GitHop Software Requirements Specification

Version 1.0

Imad BOUTBAOUCHT , Ilias SKIRIBA

Supervised by: Professor Imane FOUAD

October 4, 2025

Contents

1 Introduction

1.1 Purpose of this Document

This document provides a detailed description of the Software Requirements Specification (SRS) for the **GitHop** application. It is intended to be used by developers, project managers, testers, and stakeholders to understand the system's functionalities, constraints, and intended behavior. The SRS serves as a foundation for the design, implementation, and testing phases of the project.

1.2 Scope of the Project

The GitHop application will be a web-based platform designed to aggregate and visualize trending activity within the GitHub developer community. Its core purpose is to help users discover **the pulse of what devs are working on recently** by showcasing:

- The top explored repositories recently.
- The top growing new repositories.
- The most active developers in specific fields.
- The most popular Topics/Fields (e.g., AI tools, cybersecurity tools).

The system will present this information through a unified, scrollable Social Feed user experience, moving beyond GitHub's native trending page to provide more nuanced metrics and rankings. This version utilizes both the GitHub REST API and GraphQL API to optimize data fetching strategies.

1.3 Intended Audience and Reading Suggestions

This document is intended for:

- **Developers:** To understand what to build.
- **Project Managers:** To plan project timelines and resources.
- **Testers:** To create test cases and validation plans.
- **Stakeholders:** To review and agree upon the system's capabilities.

It is suggested that readers start with the General Description and System Features sections for a high-level overview before proceeding to the detailed requirements.

2 General Description

2.1 Product Perspective

GitHop is envisioned as a standalone web application that interfaces with both the GitHub REST API and the GitHub GraphQL API. It will process raw data from these APIs to calculate custom metrics and rankings, presenting them in an

engaging, social-media-like format. The hybrid API approach allows us to use each API for its strengths: GraphQL for complex relational queries and REST for simpler bulk data fetching or when specific endpoints are only available via REST.

2.2 User Characteristics

The primary users of this system are expected to be:

- **Developers and Tech Enthusiasts:** Individuals looking to discover new tools, libraries, and trending projects.
- **Open-Source Contributors:** Developers seeking to identify popular projects to contribute to.
- **Recruiters and Technical Managers:** Professionals looking to identify top talent and emerging technologies.

These users are assumed to be technically proficient and familiar with platforms like GitHub.

2.3 Operating Environment

The application will be a responsive web application, accessible on modern web browsers across operating systems (Windows, macOS, Linux) and device form factors (desktop, tablet, mobile).

3 System Features

This section details the high-level features and their associated functional requirements.

3.1 Feature 1: Aggregate and Process GitHub Data

The system shall periodically fetch and process data from both the GitHub REST API and GraphQL API to calculate custom ranking metrics.

Table 1: Functional Requirements for Data Aggregation

Requirement ID	**Description**	**Priority**
FR-1.1	The system shall periodically fetch data from both the GitHub REST API and GraphQL API, selecting the optimal API for each data type.	High
FR-1.2	The system shall store and process fetched data to calculate metrics for ranking.	High
FR-1.3	The system shall handle GitHub API rate limiting strategically, considering REST's request limits and GraphQL's point-based system.	High

3.2 Feature 2: Display Top Repositories

The system shall identify and display various categories of top-performing repositories.

Table 2: Functional Requirements for Displaying Repositories

Requirement ID	**Description**	**Priority**
FR-2.1	The system shall display a list of repositories with the highest "exploration score" in the last 7 days. *Acceptance: Score = (Stars * 3) + (Forks * 2) + (Unique Contributors)*	High
FR-3.1	The system shall display a list of repositories created in the last 30 days with the highest growth velocity. *Acceptance: Velocity = (Stars Gained in Last 7 Days / Repository Age in Days)*	High

3.3 Feature 3: Display Developer and Topic Rankings

The system shall identify and rank developers and topics based on activity and popularity.

Table 3: Functional Requirements for Developer and Topic Rankings

Requirement ID	**Description**	**Priority**
FR-4.1	The system shall identify and display developers with the highest activity score in specific topics/-fields. *Acceptance: Score based on contributions (commits, PRs, issues) in the last 14 days.*	High
FR-5.1	The system shall identify and display the most popular topics. *Acceptance: Popularity based on frequency of topic use in newly starred repositories over the last 7 days.*	High

3.4 Feature 4: Social Feed User Experience

The system shall present all aggregated data in a unified, engaging feed.

3.5 Feature 5: Data Management and Caching Strategy

The system shall implement a hybrid data management approach to balance performance with data freshness, using strategic caching with scheduled updates.

Table 4: Functional Requirements for Social Feed UX

Requirement ID	Description	Priority
FR-6.1	The system shall present all data in a single, vertically scrollable feed.	High
FR-6.2	Each item in the feed (repo, developer, topic) shall be displayed in a consistent "card" format.	High
FR-7.1	Users shall be able to click on any card to be taken to its respective page on GitHub.	High
FR-7.2	Users shall be able to filter the feed by a primary topic/category.	Medium

Table 5: Data Management and Caching Requirements

Requirement ID	Description	Priority
FR-8.1	The system shall cache processed ranking data in the database and serve all user requests from this cache for optimal performance.	High
FR-8.2	The system shall periodically refresh cached data through background jobs, with update frequency based on data type volatility.	High
FR-8.3	Repository exploration data shall be updated every 15 minutes to maintain freshness of trending metrics.	High
FR-8.4	Developer activity data shall be updated hourly, as activity patterns change more gradually.	Medium
FR-8.5	Topic popularity data shall be updated every 30 minutes to reflect emerging trends.	High
FR-8.6	The system shall implement cache invalidation strategies to ensure users never receive stale data beyond the defined refresh intervals.	High
FR-8.7	The system shall provide a manual refresh option that prioritizes updating viewed content in the next background job cycle.	Low

4 Data Management and Caching Strategy

4.1 Architecture Overview

GitHop will implement a robust caching architecture where all user requests are served from pre-computed database cache, while background workers maintain data freshness through scheduled GitHub API updates.

4.2 Update Frequency Strategy

Different data types require different update frequencies based on their volatility and importance:

Table 6: Data Update Frequencies

Data Type	Update Frequency	Rationale
Repository Exploration Scores	Every 15 minutes	Stars and forks can change rapidly for trending repositories
New Repository Growth	Every 30 minutes	New repositories need frequent monitoring to catch rapid growth
Developer Activity	Hourly	Developer contribution patterns change more gradually
Topic Trends	Every 30 minutes	Emerging topics can gain popularity quickly
Historical Trends	Daily	Long-term trend analysis doesn't require minute-level updates

4.3 Performance Benefits

This caching strategy provides several advantages:

- **Fast Response Times:** User requests respond in milliseconds (database queries vs. API calls)
- **Rate Limit Management:** Background jobs can be spaced out to stay within GitHub API rate limits
- **Consistent Performance:** Database queries are more predictable than variable API response times
- **Offline Resilience:** The application can continue functioning during temporary GitHub API outages
- **Scalability:** Can handle thousands of concurrent users with minimal performance impact

5 API Integration Strategy

5.1 Hybrid API Approach

GitHop will utilize both GitHub's REST API and GraphQL API to optimize performance and data completeness. The selection between APIs will be based on the specific data requirements and the strengths of each API.

5.2 Rate Limiting Considerations

The system will implement different strategies for handling API rate limits:

- **REST API:** Limited by number of requests (typically 5,000 requests per hour for authenticated users)
- **GraphQL API:** Limited by "points" (typically 5,000 points per hour), with query cost based on complexity

Table 7: API Selection Strategy for GitHop

Use Case	Preferred API	Rationale
Fetching specific repository metrics (stars, forks, issues)	GraphQL	Can retrieve multiple related data points in a single request, reducing network overhead.
Bulk listing of repositories or users	REST	More efficient for simple list endpoints where field selection is not complex.
Complex relational queries (e.g., repo issues with labels and assignees)	GraphQL	Avoids multiple round-trips (N+1 problem) inherent in REST.
Operations requiring specific endpoints only available in one API	REST or GraphQL as available	Some data and operations are exclusive to one API.
Background cache updates	Mixed approach	Use each API for its strengths in different update scenarios.

- The application will monitor rate limit status using the GraphQL `rateLimit` field and REST API headers, implementing appropriate backoff strategies and caching to maximize data collection within limits
- Background updates will be scheduled to avoid hitting rate limits during peak usage periods

6 Interface Requirements

6.1 User Interfaces

The primary UI will be a single-page web application with a central Social Feed. Key components include:

- **Header:** Application logo ("GitHop") and a topic filter dropdown
- **Main Feed:** A scrollable column of cards for repositories, developers, and topics with freshness indicators
- **Refresh Button:** Manual refresh option with visual feedback
- **Footer:** Standard links and system status information

7 Non-Functional Requirements

7.1 Performance Requirements

- The application homepage shall load in under 2 seconds

- User requests shall be served from cache with response times under 100ms
- Background data updates shall complete within their allocated time windows
- The system shall support at least 1,000 concurrent users

7.2 Usability

- The UI shall be intuitive, requiring no training for users familiar with social media or developer platforms
- The application shall be fully responsive on desktop, tablet, and mobile devices
- Data freshness shall be clearly indicated to manage user expectations

7.3 Reliability and Security

- The system shall have 99.5% uptime, excluding scheduled maintenance
- All communications shall be encrypted using HTTPS
- API credentials shall be stored securely and not exposed to clients
- The system shall gracefully handle GitHub API outages by serving cached data

8 System Models

8.1 High Level Interaction Diagram

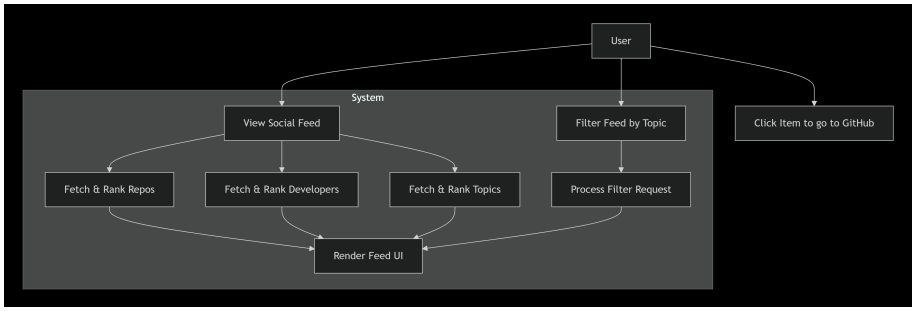

Figure 1: High Level Interaction Diagram.

8.2 Data Flow Diagram

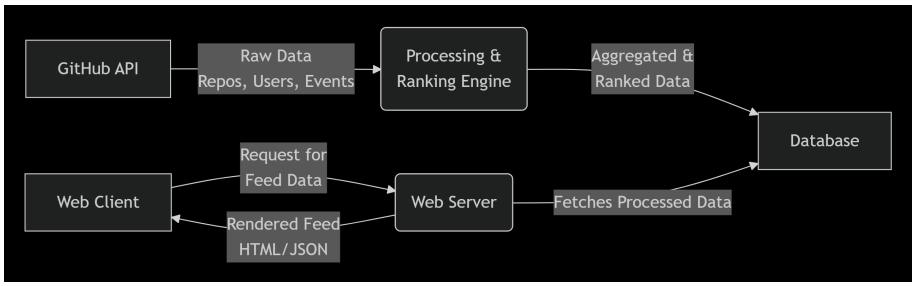

Figure 2: Data Flow Diagram.

8.3 Backend Process Diagram

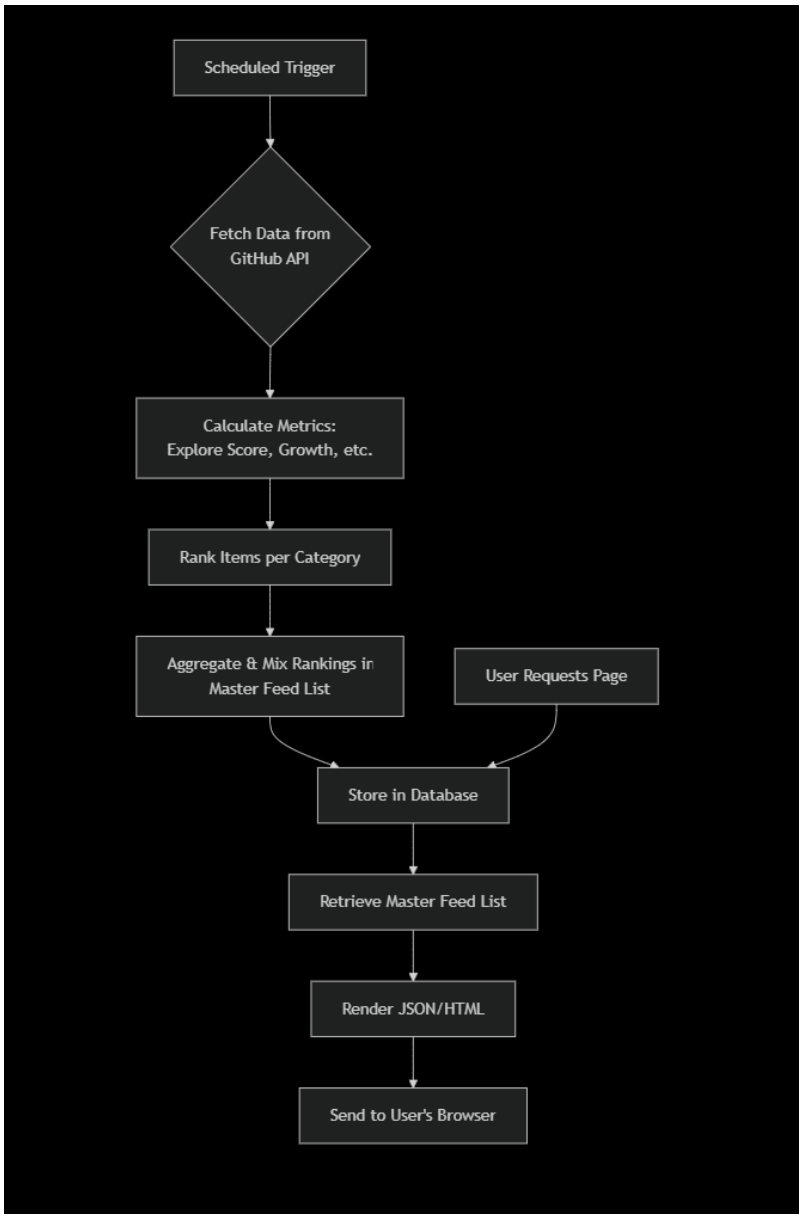

Figure 3: Backend Process Diagram.

9 Definitions, Acronyms, and Abbreviations

- **SRS:** Software Requirements Specification
- **API:** Application Programming Interface
- **UI/UX:** User Interface / User Experience
- **REST:** Representational State Transfer
- **GraphQL:** Graph Query Language
- **TTL:** Time To Live (cache expiration time)
- **CRON:** Command to schedule background jobs on Unix systems