. 2026.

[8] Docker, Inc. *Docker Documentation*. https://docs.docker.com/. Accessed: 2026-01-07. 2026.

[9] GitHub, Inc. *GitHub*. https://github.com/. Accessed: 2026-01-07. 2026.

[10] GitHub, Inc. *GitHub API*. https://docs.github.com/en/rest/. Accessed: 2026-01-07. 2026.

[11] GitHub, Inc. *GitHub Events API*. https://docs.github.com/en/rest/activity/. Accessed: 2026-01-07. 2026.

[12] Ilya Grigorik. *GH Archive*. https://www.gharchive.org/. Accessed: 2026-01-07. 2026.

[13] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. Vol. 11. 2011. Athens, Greece. 2011, pp. 1–7.

[14] Meta Platforms, Inc. *React Documentation*. https://react.dev/. Accessed: 2026-01-07. 2026.

[15] Meta Platforms, Inc. *RocksDB Documentation*. https://rocksdb.org/docs. Accessed: 2026-01-07. 2026.

[16] Microsoft. *TypeScript Documentation*. https://www.typescriptlang.org/docs/. Accessed: 2026-01-07. 2026.

[17] Oracle Corporation. *Java Platform, Standard Edition 21 Documentation*. https://docs.oracle.com/en/java/javase/21/. Accessed: 2026-01-07. 2026.

[18] PostgreSQL Global Development Group. *PostgreSQL Documentation*. https://www.postgresql.org/docs/. Accessed: 2026-01-07. 2026.

[19] QOS.ch Sarl. *Logback Project*. https://logback.qos.ch/. Accessed: 2026-01-07. 2026.

[20] QOS.ch Sarl. *SLF4J user manual*. https://slf4j.org/manual.html. Accessed: 2026-01-07. 2026.

[21] Recharts Group. *Recharts Documentation*. https://recharts.github.io/. Accessed: 2026-01-07. 2026.

[22] RxJava Contributors. *RxJava Documentation*. https://github.com/ReactiveX/RxJava/wiki. Accessed: 2026-01-07. 2026.

[23] RxJS Contributors. *RxJS Documentation*. https://rxjs.dev/. Accessed: 2026-01-07. 2026.

[24] Tatu Saloranta. *Jackson Documentation*. https://github.com/FasterXML/jackson-docs. Accessed: 2026-01-07. 2021.

[25] Shopify, Inc. *React Router Documentation*. https://reactrouter.com/. Accessed: 2026-01-07. 2026.

[26] Tailwind Labs Inc. *Tailwind CSS Documentation*. https://tailwindcss.com/docs/. Accessed: 2026-01-07. 2026.

[27] Timescale, Inc. *TimescaleDB Documentation*. https://www.tigerdata.com/docs/api/latest. Accessed: 2026-01-07. 2026.

[28] Tatsuhiro Tsujikawa. *The Argparse4j User Manual*. https://argparse4j.github.io/usage.html. Accessed: 2026-01-07. 2015.

[29] VoidZero Inc. *Vite Documentation*. https://vitejs.dev/guide/. Accessed: 2026-01-07. 2026.