Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

June 6, 2025

STORM

Sonification and Transformation of Open-source Repository Metrics

Stipe Peran

Abstract

The evolution of software projects is captured through a chronological sequence of events that reflect ongoing changes introduced during development and maintenance. These records offer insights into various aspects of software work, including programming activity, collaboration, issue tracking, and long-term maintenance.

In practice, extracting meaningful insights from this data remains challenging. Repository data is often large and complex, typically distributed across multiple independently evolving projects, making it difficult to track historical changes, detect anomalies, or compare behaviors across repositories.

Despite the availability of visualization tools, most rely on static charts or dashboards that only present high-level summaries. They do not support interactive exploration of project evolution or side-by-side comparison across repositories, limiting their usefulness for understanding dynamic or anomalous behavior.

To address these limitations, we present STORM (Sonification and Transformation of Open-source Repository Metrics), a tool for exploring software evolution through dynamic visualization and real-time sonification. STORM enables users to explore repository behavior over time via animated scatter plots and auditory cues, offering a multimodal, interactive approach that complements traditional static analysis.

Keywords: software evolution; commit activity; sonification; interactive visualization; software analytics; repository dynamics

Advisor:
Michele Lanza
Co-advisors:
Carmen Armenti, Marco Raglianti

Approved by the advisor on Date: February 25th 2025

Contents

List of Figures

1 Introduction

Software repositories serve as detailed records of how projects evolve over time. They capture a chronological sequence of development events—ranging from feature additions and bug fixes to code reviews and issue resolutions—that reflect ongoing changes in both the codebase and collaborative workflow. Beyond storing source code, repositories preserve a rich temporal history of software activity.

Understanding this evolution is valuable across many domains. Developers assess project progress and stability; project managers monitor workload and detect bottlenecks; researchers study development practices; and educators use real-world data to illustrate software lifecycles. However, analyzing how systems change over time remains a challenge. Projects often span years, involve many contributors, and are distributed across multiple repositories with independent timelines and structures. This complexity makes it difficult to track historical changes, detect anomalies, or compare trends between projects.

Although tools such as GitHub Insights[1], Gource [1] and PyDriller [2] provide access to rich repository data, they predominantly rely on static representations—charts, timelines, and text-based summaries. These formats are useful for high-level reporting but lack support for interactive exploration, temporal navigation, or multimodal perception of change. As a result, users may miss important patterns such as bursts of activity, behavioral shifts, or coordinated changes across repositories.

In this project, we use PyDriller [2] as a framework for mining Git repositories and extracting relevant commit metadata. PyDriller [2] simplifies access to historical data, such as commit timestamps, authorship, and file changes. However, it is not designed for user-facing interaction or visual/auditory representation. To bridge this gap, we introduce **STORM** (Sonification and Transformation of Repository Metrics), a tool that transforms static repository data into dynamic visual and auditory experiences.

STORM combines animated scatter plots with real-time sonification to support a more intuitive, perceptual understanding of software evolution. By mapping repository metrics to motion and sound, it allows users to explore development activity over time and across multiple projects. STORM complements traditional analytics tools by providing a new modality to engage with repository dynamics, making software behavior more observable, comparable, and interpretable.

Report Structure

The remainder of this report is organized as follows:

- **Section 2** reviews existing tools and research in repository visualization and sonification, outlining their strengths and limitations;
- **Section 3** introduces the conceptual motivation and vision behind STORM, outlining its purpose as an exploratory tool for engaging with repository evolution;
- **Section 4** defines the functional and non-functional requirements for the STORM tool;
- **Section 5** presents the overall software architecture and describes the main components in detail;
- **Section 6** illustrates typical use cases that demonstrate STORM's practical applications;
- **Section 7** summarizes the key contributions of the project and discusses possible directions for future work.

[1] `https://docs.github.com/en/repositories/viewing-activity-and-data-for-your-repository`

2 State of the Art

Understanding software evolution has long been a focus of tools and research efforts that mine version control systems such as Git [3]. These efforts have led to a variety of platforms that visualize activity trends, summarize contributions, and support metrics-driven insights.

For example, GitHub's "Insights" tab provides built-in charts showing commit history, code frequency, and contributor timelines. These visualizations offer a quick summary of project activity but are largely static and limited in scope. They lack customization, interactivity, and support for side-by-side comparisons across repositories.

More expressive visual tools such as Gource [1] animate repository activity through visually engaging tree structures. While effective for storytelling or demonstrations, Gource does not offer analytical features such as data filtering, metric selection, or timeline control, making it less useful for deeper exploration.

On the research side, libraries like PyDriller [2] have been instrumental in enabling fine-grained analysis of repository histories. PyDriller [2] offers a rich programmatic interface for extracting commit-level metadata, including authorship, file changes, and timestamps. However, it provides no visualization or interactivity; researchers must build additional infrastructure to interpret the data.

Despite the value of these tools, they all share a common limitation: they rely exclusively on static or visual representations of data. This can obscure temporal dynamics, limit user engagement, and hinder the identification of irregular or emergent behaviors across multiple projects.

The field of sonification—mapping data to sound—has found applications in various scientific domains, but remains underexplored in software engineering. When applied, it often focuses on code structure or quality, rather than capturing temporal activity.

STORM's Contribution

While sharing the goal of improving repository comprehension, STORM takes a different approach. Rather than relying solely on static summaries or aesthetic animations, it introduces a multimodal interface where temporal behavior is both visualized and heard.

Specifically, STORM enables:

- Interactive visualization of commit activity using animated scatter plots driven by D3.js;
- Real-time sonification that maps commit metrics to auditory parameters such as pitch, volume, and stereo position;
- Comparative exploration across multiple repositories, highlighting synchronous patterns, bursts of activity, or stagnation periods.

In contrast to existing tools, STORM emphasizes perception and interaction, supporting educational, analytical, and research use cases where traditional dashboards may fall short. By integrating motion and sound, it helps users grasp not just what happened, but how software development unfolds over time.

3 STORM: Concept and Vision

The conceptual foundation of STORM stems from the observation that software repositories encode more than just source code—they record the history of a project's growth, stagnation, and change. While this history is typically interpreted through static summaries, such representations struggle to convey the *temporal continuity* and *behavioral patterns* embedded in the data. STORM addresses this gap by rethinking how repository evolution can be perceived and understood.

At its core, STORM is an **exploratory tool** designed to enable users to engage with software development as a temporal phenomenon. It combines two complementary modalities:

- **Animated visualizations** capture the motion and progression of repositories over time, using scatter plots that evolve as new data unfolds. This highlights trends, bursts of activity, and structural shifts in a continuous visual timeline.
- **Real-time sonification** transforms repository metrics into sound, allowing users to "hear" changes in activity. By mapping commit frequency to musical parameters like pitch, volume, and timbre, STORM extends perception beyond sight.

This multimodal design is rooted in the idea that perception is not limited to visual cognition. In contexts such as education, analysis, or exploration, sound can serve as a parallel channel for detecting patterns or anomalies, especially those that are difficult to distinguish visually, such as synchronized changes across projects.

Rather than replacing traditional dashboards, STORM aims to *complement* them. It invites users to observe repositories not as static artifacts, but as evolving entities with their own rhythm and flow. This makes it especially suitable for:

- Exploring dynamic repository behavior over time;
- Comparing activity patterns across multiple projects;
- Identifying irregularities or bursts that may indicate important development events.

In sum, STORM challenges conventional approaches to repository analysis by embracing temporality, interactivity, and perceptual diversity.

4 Functional and Non-Functional Requirements

4.1 Functional Requirements

STORM must support the following user-facing functionalities:

1. **Repository Filtering:** Users must be able to select a subset of repositories based on:
    - Repository name
    - Programming language
    - Oldest commit date (start date)
    - Latest commit date (end date)
    - Repository lifetime: The duration of activity, computed as the difference between the latest and oldest commit dates.
2. **Repository Browsing and Selection:** Users must be able to browse a grid of all available repositories and freely select one or more repositories for further inspection or visualization. The grid must display key information for each repository, including its name, URL, date of the oldest commit, date of the latest commit, and the main programming languages used. All columns, except the programming language column, must support both ascending and descending sorting to allow flexible and efficient exploration of the dataset.
3. **Animation Configuration:**
    - Users must be able to set the animation duration.
    - Users must be able to configure the size of the sliding window for both X and Y axes.
    - Users must be able to define the step size in days (i.e., how many days to skip per animation frame).
4. **Tooltip Behavior:**
    - When animation is paused, hovering over a circle must display a tooltip with:
        - Repository name
        - X-axis value
        - Y-axis value
        - Played instrument
    - The tooltip must not be shown while the animation is running.
5. **Click Behavior on Circles:** Clicking on a repository's animated circle triggers a callback that opens the corresponding GitHub page in a new browser tab.
6. **Trend Type Selection:** Users can choose the type of trend computation used for animation:
    - **Global Trend:** computed over the entire dataset, from the earliest commit date across all repositories to the latest.
    - **Local Trend:** computed only within the selected date range.
7. **Sidebar Layout and Controllers:** Both the repository overview and the animated scatterplot views must include dedicated sidebar panels to host interactive controllers. These sidebars should provide users with dynamic control over filtering, selection, and animation parameters. Panels must be toggleable, allowing users to open or close them as needed for an uncluttered interface.
8. **Responsive Scatterplot Scaling:** The animated scatterplot must provide user-selectable scaling options for both axes, allowing users to choose between fixed scaling modes (e.g., linear, square root, logarithmic) and dynamic, frame-by-frame automatic scaling. This flexibility ensures optimal use of screen space throughout the animation, minimizing empty visual gaps and enhancing interpretability.

4.2 Non-Functional Requirements

In addition to core functionality, STORM must meet the following quality and performance criteria:

1. **Animation Smoothness:** The web application must ensure that each animation frame lasts at least 500 milliseconds, allowing smooth transitions. This constraint is enforced by dynamically adjusting the effective step size or displaying a warning if the configuration would result in a too-fast animation.

2. **Responsiveness:** UI components—including the scatterplot and virtualized repository list (powered by `react-window`) — must update without noticeable lag, even with up to 100 repositories selected and animated simultaneously.

3. **Repository Table Interaction:** The repository table is implemented using the `react-window` library [4] to efficiently virtualize rows and optimize performance for large datasets. The table supports sorting on all columns **except** the programming language column.

4. **Usability:** All primary interactions, including filtering, sorting, repository selection, and animation configuration, must be accessible in no more than two clicks and be intuitive to a first-time user. Tooltips and visual feedback (e.g., warnings) must support discoverability.

5. **User Feedback and Warnings:** The application must provide real-time feedback messages in the sidebar, including warnings (e.g., when animation timing constraints are violated).

6. **Cross-Browser Compatibility:** The application must function consistently across all major modern browsers, including Chrome, Firefox, and Safari.

5 System Design

The STORM tool has been designed with a modular architecture that separates data extraction, backend processing, and frontend visualization into distinct but interconnected components. This section presents the object model, overall architecture, and the implementation details of each major component.

5.1 Object-Oriented Model

STORM is structured around a simple but effective object-oriented design. It consists of three main entities: `Miner`, `Repository`, and `Commit`, which together reflect the natural hierarchy of the data mining pipeline.

These classes are not meant to perfectly mirror implementation details, but rather to provide a conceptual model that clarifies how mined data is structured, organized, and processed throughout the system.

- **Commit**: Represents a single Git commit, storing its `hash` and the `date` on which it was created. These two fields are enough to capture the essential time-based identity of each commit.
- **Repository**: Represents a single GitHub repository. It includes metadata such as `name`, `url`, and a list of `languages` (not just one, since repositories often contain multiple). Additionally, it stores:
  - the number of stars (it is solely intended to ensure dataset consistency, see Section 5.3.1)
  - a list of `Commit` objects,
  - a `dailyCommits` list of integers aligned to a global date range,
  - methods to compute `slidingCommits` over any date interval and window size.

  This class provides functionality to query commit activity over time, extract trend information, and compute derived statistics like the dominant language or time bounds of activity.
- **Miner**: Responsible for the scraping and preprocessing phase. While not tied to any specific programming language, this entity abstracts the overall behavior of the Python scraping pipeline:
  - it scrapes a set of repositories from GitHub,
  - it analyzes and extracts commit and metadata information using PyDriller [2] and other tools,
  - it writes out JSON files containing repository metadata and commit lists,
  - it generates one CSV per repository with daily commit counts,
  - it produces a global matrix CSV aligned to the earliest and latest date across all repositories.

  While the `Miner` class in the UML diagram does not include a `repositories` field explicitly (since the real storage is done through JSON/CSV files), it encapsulates the core mining behavior through clearly defined methods such as `mineRepositories(csvPath)`, `cloneAndAnalyzeRepo(url)`, and various file generation utilities.

An overview of this model is shown in Figure 1.

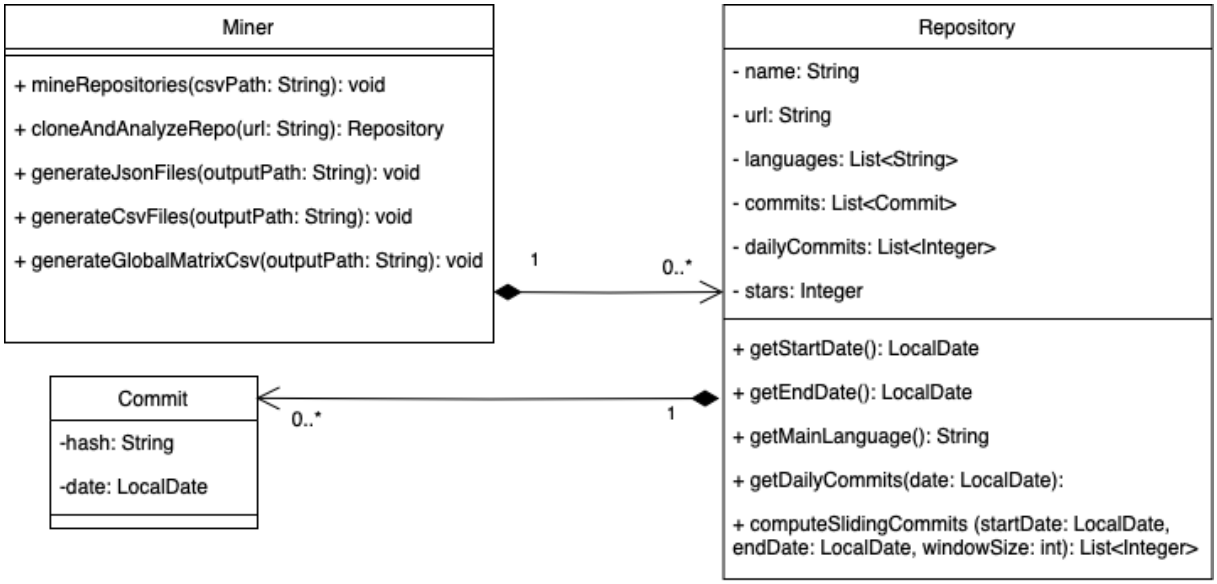


Figure 1. Class UML Diagram of Miner, Repository, and Commit

Rationale Behind Key Design Decisions

- The `dailyCommits` field in `Repository` is modeled as a `List<Integer>` instead of a map, since all repositories are aligned to the same global timeline. Each index corresponds to a day, starting from the oldest commit date across all repositories to the newest. This decision simplifies processing and storage, particularly in CSV format.
- The `languages` field is modeled as a `List<String>` to reflect real-world repositories which often contain a mix of technologies. In the frontend, only the dominant language is used (for example, to determine circle color), but the backend preserves the full set of languages.
- The UML diagram may contain redundant or abstracted methods not directly present in the source code. This is intentional: the purpose of the UML is to clarify design intent, not to reflect implementation line-by-line. For example, `computeSlidingCommits(start, end, windowSize)` generalizes the trend calculations used for both X and Y axes.
- The `Miner` is conceptualized as an active component rather than a static holder of data. Its role is to encapsulate the entire scraping pipeline, providing methods to mine, analyze, and serialize repository data into structured artifacts.

5.2 System Architecture

STORM follows a three-tier architecture, depicted in Figure 2. The tiers are:

1. **Mining Layer (Scraper)**: This layer is responsible for retrieving and preparing the data. The process begins with repository selection using the GitHub Search API [5], applying filters based on programming language, popularity (e.g., stars), and recency to ensure dataset consistency (see Section 5.3.1). Selected repositories are then cloned and mined using PyDriller [2] to extract detailed commit activity. The results are stored in structured formats: each repository is written as a separate JSON file and CSV file, and a global CSV file is generated for statistical and analytical purposes. These files are saved to disk (as shown in the diagram) for further processing. In addition, statistics such as language balance, repository age, and commit distribution are computed to support dataset quality and fairness.
2. **Backend Layer**: The backend is responsible for loading the previously mined data (JSON/CSV) from disk and reconstructing an in-memory model — represented in the figure as the transition from Disk to Heap. Each repository object includes metadata, daily commit counts, and dynamic aggregations over user-defined sliding windows. These metrics are generated on demand in response to requests from the frontend, making the backend both a data store and a computation engine. REST API endpoints allow the frontend to retrieve customized metrics for selected repositories and timeframes.
3. **Frontend Layer**: The frontend provides the user interface for exploring repository activity. As shown in the diagram, the user interacts with the system through this layer. It sends REST API requests to the backend and visualizes the results as an animated scatterplot using D3.js [6]. Users can control playback speed, animation step size, repository filters, and enable real-time sonification, which maps activity metrics to sound features. This makes both visual and auditory trends in repository development easily accessible.

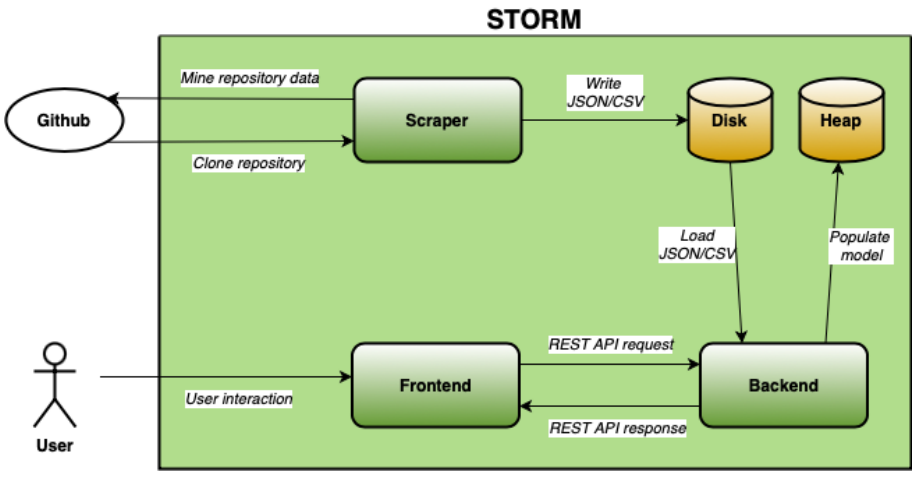


Figure 2. High-level architecture of STORM

5.3 Scraping module

5.3.1 Dataset Consistency Requirements

To ensure fair and meaningful comparisons across repositories, the dataset was carefully constructed using the GitHub Search API [5], aiming for a *balanced* distribution across key dimensions: **programming languages, repository size, popularity, and lifetime**.

Ten programming languages were selected to represent diverse ecosystems—including web development, systems programming, and scripting: `Python`, `JavaScript`, `Java`, `C++`, `Go`, `Ruby`, `Haskell`, `Rust`, `Swift`, and `TypeScript`.

A *stratified sampling strategy* was adopted to ensure internal balance within each language based on commit activity. Three commit count buckets were defined:

- **Small:** fewer than 1,000 commits
- **Medium:** between 1,000 and 10,000 commits
- **Large:** more than 10,000 commits

Each language contributed approximately 50 repositories, with uniform distribution across the commit buckets wherever possible. Additional metadata—such as star count and repository age (in years)—was collected for all entries. The dataset, saved as `repositories_balanced.csv`, was collected using the GitHub Search API on **May 25, 2025**. This timestamp is important as repository metadata (e.g., stars, commits) may have changed after scraping.

Dataset Overview. Figure 3 provides an overview of the dataset with four plots summarizing:

- Repository count per language
- Average commits per language
- Average stars per language
- Average age per language

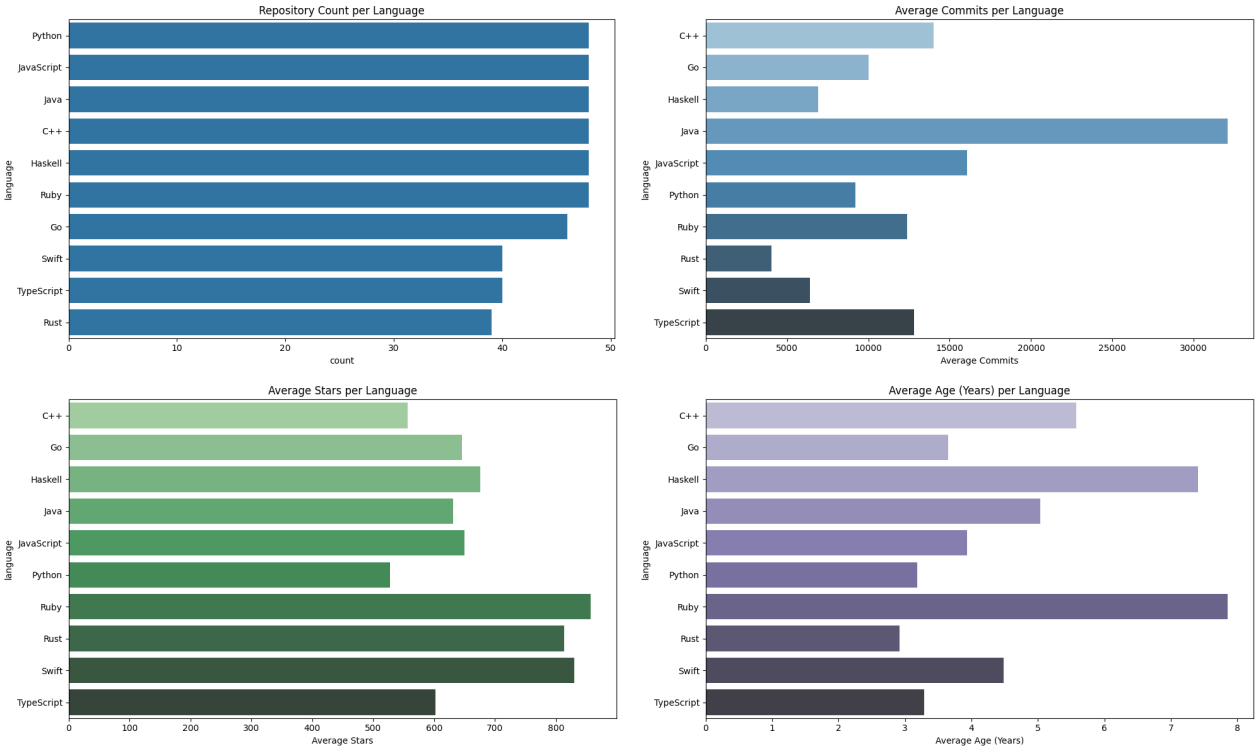


Figure 3. Dataset Overview: (Top-Left) Repository Count per Language, (Top-Right) Average Commits, (Bottom-Left) Average Stars, (Bottom-Right) Average Age

Validation of Stratification. To further validate the stratification, commit count and repository age distributions were plotted in Figure 4. These confirm that the stratified sampling strategy yielded a varied and balanced population across key metrics.

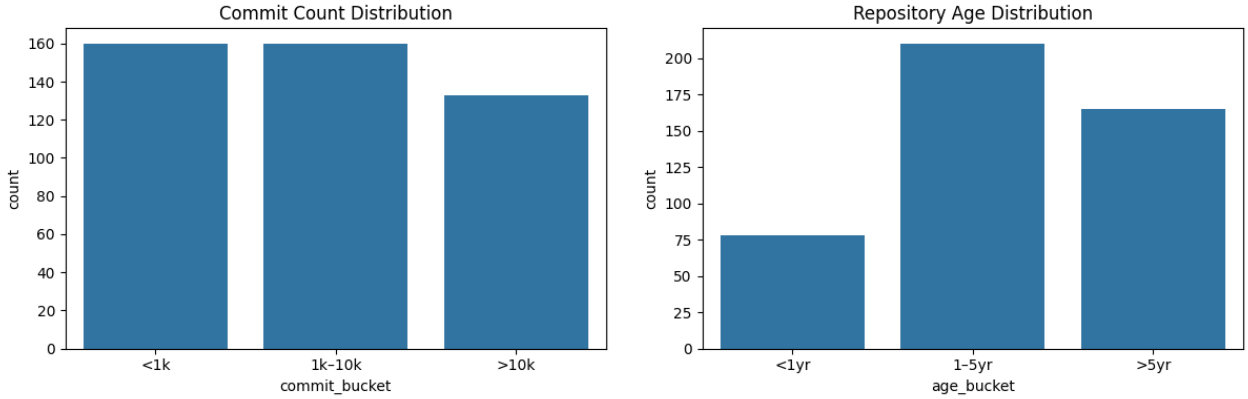


Figure 4. (Left) Commit Count Distribution, (Right) Repository Age Distribution

Stars vs Commits. A final overview is provided in Figure 5, showing a log-log scatter plot of stars versus commit count, colored by language. This offers a visual intuition of the diversity and relative popularity of the repositories across the dataset.

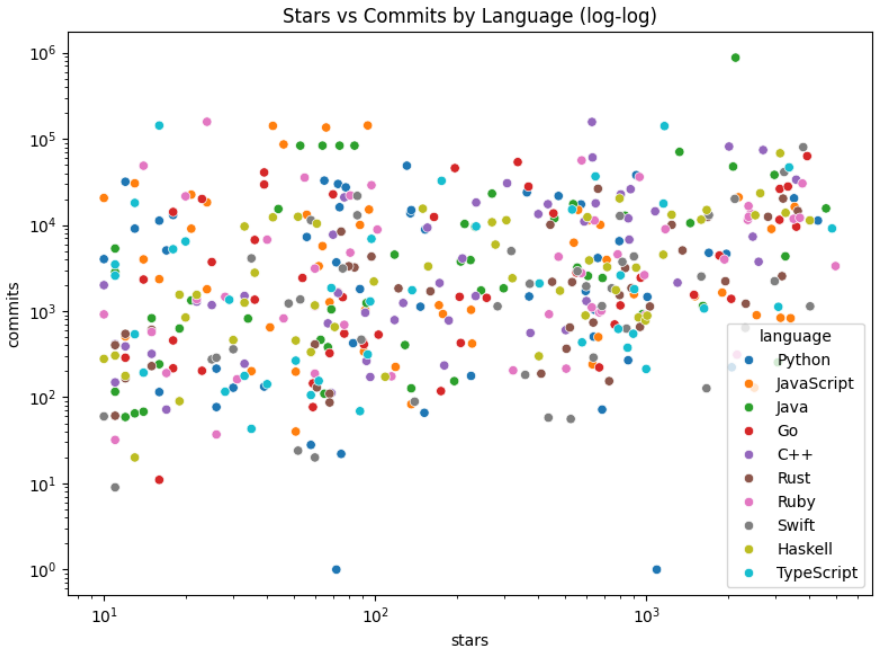


Figure 5. Stars vs Commits by Language (log-log scale)

By enforcing these requirements and validating the results, the dataset minimizes sampling bias and ensures that STORM is evaluated on a realistic and heterogeneous population of repositories. Such consistency is critical to the robustness and generalizability of the system's visualization and sonification capabilities.

5.3.2 Repository Collection and Preprocessing

The **Scraping module** is responsible for collecting and preprocessing open-source repository data to prepare it for visualization and sonification in STORM. Its goal is to construct a consistent, well-aligned dataset enriched with metadata such as commit activity, programming languages, repository age, and popularity. The module ensures uniform formatting and time alignment across repositories to support downstream processing.

The scraping workflow is modularized into three main scripts:

1. **`github_search.py`**: Queries the GitHub REST API to retrieve repository metadata. It filters out forks and stale projects, computes commit count and repository age, and samples a balanced subset of repositories across languages and commit buckets. The result is stored in `repositories_balanced.csv`, the primary dataset file used throughout the pipeline.

2. **`analyze_balanced_dataset.py`**: A standalone analysis script that validates dataset consistency by generating statistical plots. These include commit bucket histograms, language distributions, and scatter plots. Most figures in Section 5.3.1 were produced using this script and saved under `storm-report/figures/`.

3. **`main.py`**: Acts as the central orchestrator for scraping and preprocessing. It reads the list of selected repositories and performs metadata enrichment, time series extraction, and output formatting.

Several helper modules support the scraping process: `utils.py` provides file and path handling utilities; `repo_analysis.py` handles commit time series generation; `log_config.py` standardizes logging across scripts for consistent debugging and progress tracking.

Libraries and Technologies Used The scraping module leverages the following open-source Python libraries:

- **PyDriller** [2]: Traverses Git commit histories to extract commit hashes and timestamps.
- **GitPython** [7]: Handles temporary cloning and cleanup of GitHub repositories.
- **requests**: Communicates with the GitHub REST API to fetch metadata.
- **pandas**: Converts commit timestamps into time series, computes daily frequencies, and exports CSV files.
- **BeautifulSoup** [8]: Parses HTML pages as a fallback to retrieve language data if API limits are reached.
- **datetime, logging (built-in)**: Manages timestamp formatting and logs structured output with progress indicators.
- **matplotlib** [9] and **seaborn** [10] (in `analyze_balanced_dataset.py`): Used for statistical visualizations.

These libraries are lightweight, widely adopted, and well-documented—making the scraping module easy to extend in future projects.

Figure 6 illustrates the execution flow and responsibilities of each script in the scraping module.

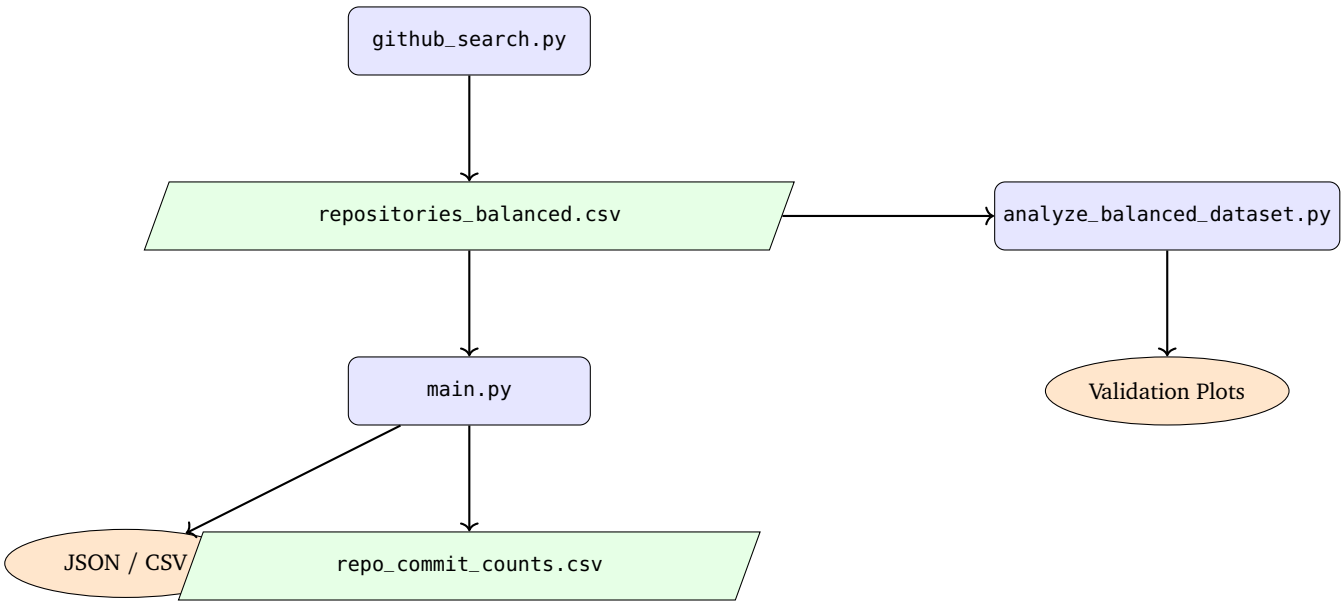


Figure 6. Overview of the scraping module pipeline, highlighting the role of each Python script

5.3.3 Detailed Scraping Pipeline

The scraping workflow implemented in `main.py` transforms the list of selected repositories into structured time series data and metadata-rich JSON files. The process includes the following steps:

1. **Repository Selection:** A list of GitHub repository URLs is read from `repositories_balanced.csv` using the `get_all_urls_and_languages` function.

2. **Local Cloning and Mining:** Each repository is cloned into a temporary directory using GitPython [7]. The `fetch_repository_data` function uses PyDriller [2] to extract all commit hashes and dates.

3. **Language Detection:** To enrich each repository with programming language metadata:
    - The GitHub REST API is queried for language information.
    - If the API fails or rate limits are reached, the HTML page is parsed via BeautifulSoup [8] as a fallback.
4. **JSON Output:** For each repository, a JSON file is created containing:
    - Repository name and URL,
    - List of detected languages,
    - Complete list of commits (date and hash).

    These files are saved in `backend/src/main/resources/data/json/`.
5. **Commit Frequency Aggregation:** The extracted commit dates are converted into daily commit frequencies using `pandas`. Each repository receives a CSV file aligned to a shared global date range, saved in `backend/src/main/resources/data/csv/`.
6. **Global Commit Matrix:** Individual time series are aggregated into a global matrix, `repo_commit_counts.csv`, where:
    - Rows correspond to repositories,
    - Columns correspond to dates,
    - Cells represent daily commit counts.
7. **Cleanup:** Temporary repository clones are deleted after processing to save disk space and avoid clutter.

All outputs are saved under `backend/src/main/resources/data/`, organized into subfolders for `json/`, `csv/`, and temporary working directories.
Each scraping iteration is tracked with structured logging, e.g.:

```
Scraping repository 132/423:  facebook/react (https://github.com/facebook/react)
```

The final directory layout, shown in Figure 7, organizes all output files generated by the scraping module—including raw metadata, time-aligned commit series, and aggregated matrices—under the backend's resource directory. This structure ensures seamless access by the Spring Boot backend, enabling it to serve processed data directly to the frontend without additional transformation steps.

```
backend/
|- src/
   |- main/
      |- resources/
         |- data/
            |- csv/
            |  |- repo1.csv
            |  |- repo2.csv
            |  \- ...
            |- json/
            |  |- repo1.json
            |  |- repo2.json
            |  \- ...
            |- repo_commit_counts.csv
            \- repositories_balanced.csv
```

Figure 7. Output directory structure of the Scraping module

Although scraping logic is implemented as a separate Python module, all output files, including JSON metadata, daily commit CSVs, and the global commit matrix, are saved directly under the `backend/src/main/resources/data/` directory. This design decision ensures consistency with the system architecture: since the backend is responsible for serving and transforming data models, it is logical for the data storage to reside within the backend module rather than within the scraping module. This also simplifies deployment and access, as the backend can directly read and expose the data without additional transfer steps.

5.4 Backend Module

The backend module of STORM is implemented in Java using the Spring Boot framework [11]. It serves as the data orchestration layer, responsible for loading, preprocessing, and exposing repository metrics to the frontend via a RESTful interface. To maximize responsiveness and simplify deployment, the backend operates entirely in memory, foregoing persistent storage. This design is suitable for moderate-scale datasets and supports rapid prototyping and exploration.

Architecture and Dependencies The backend relies on several well-established libraries:

- **Lombok** [12]: Eliminates boilerplate code using annotations such as `@Getter`, `@Setter`, and `@RequiredArgsConstructor`;
- **Jackson** [13]: Parses JSON metadata files and maps them to Java objects;
- **SLF4J** [14]: Facilitates consistent and structured logging throughout the backend;
- **JUnit 5** [15]: Provides unit tests to verify key functionalities such as trend computation;
- **Spring Web MVC**: Implements RESTful routing, request parsing, and response serialization;
- **Jakarta Annotations**: Supports lifecycle hooks such as `@PostConstruct` for initialization routines.

Initialization and In-Memory Model At application startup, the `RepositoryService` loads all repository data from the `data/json/` and `data/csv/` directories. The initialization process involves:

- Parsing each JSON metadata file into a `Repository` instance containing name, URL, dominant languages, and date range;
- Loading daily commit counts from corresponding CSV files;
- Aligning time series to a shared global timeline;
- Precomputing default sliding commit trends (`slidingCommitsX` and `slidingCommitsY`) using a window size of 1;
- Storing the resulting repository objects in an in-memory list.

This in-memory strategy enables constant-time lookup and fast computation, but may become memory-bound with large datasets.

Each `Repository` object encapsulates:

- Raw metadata fields (e.g., `name`, `url`, `languages`);
- A daily commit time series represented as a list of integers;
- Trend vectors `slidingCommitsX` and `slidingCommitsY`, each computed via a configurable moving window.

Both global trends (computed across the entire dataset) and local trends (restricted to a user-defined date interval) are supported, depending on frontend request parameters.

REST API Endpoints The backend exposes its functionality through a set of RESTful endpoints defined in `RepositoryController`:

- `GET /api/repositories` – returns the full list of loaded repositories;
- `POST /api/repositories/filtered` – filters and returns only the repositories specified by a list of IDs;
- `POST /api/repositories/sliding-commits` – computes and returns X and Y trend vectors based on frontend parameters.

Requests from the frontend (typically running at `http://localhost:3000`) are permitted via a CORS configuration defined in `WebConfiguration`.

Trend Computation Logic Trend smoothing is implemented in the `Repository` class using the method:

```
getSlidingCommits(isGlobalTrend, windowSize, startIndex, endIndex)
```

This method applies a moving sum over the commit time series, clamping bounds to avoid index errors. Edge cases such as empty intervals or excessive window sizes are handled defensively. The backend aggregates bulk trend computations using:

```
computeSlidingCommits(windowX, windowY, isGlobal, startIdx, endIdx, selectedIds)
```

This method processes all repositories or a selected subset and returns structured trend data to the frontend.

As shown in Figure 8, the backend module orchestrates data loading, in-memory processing, and dynamic trend computation before serving results through a REST API interface.

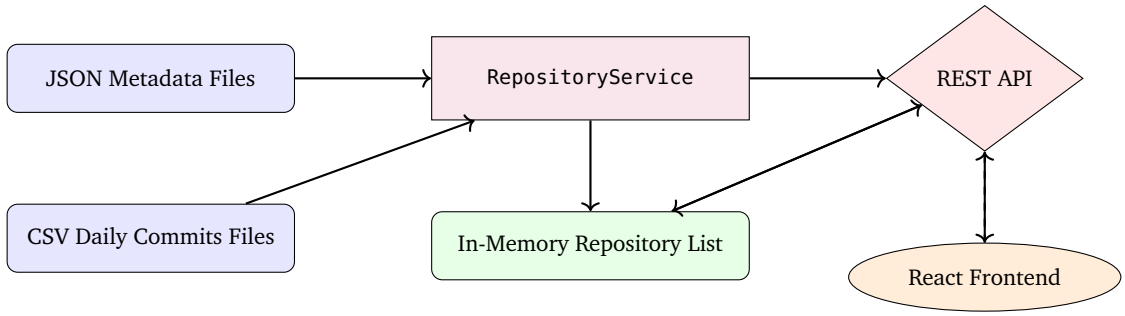


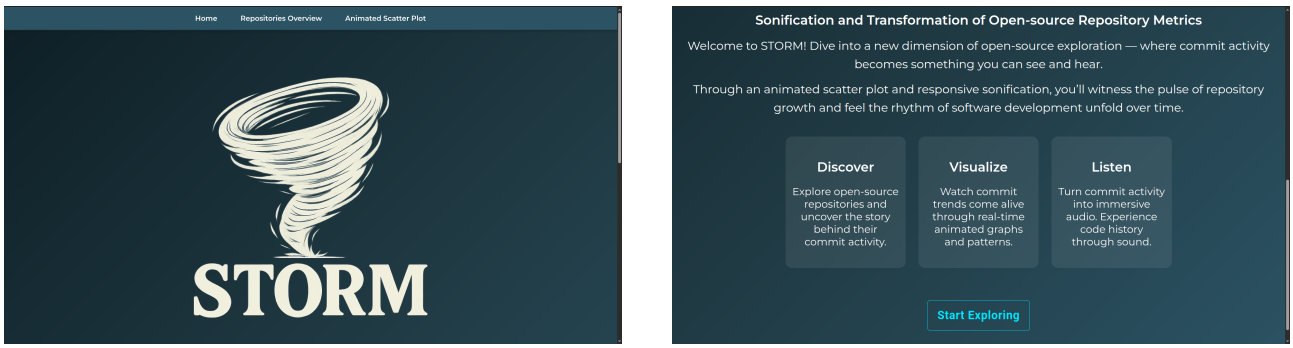
Figure 8. Backend data flow. `RepositoryService` loads and processes repository data at startup and upon request. The REST API layer retrieves this data from memory or delegates computations to the service, responding to frontend queries.

5.5 Frontend Module

The frontend is implemented using React [16], a declarative JavaScript framework, and D3.js [6], a powerful library for data visualization. It provides users with a highly interactive experience for exploring repository commit trends both visually and sonically.

Component Overview The frontend is organized into several modular components. The most important ones are described below, each accompanied by a representative screenshot.

- **HomePage**: Introduces users to STORM with animated cards and a "Start Exploring" button leading to the overview.



(a) STORM logo

(b) Homepage with "Start Exploring" section

Figure 9. HomePage component with branding and entry point

- **RepositoriesOverview**: Displays a virtualized and sortable list of repositories using `react-window` [4]. Users can filter repositories by name, language, and year. The table presents essential repository metadata such as name, URL, number of commits and programming languages. It supports sorting on all columns except the programming language column.

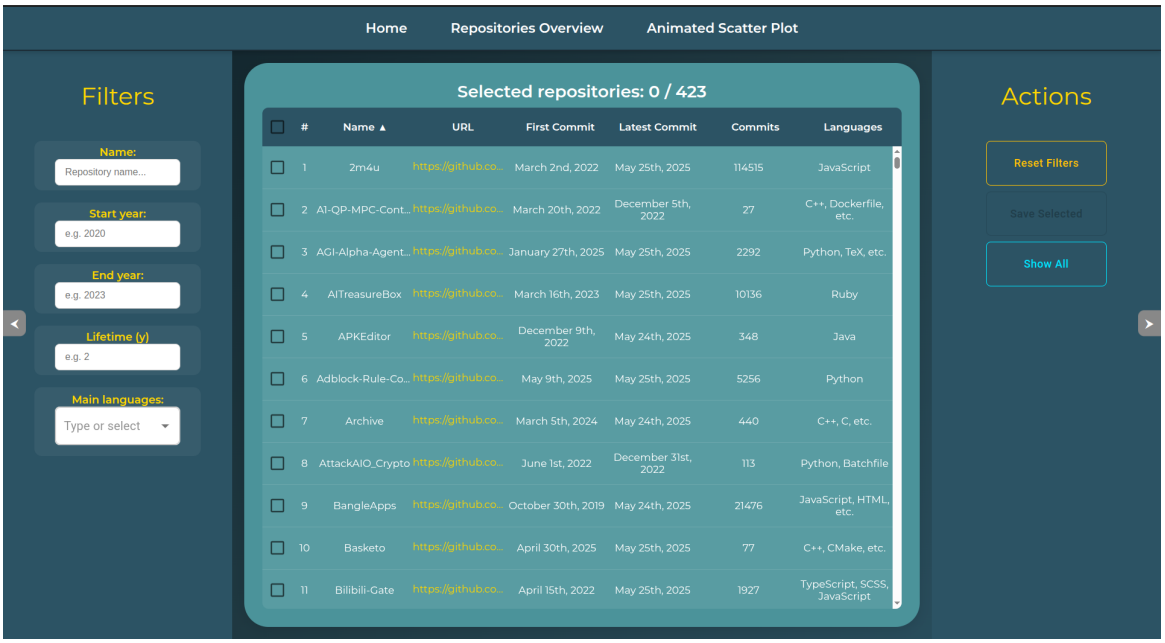


Figure 10. RepositoriesOverview component with filters and sorting controls in the sidebars

- **AnimatedScatterPlot**: The core visualization component that animates and sonifies repository trends over time using a scatterplot.

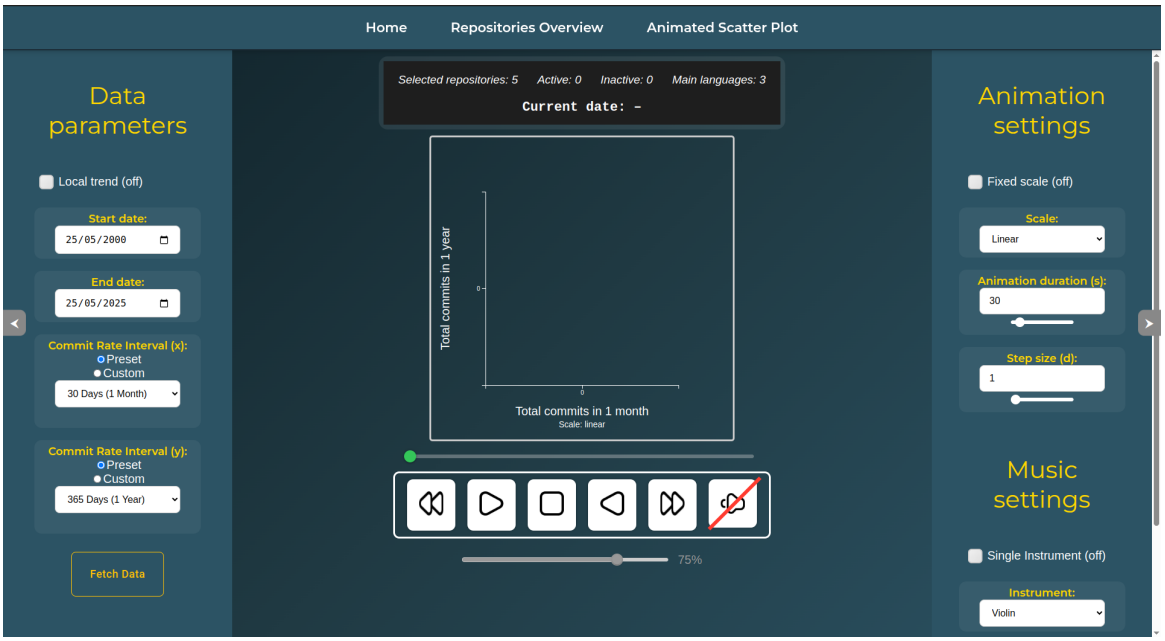


Figure 11. AnimatedScatterPlot view with playback controls, tooltips, and musical settings

- **Sidebars**: Each view has collapsible left and right sidebars for filters, settings, and actions. These are implemented in reusable components, apart from the home page.
- **CenterPlot and CenterPlotHeader**: Render the SVG plot, tooltips, timeline controls, and sound settings in a responsive layout.

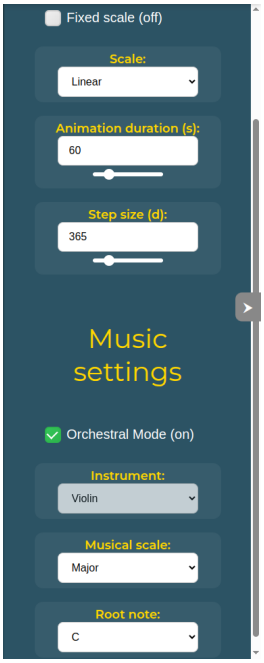


Figure 12. Sidebar controls for animation and sonification configuration

Repository Overview Page This view allows users to explore repositories using:

- **Filtering** by name, language, and start/end year;
- **Sorting** on all columns (except language);
- **Saving selections** that persist across navigation;
- **Virtual scrolling** for efficient rendering of large datasets.

Sidebars allow users to reset filters, save selections, or show all repositories again. Clicking a repository's URL opens it in a new tab.

Animated Scatterplot Page This is the most interactive part of the system, allowing visual and auditory inspection of commit dynamics over time. Each repository is shown as a moving circle in a D3 scatterplot.

- **Motion**: Circles move with cubic Bézier curves to simulate a "dancing" animation style.
- **Tooltip and Interactivity**: Tooltips show details on hover (only when paused), and clicking a circle opens the GitHub repository.
- **Axes and Scaling**: Users can select between linear, square root, and logarithmic scaling. Optionally, axes can auto-rescale during animation.
- **Trend Configuration**: The user selects between *global trends* (computed over the entire dataset) or *local trends* (within a custom date interval).
- **Animation Settings**: Users configure total duration, step size (days per frame), and sliding window size (X and Y).

Sonification Controls Users can explore repository trends through an interactive sonification panel:

- **Pitch Mapping**: Each X-axis value is translated into a musical note based on the selected scale, root note, and instrument-specific octave range.
- **Volume Mapping**: Each Y-axis value determines the note's volume, scaled logarithmically for perceptual consistency.
- **Audio Settings**: Users can toggle sound on/off, select an instrument (or activate orchestral mode), adjust volume via a slider, and configure musical parameters such as root note and scale.

Responsiveness and Routing The application uses a dynamic grid layout that adapts to sidebar visibility. Routing is implemented via `react-router-dom` [17], allowing seamless navigation between:

- Home
- Repositories Overview
- Animated Scatterplot

A persistent top navigation bar enables quick access to these pages.

Data Synchronization All state changes—such as repository selection, date intervals, or window sizes—trigger REST API calls to the backend, ensuring synchronized and up-to-date views. Data is re-fetched whenever configuration parameters change.

Sonification Module The system includes an advanced sonification layer that converts commit activity data into musical audio, enriching the visual animation with an auditory dimension. This module is highly customizable and provides nuanced control over the generated sound:

- **Pitch Mapping**: Each X-axis value is mapped to a specific musical note according to the selected *scale* (e.g., major, minor), *root note* (e.g., C, D#), and an octave range (e.g., 4–6). This mapping is handled dynamically to ensure harmonic coherence, allowing multiple repositories to play different notes simultaneously.
- **Volume Mapping**: Y-axis values are converted to audio volume using a logarithmic scale to better align with human auditory perception. This prevents lower commit activity from being inaudible and reduces harsh jumps in loudness.
- **Instrument Assignment**: In *orchestral mode*, active repositories are automatically assigned orchestral instruments (e.g., violin, trumpet, tuba) to reflect an ensemble performance. If disabled, all repositories use the same user-selected instrument.

- **Stereo Panning**: When multiple repositories are active, their sounds are spatially distributed across the stereo field to improve clarity and realism.
- **Note Filtering**: To reduce cacophony, only the most prominent repository per note (i.e., highest Y value) is allowed to play that note during each frame.

This feature offers a complementary channel for perceiving patterns, trends, and outliers, particularly useful in dense datasets or when visual information alone may be overwhelming.

6 Usage scenarios

There will be three usage scenarios. All of them have a dedicated YouTube video, but only one is fully documented in this report with screenshots and explanations. The links to the video walkthroughs are provided below:

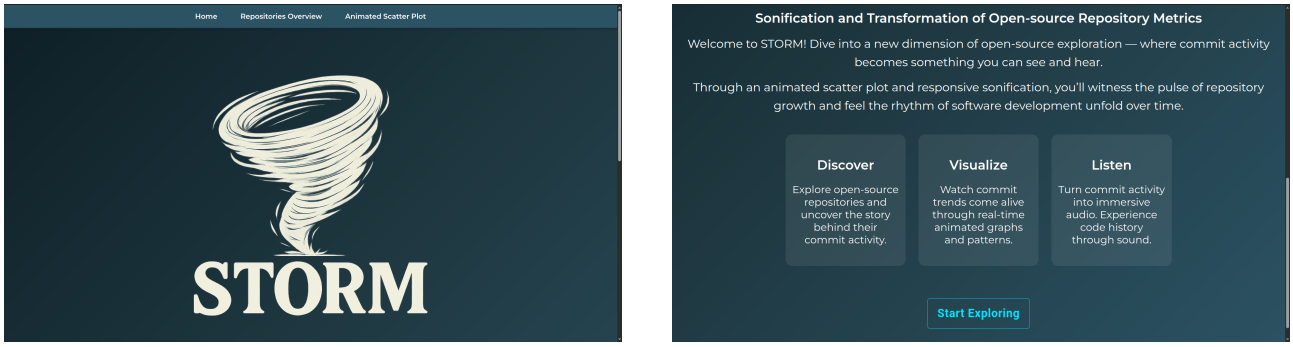
- **The Five Faces of Open-Source Evolution**: `https://youtu.be/gSySB7Cy61s`
- **The Life of a Star**: `https://youtu.be/T9kC-eL8YYY`
- **Infinity War**: `https://youtu.be/l0UVqcc9Q_k`

6.1 Complete Usage Scenario: The Five Faces of Open-Source Evolution

The following usage scenario, titled **"The Five Faces of Open-Source Evolution"**, showcases the full capabilities of the STORM application. It includes step-by-step explanations and annotated screenshots to illustrate how users can interact with the app to explore the lifecycle of a highly active repository over time.

6.1.1 Homepage

When a user first accesses the **STORM Web App**, they are greeted by a clean and minimalist homepage, prominently featuring the **STORM logo** and a set of cards that introduce the main features of the application. This immediate visual branding sets the tone for the user experience, as shown in Figure 13.

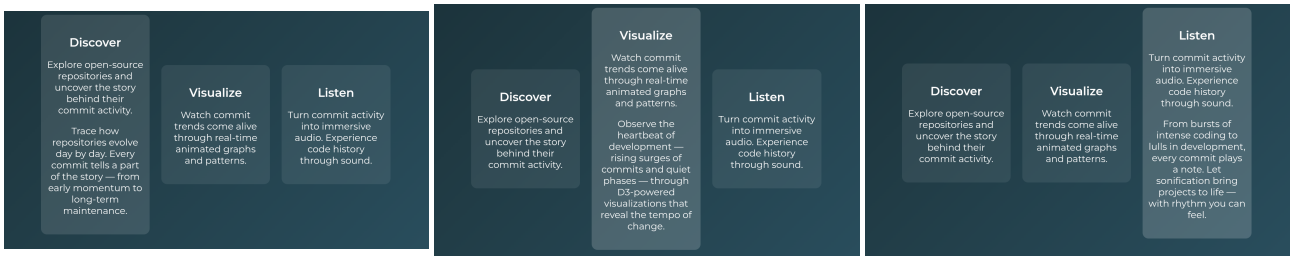


(a) STORM logo

(b) Start Exploring section with feature cards

Figure 13. Homepage layout: branding and entry point to exploration

Clicking on any card expands it to reveal more detailed information about the corresponding feature, including how it works and the insights users can gain.



(a) First card expanded

(b) Second card expanded

(c) Third card expanded

Figure 14. Expanded views of the three HomePage cards: Discover, Visualize, and Listen

Once the user has reviewed the features, they click the **"Start Exploring"** button to proceed to the repository exploration interface—the core of the application.

6.1.2 Repository Overview

The user is presented with an interface displaying the entire dataset:

(a) Sidebars open

(b) Sidebars closed

Figure 15. Comparison of repository overview with sidebars open and closed

In this scenario, the user investigates the five oldest repositories in the dataset using orchestral sonification and long-term commit trends. By clicking the *Oldest Commit Date* column header, the user sorts repositories in ascending order and selects the first five entries:

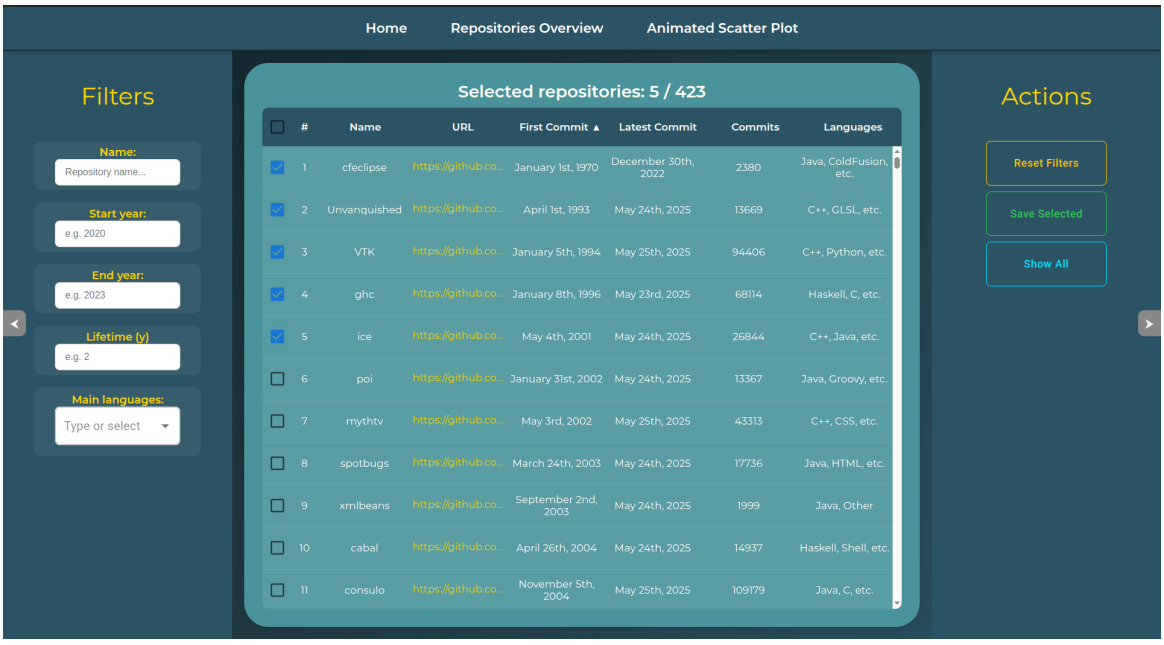


Figure 16. Selection of the five oldest repositories

After making the selection, the user clicks the *Save* button, which stores the selected repositories and automatically redirects them to the animated scatterplot view:

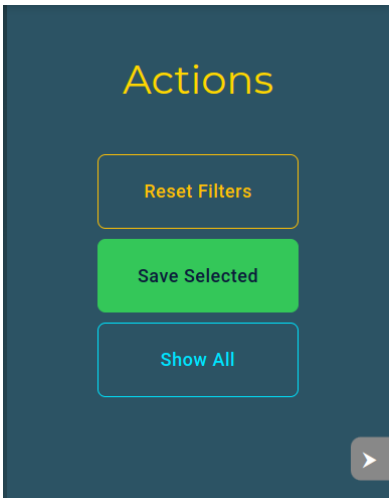


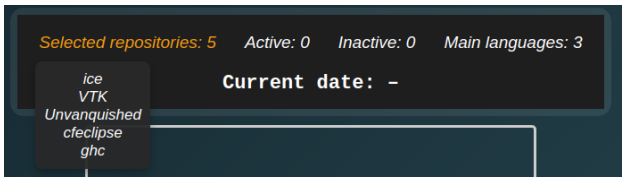
Figure 17. Action buttons

6.1.3 Animated Scatterplot

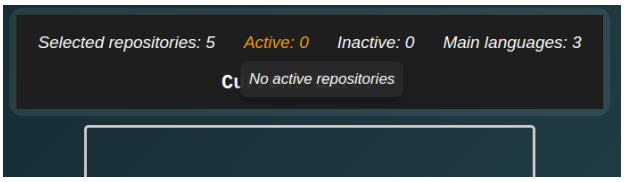
Upon redirection, the user sees the animated scatterplot interface:

Figure 18. Animated scatterplot with sidebars open

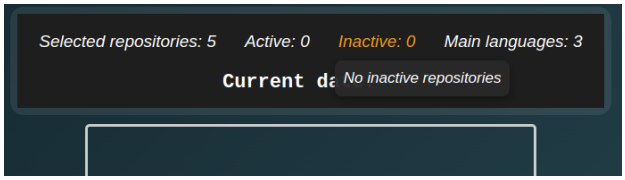
The user hovers over the four summary trackers in the header—*Selected Repositories*, *Active*, *Inactive*, and *Main Languages*—to gain a clearer understanding of the visualization context:



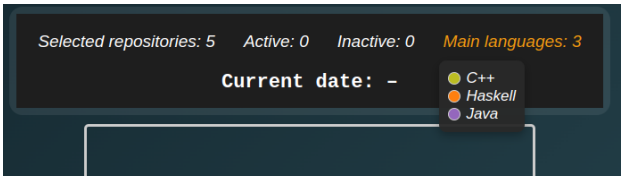
(a) Tooltip: Selected repositories



(b) Tooltip: Active repositories



(c) Tooltip: Inactive repositories



(d) Tooltip: Main languages

Figure 19. Header tooltips for visual summary of selected, active, inactive, and main language repositories

Next, the user configures the following settings via the sidebars:

- **Trend Mode:** Local
- **Date Range:** From `25/05/2000` to `25/05/2025`
- **X-Axis Commit Rate:** 365 days
- **Y-Axis Commit Rate:** 3650 days
- **Scale Type:** Fixed
- **Visualization Scale:** Linear
- **Animation Duration:** 60 seconds
- **Step Size:** 365 days
- **Sonification Mode:** Orchestral (instrument selection disabled)
- **Musical Scale:** Major
- **Root Note:** C

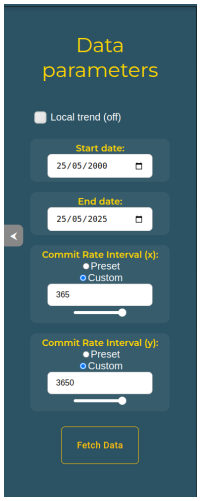


Figure 20. Data parameters

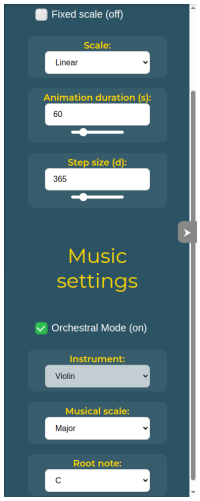


Figure 21. Animation and music settings

The user closes both sidebars to maximize the plot area and clicks the **“Fetch Data”** button:

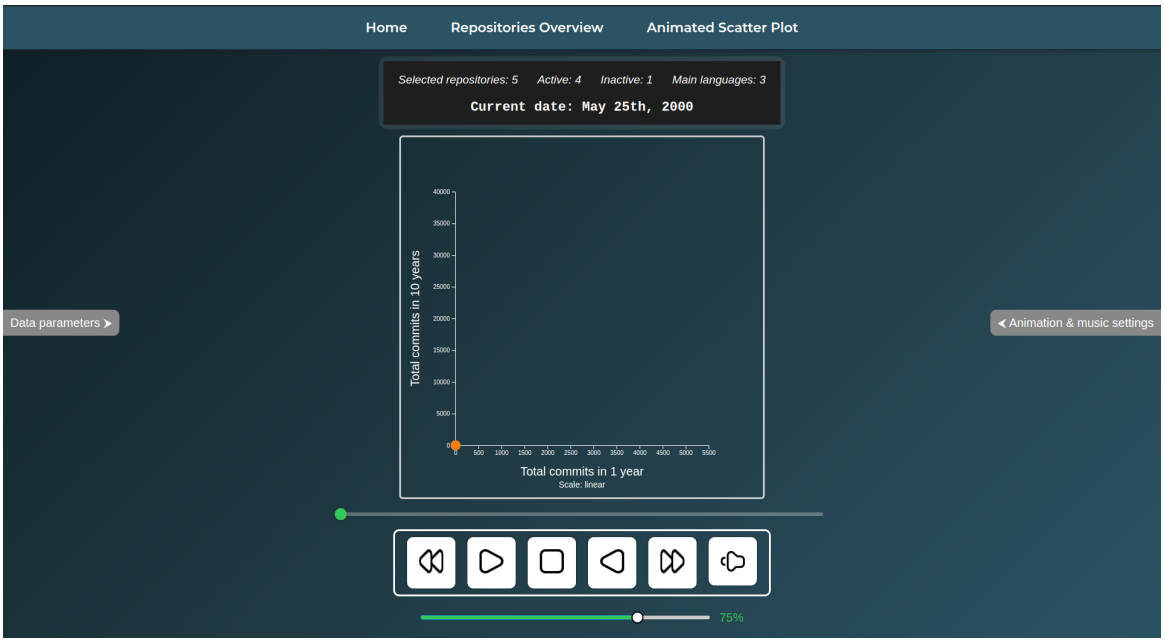


Figure 22. Full-screen plot view ready for animation

Then, the user starts the animation:

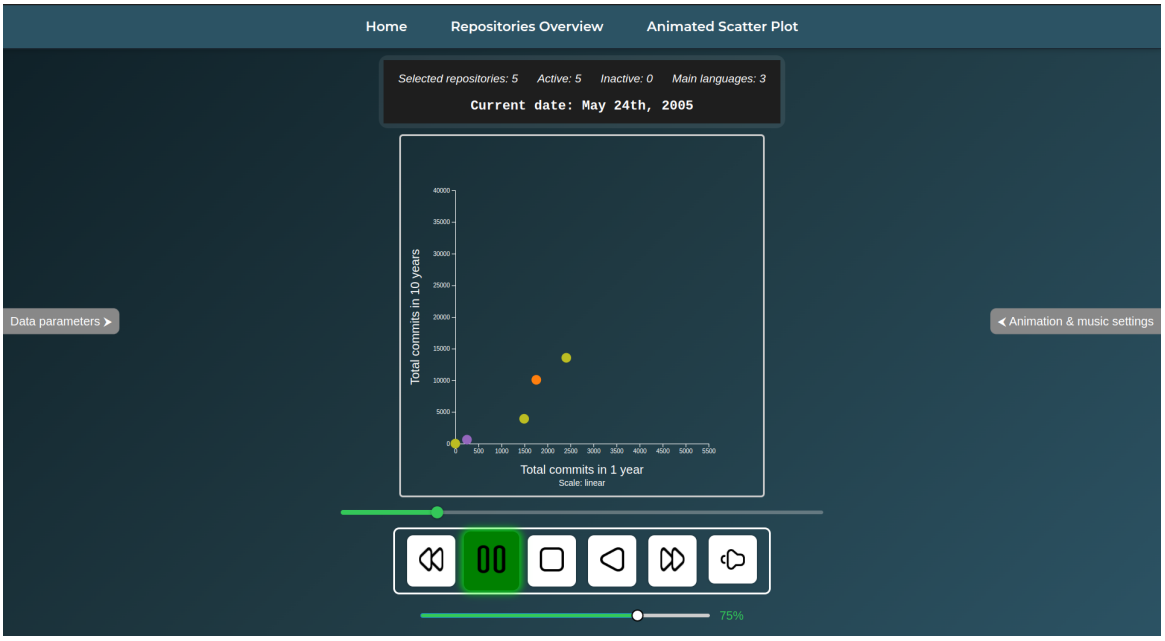
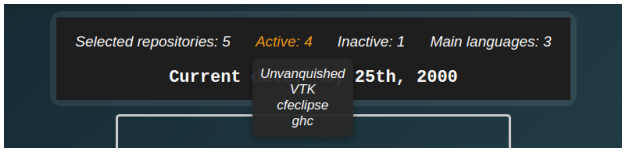


Figure 23. Running the animation

At one point, when a quieter repository becomes inactive, the user inspects the *Inactive* tracker to identify which repository just died:



(a) Updated inactive tracker

(b) Inspecting the tooltip

Figure 24. Inspecting which repository became inactive after a drop in activity

Finally, the user plays the animation in reverse to revisit earlier repository states:

Figure 25. Reversing the animation

6.2 Usage Scenario 2: The life of a Star

In this scenario, the user explores the commit activity of a single repository over a four-month period using both visual and auditory cues.

1. **Repository Selection**
   In the *Repositories Overview* view, the user filters the list by typing `postgres-language-server` into the name search bar. The matching repository is selected and saved. Upon clicking the *Save* button, the user is automatically redirected to the *Animated Scatter Plot* view.

2. **Settings Configuration**
   The user configures the following parameters:

   - **Trend Mode:** Global (Local trend OFF)
   - **Date Range:** From `25/11/2024` to `25/03/2025`
   - **X-Axis Commit Rate:** 1 day
   - **Y-Axis Commit Rate:** 30 days
   - **Scale Type:** Fixed
   - **Visualization Scale:** Linear
   - **Animation Duration:** 61 seconds (to avoid warning for non-divisible duration)
   - **Step Size:** 1 day
   - **Sonification Mode:** Single instrument
   - **Instrument:** Trumpet
   - **Musical Scale:** Balinese
   - **Root Note:** C#/Db

3. **Interaction with the Animated Visualization**
   The user closes both the filter and settings sidebars to maximize the plot area. The animation is started. Upon noticing that the audio is too soft, the user increases the *volume slider*. At a certain point, the user *pauses* the animation, rewinds to earlier frames using the *animation slider*, and *resumes* playback from that point. Once the animation reaches the end of the selected date range, the user *plays it backward*, occasionally pausing to *hover over the repository circle* and view detailed tooltips. Finally, the animation continues in reverse until it reaches the start date.

6.3 Usage Scenario 3: Infinity War

In this scenario, the user explores the long-term commit trends of multiple repositories using an orchestral sonification mode and a linear visual scale.

1. **Repository Selection**
   In the *Repositories Overview* view, the user filters the list by selecting `Java` and `JavaScript` as the main languages. After reviewing the filtered list, the user saves the selection. Upon clicking the *Save* button, the application automatically redirects to the *Animated Scatter Plot* view.

2. **Settings Configuration**
   The user configures the following parameters:

   - **Trend Mode:** Global (Local trend OFF)
   - **Date Range:** From `25/05/2018` to `25/05/2025`
   - **X-Axis Commit Rate:** 1 month
   - **Y-Axis Commit Rate:** 1 year
   - **Scale Type:** Auto-rescale
   - **Visualization Scale:** Linear
   - **Animation Duration:** 30 seconds
   - **Step Size:** 60 days
   - **Sonification Mode:** Orchestral mode
   - **Musical Scale:** Blues
   - **Root Note:** D#/Eb

3. **Interaction with the Animated Visualization**
   With both sidebars closed to maximize viewing area, the user starts the animation and listens as different orchestral instruments represent the activity of various repositories. Once the animation completes, the user hovers over several repository circles to inspect detailed tooltips showing commit information and metadata. Curious to revisit earlier activity, the user scrubs the animation slider to the 3/4 mark and plays the animation in reverse to observe how the commit patterns evolved.

7 Conclusions

STORM introduces a novel, multimodal approach to analyzing software repository evolution by integrating animated visualizations with real-time sonification. Through an interactive UI and a robust backend, it enables users to perceive commit dynamics not just visually but also auditorily—offering an alternative lens to understand temporal behavior across repositories.

The project successfully demonstrates how static commit histories can be transformed into engaging, perceptual experiences. Its strengths lie in the balanced dataset construction, the extensible modular architecture, and the range of supported user interactions—from filtering and sorting to detailed configuration of animation and sound parameters.

However, several limitations remain. Sonification, while functional, still requires refinement to achieve musicality and clarity when multiple repositories are active. Instrument loading times, volume balancing, and auditory overload under high density need further optimization. Similarly, the scatterplot, while expressive, may become visually cluttered when too many repositories are shown simultaneously.

Despite these challenges, STORM opens up promising directions for future work. Potential extensions include clustering similar repositories for comparative analysis, introducing rhythm- or harmony-based sonification, and supporting timeline annotation for event correlation. As a research and educational tool, STORM encourages rethinking how software activity can be perceived—bridging the gap between data analysis and perceptual insight.

References

[1] Andrew Caudwell. Gource - software version control visualization. https://github.com/acaudwell/Gource, 2025. Accessed: 2025-05-21.

[2] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 837–841, 2018.

[3] Linus Torvalds and Junio C. Hamano. Git: Fast version control system. https://git-scm.com/, 2025. Accessed: 2025-05-29.

[4] Brian Vaughn. react-window: React components for efficiently rendering large lists and tabular data. https://github.com/bvaughn/react-window, 2019. Accessed: 2025-05-13.

[5] GitHub. Search - github rest api v3. https://docs.github.com/en/rest/search, 2025. Accessed: 2025-06-02.

[6] Mike Bostock. D3.js - data-driven documents. https://d3js.org/, 2025. Accessed: 2025-06-04.

[7] GitPython Developers. Gitpython: Python git library. https://gitpython.readthedocs.io/, 2025. Accessed: 2025-05-13.

[8] Leonard Richardson. Beautiful soup documentation. https://www.crummy.com/software/BeautifulSoup/, 2025. Accessed: 2025-05-13.

[9] John D. Hunter. Matplotlib: Visualization with python. https://matplotlib.org/, 2025. Accessed: 2025-05-13.

[10] Michael Waskom and contributors. Seaborn: Statistical data visualization. https://seaborn.pydata.org/, 2025. Accessed: 2025-05-13.

[11] Inc. Pivotal Software. Spring boot. https://spring.io/projects/spring-boot, 2025. Accessed: 2025-05-13.

[12] Project Lombok. Project lombok. https://projectlombok.org/, 2025. Accessed: 2025-05-13.

[13] FasterXML. Jackson: Json for java. https://github.com/FasterXML/jackson, 2025. Accessed: 2025-05-13.

[14] QOS.ch. Slf4j: Simple logging facade for java. http://www.slf4j.org/, 2025. Accessed: 2025-05-13.

[15] JUnit Team. Junit 5. https://junit.org/junit5/, 2025. Accessed: 2025-05-13.

[16] Inc. Meta Platforms. React - a javascript library for building user interfaces. https://reactjs.org/, 2025. Accessed: 2025-05-13.

[17] Remix Software. React router: Declarative routing for react.js. https://reactrouter.com/, 2025. Accessed: 2025-05-13.