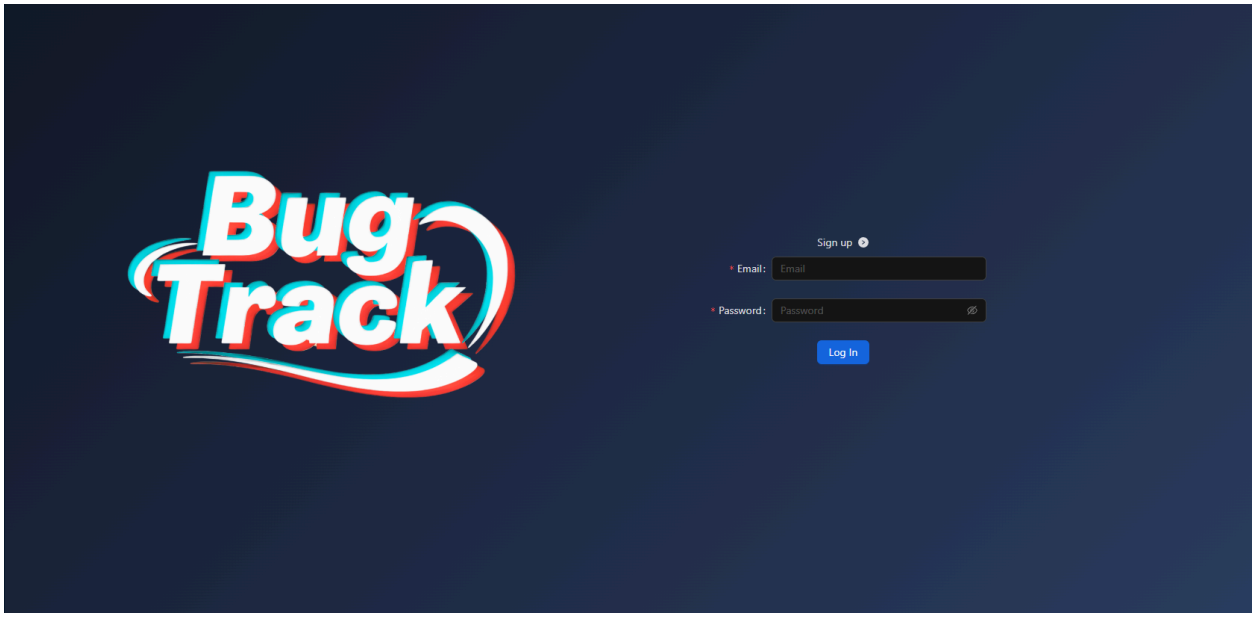
Group 10: BugTrack Documentation

User Guide

1. Authenticating

BugTrack requires users to log in before using its services.

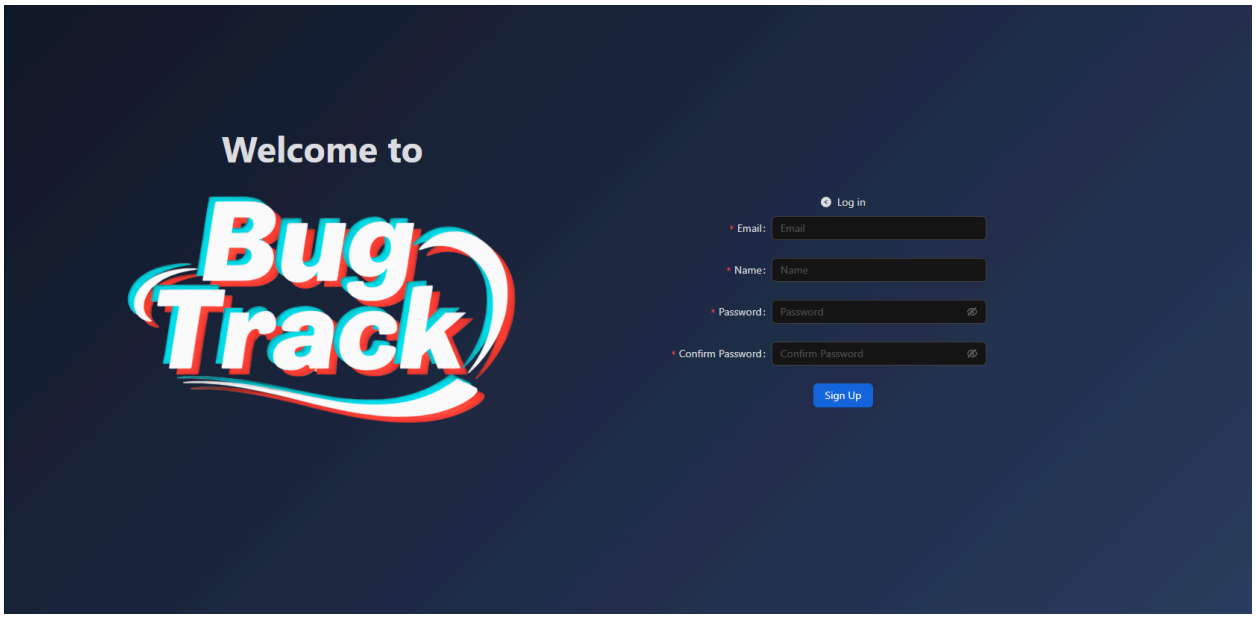
1.1. Logging In with /login



When loading BugTrack for the first time, you will likely be redirected to the login screen. If this is your first time using BugTrack, chances are that you have no account. Click the text that says “Sign up” to begin your account creation.

If you already have an account, you should be redirected on successful login, and there should be a small popup at the top of the screen informing you of the success.

1.2. Signing Up with /signup

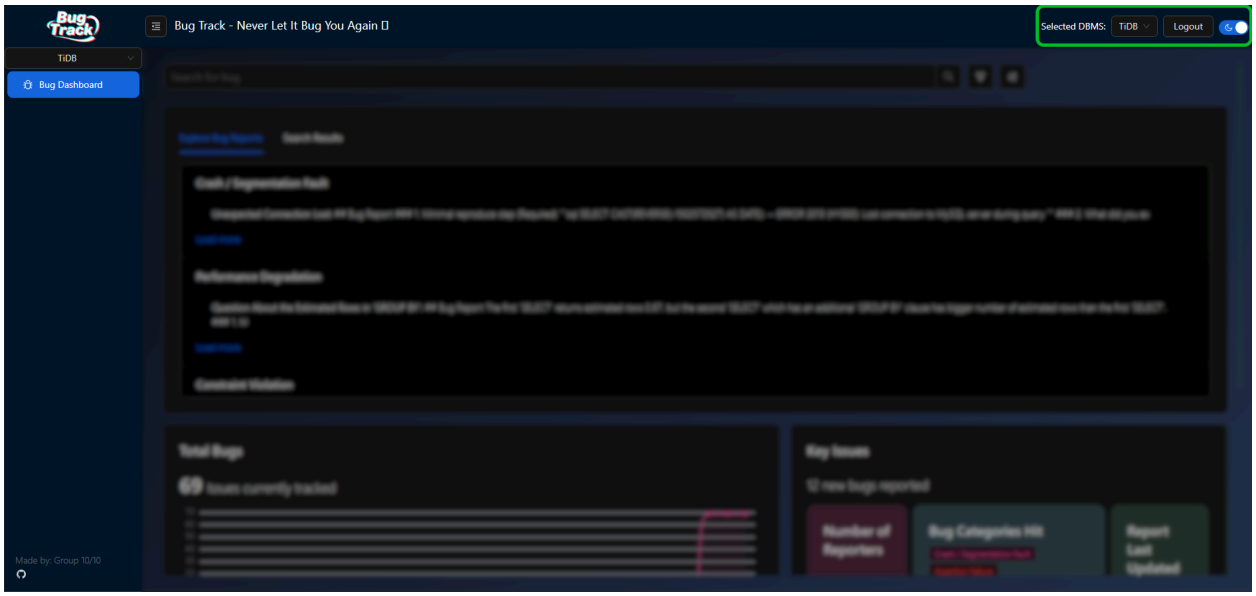


In the signup page, you can input account details for a new account. If you already have an account, click on the “Log in” text to go back to the login page.

Note that you cannot create an account with an email that you’ve already used. The name field is a display name that other users can see when you post comments in the bug discussion. The two password fields must match before you can proceed.

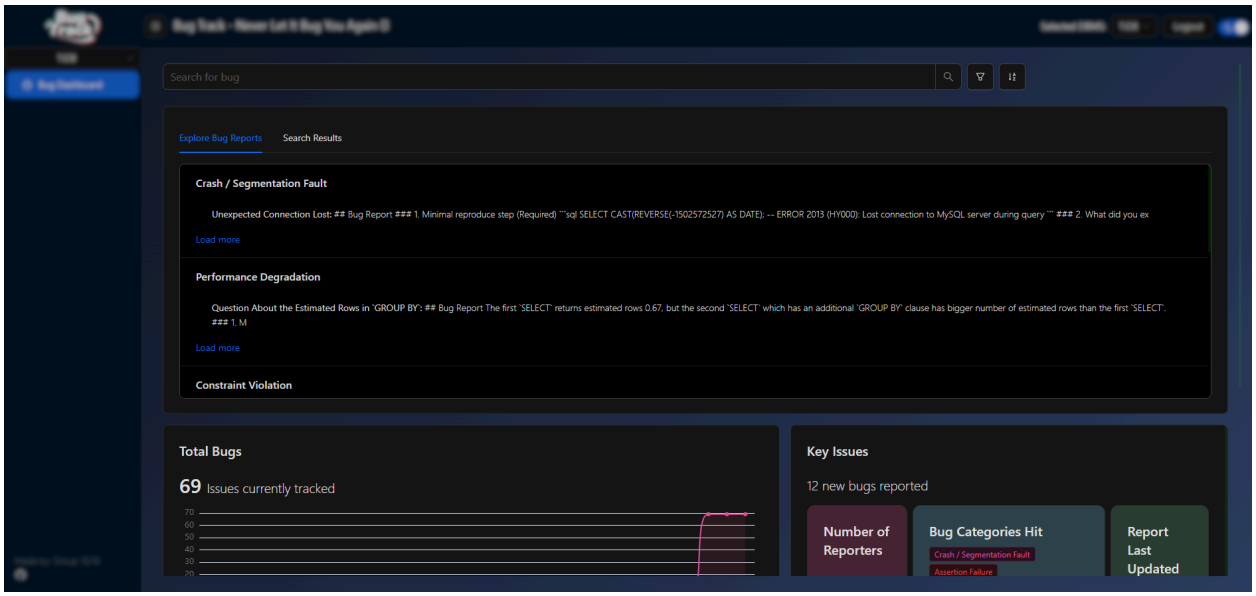
Successfully signing up will redirect you to the dashboard, and there should be a small popup at the top of the screen informing you of the success.

1.3. Selecting your DBMS and Logging Out



Now that you have logged in, the header should contain new options. You can now choose the currently viewed database from a dropdown list of databases you have access to, logout by clicking the respective button, or change between light and dark themes for the website by clicking the toggle element.

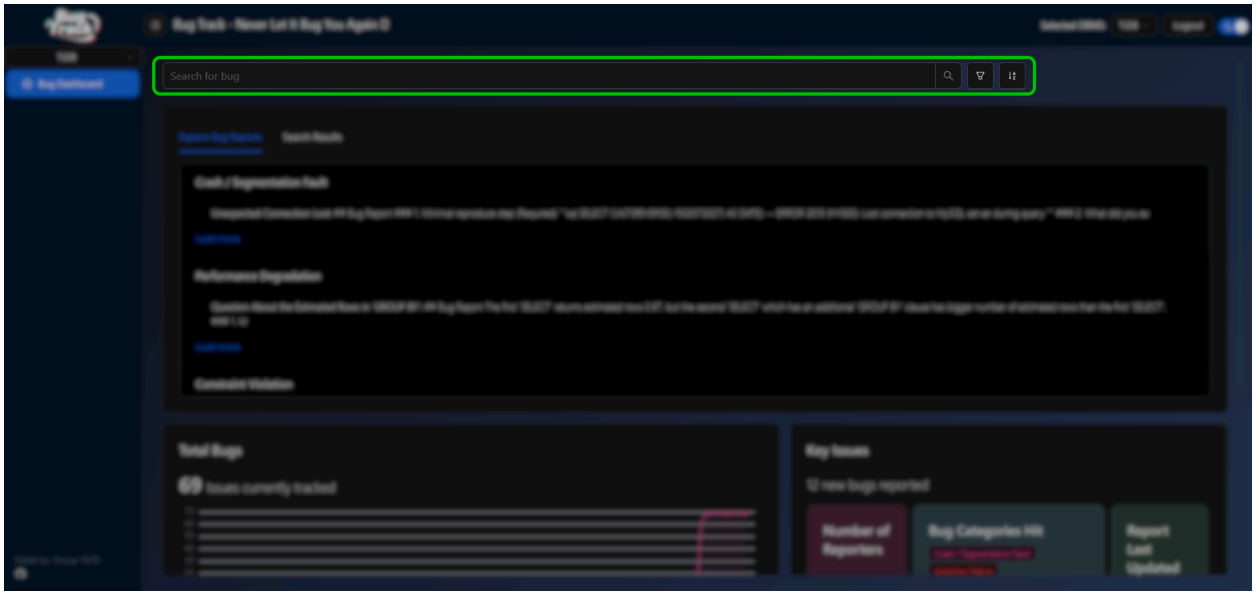
2. Analyzing your DBMS with the Dashboard



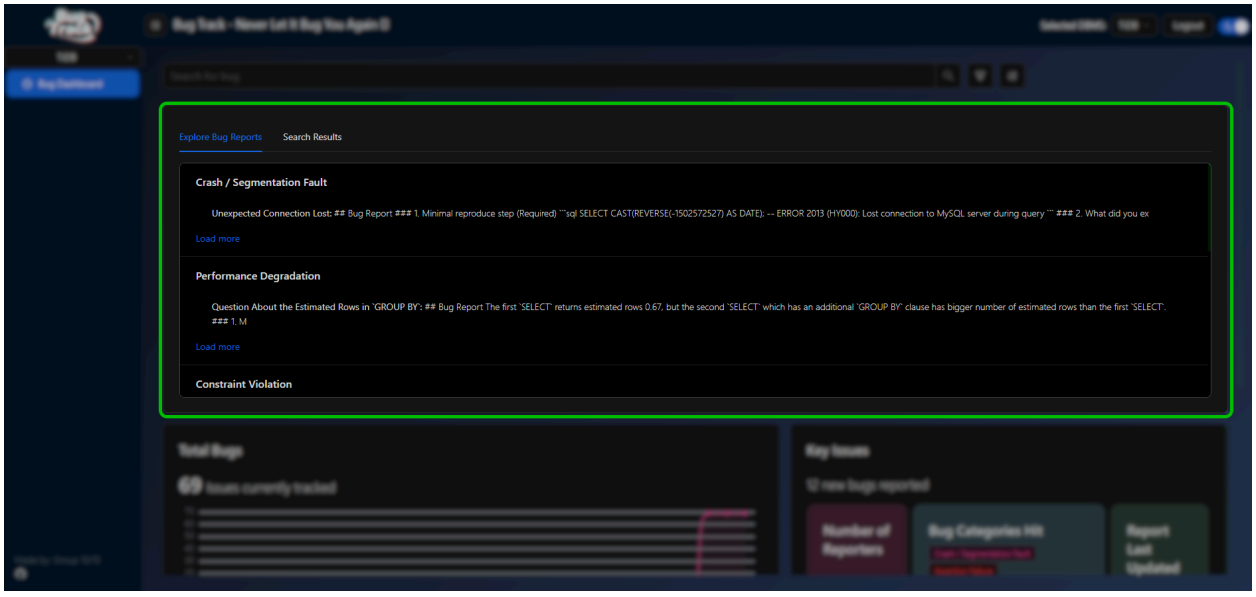
After logging in, you should land on BugTrack's Dashboard. The dashboard is meant to give you a sweeping overview of your entire DBMS, including any historical trends and key issues.

2.1. Searching for Bugs

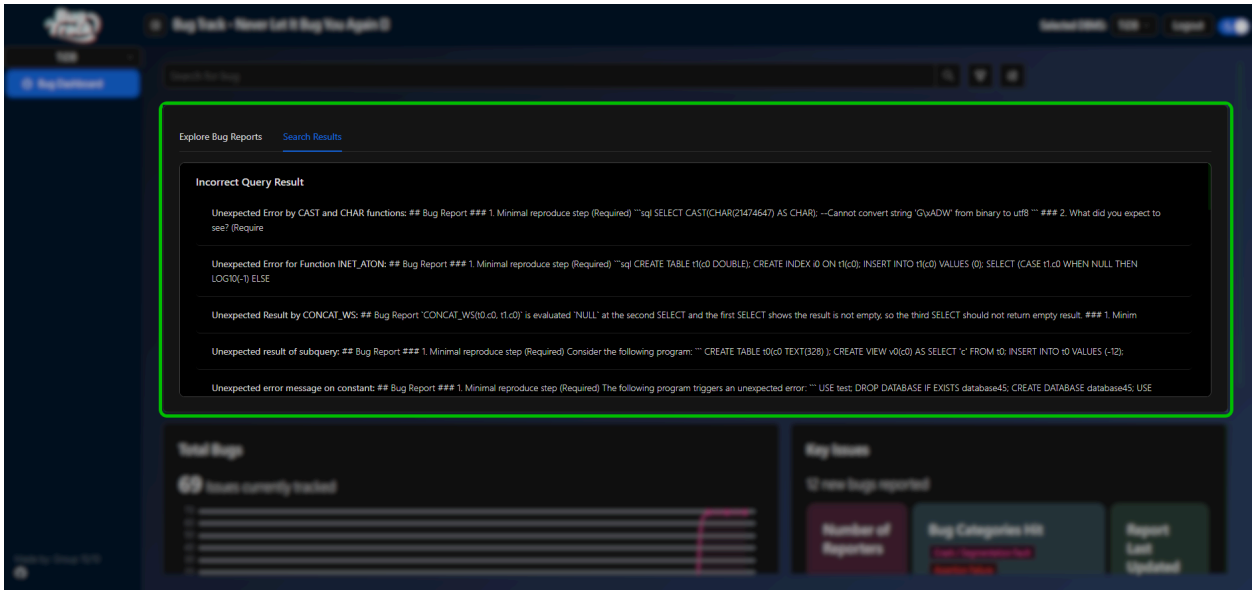
At the very top of the dashboard, a search bar allows you to search for keywords in the bug issue title or descriptions. In addition, you can use the funnel icon on the right to filter based on bug categories or priorities.



Before searching, the bug explore tab should be open. This allows you to immediately scroll and look through individual bug reports. Clicking on any entry will bring you to the bug detail page, where you can find additional information on the bug.



After searching, results will be shown in the search results tab. Results can be sorted by the rightmost button next to the search bar.



2.2. Inspecting Total Bugs and Bug Count History

This chart shows the number of total bugs over time. This can give you an idea of the increase in bugs found in your DBMS, and may be a signal of accumulating technical debt.

2.3. Viewing Key Issues

If the goal is to get a time-sensitive update on the current state of affairs, the Key Issues card will quickly bring you up to speed. It summarizes the number of reports for new issues, as well as the number of reporters for those issues. Since this update is time sensitive, this card also conveys when this information was validated.

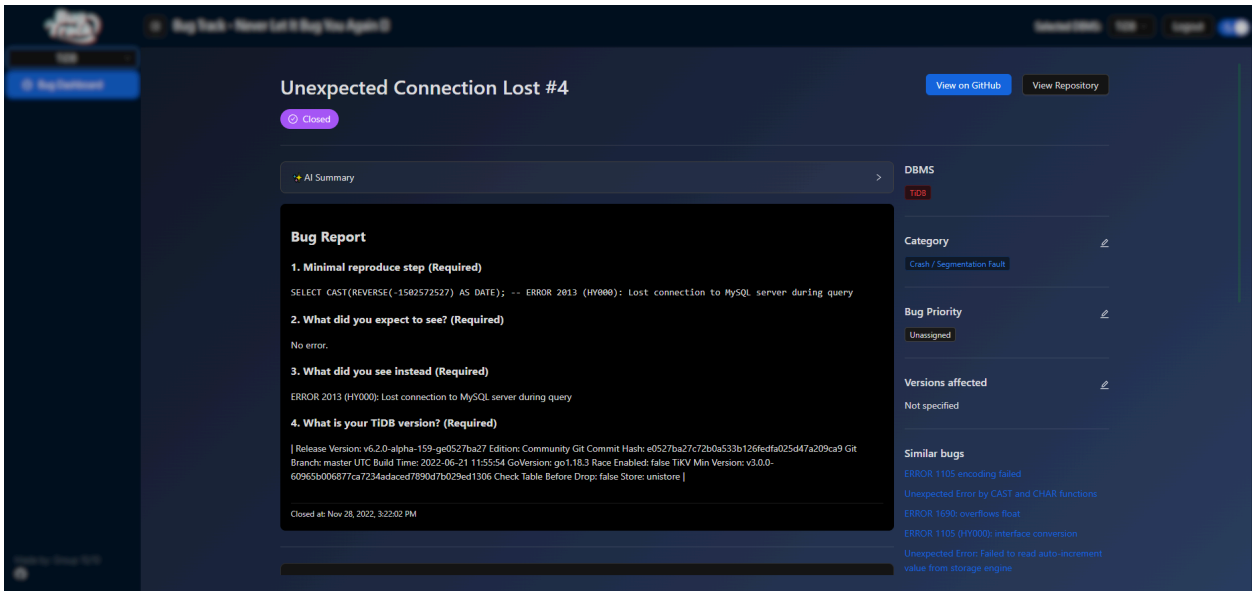
2.4. Viewing AI Summary

If you want more details about the issues involved, you can instead glance at the AI summary, where a large language model provides a concise report that mentions commonalities across the most critical bugs affecting your dbms.

2.5. Inspecting Bug Distribution by Category

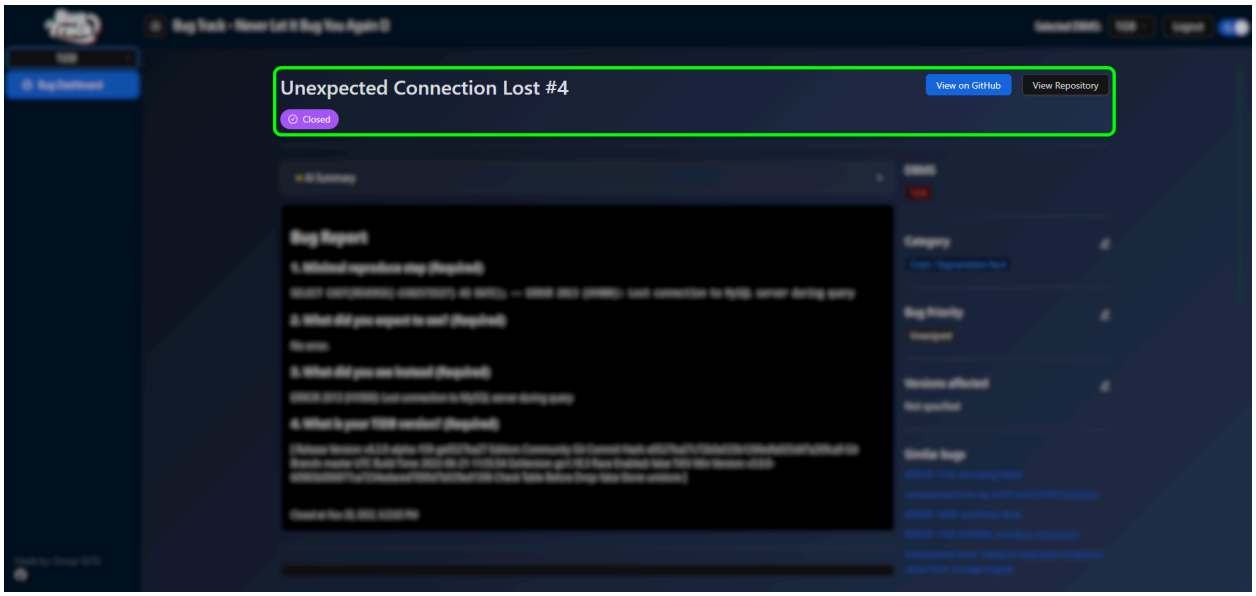
Alternatively, if you would like to see which types of bugs affect your dbms the most, you can take a look at the Bug Distribution card, which may give further insights for where to prioritize development efforts.

3. Inspecting Details of Bugs



Clicking on a bug entry from the dashboard will take you to the **Bug Report Page**, where you can view detailed information about the selected bug. This page is organized into several key sections to help you quickly understand the issue and take action.

3.1. Viewing the Bug Header

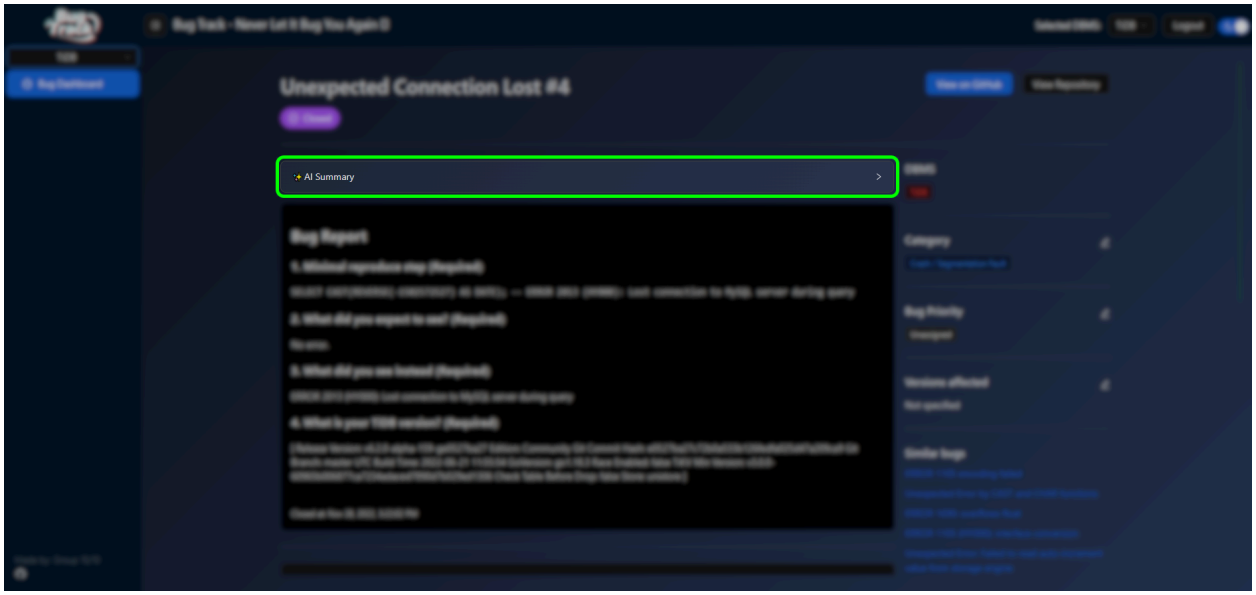


At the top of the page, the header provides a quick overview of the bug. It includes:

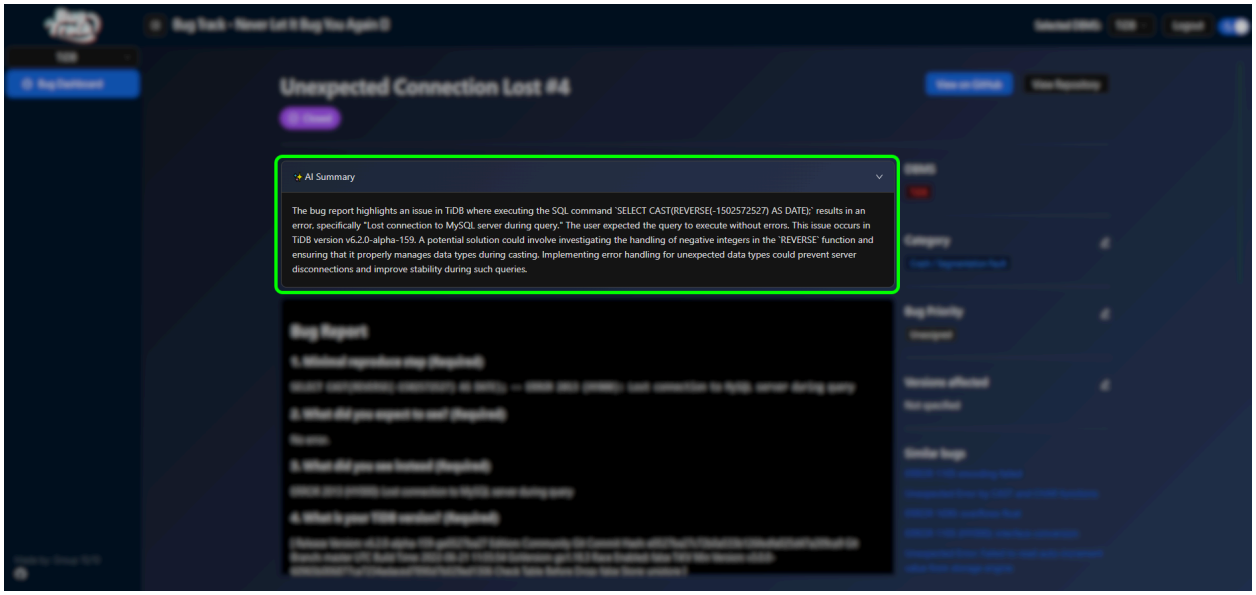
- Title - The name of the bug
- Status - Indicates whether the issue is open or closed.

- Github links - Direct links to the associated GitHub repository and issue so you can track progress and contribute from there.

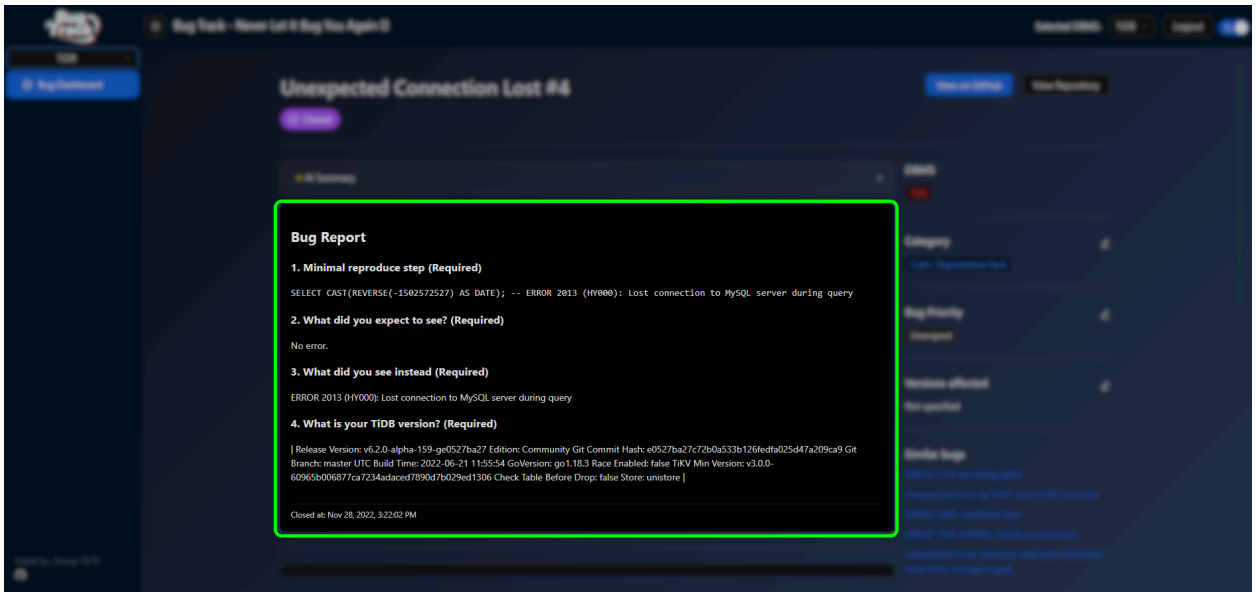
3.2. Viewing an AI Summary of the Bug



The **AI Summary** component offers a concise, AI-generated overview of the bug, along with a suggested solution. It is collapsed by default for a cleaner view, and you can click on it to reveal the AI overview. This can help you save time by understanding the core issue at a glance, especially for longer reports.



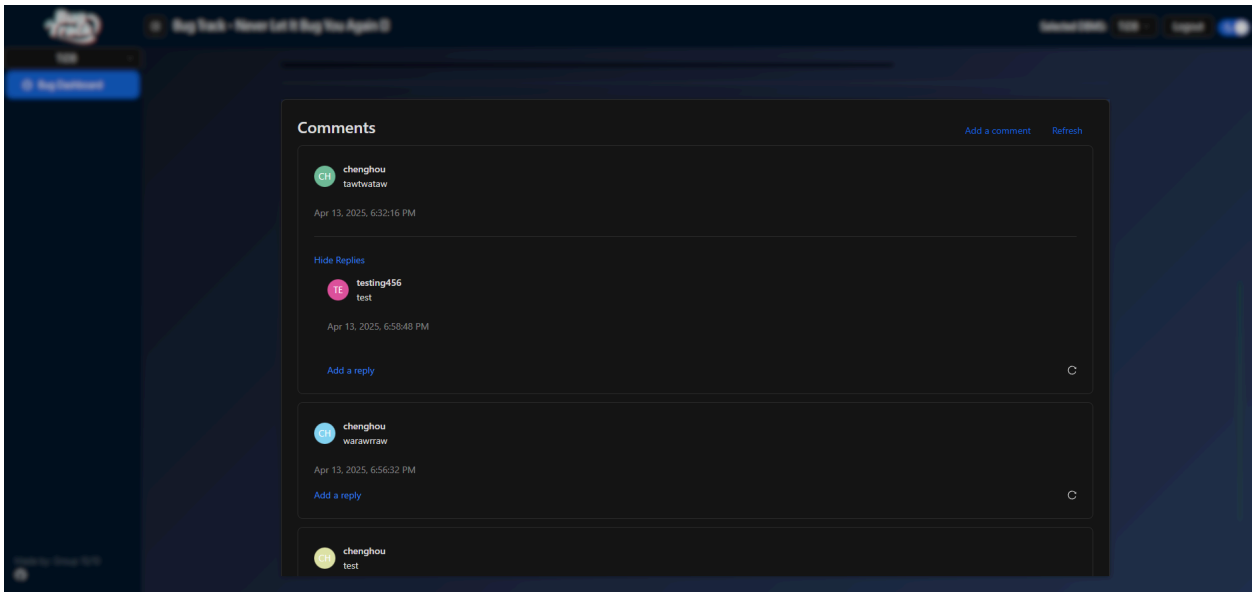
3.3. Viewing the Bug Description



This section contains the full description of the bug as submitted by the reporter. It may include:

- Steps to reproduce
- Expected vs. actual behaviour

3.4. Commenting on Bugs



Beneath the description, you can find the **comment section** where users can discuss the bug. In this section, you can:

- Post comments - Share insights, updates, or questions

- Reply to others - Engage in threaded discussions to keep conversations organised
- Markdown support - Format your responses for clarity using Markdown

3.5. Viewing and Editing Bug Metadata in the Sidebar

Located on the right-hand side, the **Sidebar** displays key metadata about the bug, including:

- DBMS
- Category
- Priority
- Versions affected

You can click the edit icon to modify the fields if the classification needs updating.

3.6. Viewing Similar Bugs in the Sidebar

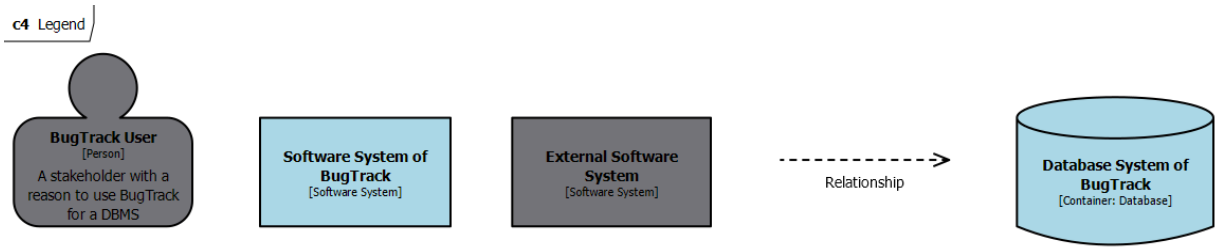
Just below the metadata section, you will find the **Similar Bugs** section. This displays a list of other bug reports that share similar traits to help you quickly find existing solutions. Click on any listed bug to navigate to its bug report page.

Developer Guide

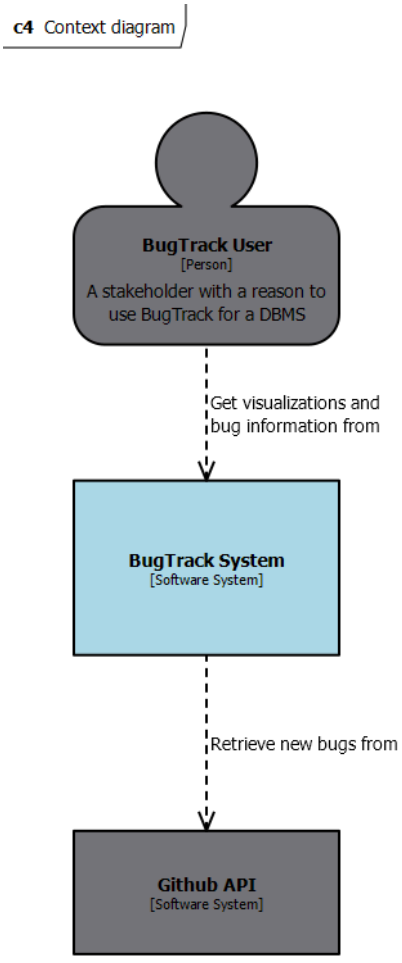
1. Overview

1.1. Diagrams Legend

The following legend applies for all C4 diagrams in this document.



1.2. System Context Diagram

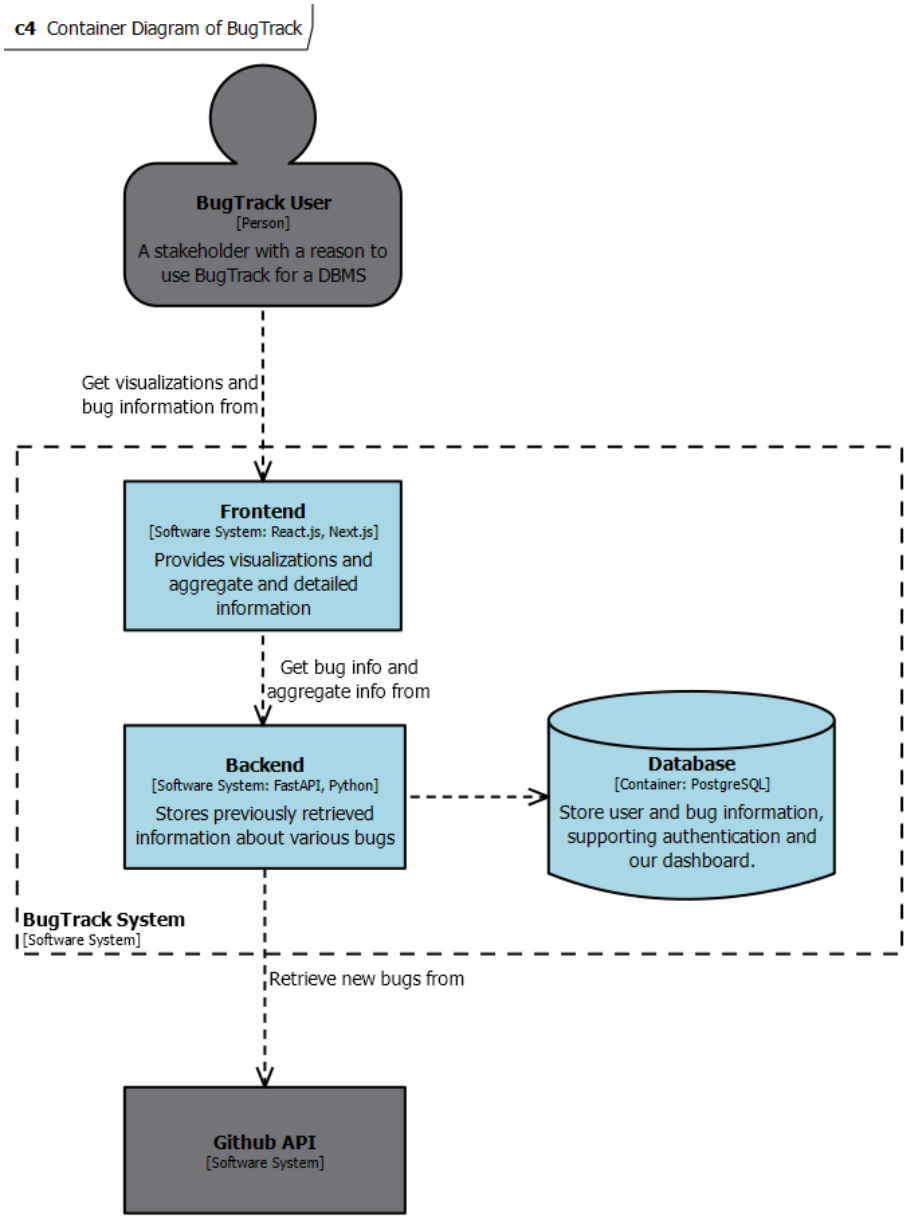


BugTrack retrieves bug information from public GitHub Repositories, and interacts directly with the user. We chose a monorepo, which is shown in this diagram.

Apart from this, BugTrack also calls the OpenAI API for its AI summary features in both the Dashboard and the Bug Detail Page; however this is relatively self-contained and is not a significant architectural decision, hence is not depicted in the big-picture overview here.

1.3. Container Diagram

Zooming into our monorepo, we have the frontend and backend, and a (separately hosted) PostgreSQL database. The Frontend is in Javascript and delivers interactive content to the user. The Backend provides updated bug information to the frontend, and handles requests to authenticate and edit. Finally, the Database stores authentication information about BugTrack users, as well as the latest snapshot of GitHub Issues provided by the Backend.



2. Deployment and Tooling

2.1. Tech Stack Overview

Our project is built using a modern, modular tech stack designed for scalability, maintainability, and developer efficiency. On the frontend, we use the React and Next.js framework and meta-framework respectively, in combination Ant Design as our UI library. For additional styling, we use TailwindCSS due to its simple, easy-to-use API. For our data visualization and charting library, we use Apache Echarts due to its declarative syntax as well as flexible and powerful charting features.

The backend is written in Python, specifically using the FastAPI framework. Python was chosen due to its extensive ecosystem of packages, especially those relating to data analytics as well as Artificial Intelligence (AI), Machine Learning (ML) and Natural Language Processing (NLP). Choosing Python as a backend language would allow for the most room for future improvements without having to resort to the additional complexity of multiple/micro-services, especially in a world that is seeing increasing use for AI. Given the choice of backend language, FastAPI is thus the API framework of choice, also due to its extensive ecosystem of plugins and extensions that support rapid feature development.

We also integrate OpenAI and Celery into our backend to support asynchronous and/or scheduled job/task execution and AI-driven features. Our PostgreSQL database is hosted on Neon (which internally uses AWS) for both development and production environments, enabling shared access and streamlined collaboration across the team. Of lesser-but-still-noteworthy importance is a Redis cache, which while not used directly, is required to act as a broker between the various task runner/worker threads that are spawned by Celery.

The frontend deployment is handled using Vercel at https://bug-track.projects.richarddominick.me/, while the backend is hosted on Render at https://api.bug-track.projects.richarddominick.me/, alongside the various supporting infrastructure as mentioned above.

2.2. Tooling and Code Maintainability

To maintain code quality and consistency, we apply formatters and linters throughout the codebase. For our frontend (TypeScript) codebase, we use the two most popular formatters and linters available – Prettier and ESLint. This ensures that most developers, as well as future developers who may take over the project, would be familiar with how they work and are able to very easily set up their local development environments to accommodate these tools. For the Python backend, we use the Black formatter.

The project is structured as a monorepo using Turborepo, which improves dependency management and allows us to enforce consistent workflows across the frontend and backend.

A notable feature of our setup is automatic type generation. We leverage the fact that FastAPI enables automatic generation of OpenAPI documentation (available on https://api.bug-track.projects.richarddominick.me/docs/) from Python type hints present in our backend code, to then feed this into openapi-typescript, an automatic code-generation (codegen) tool to convert OpenAPI documentation into TypeScript types.

This removes one of the biggest pain points, as well as source of bugs, when working with codebases consisting of more than one programming language, as there is no need to manually port over or write what is essentially duplicated code, just to enable type safety across the languages. The codegen tool will regularly poll the API to see if there have been any changes to any endpoints, and in addition to generating the types in TypeScript, will also generate a fully type-safe API client. Thus, developers do not need to implement the data fetching logic, as the relevant functions are automatically available for use once the backend endpoints have been set up.

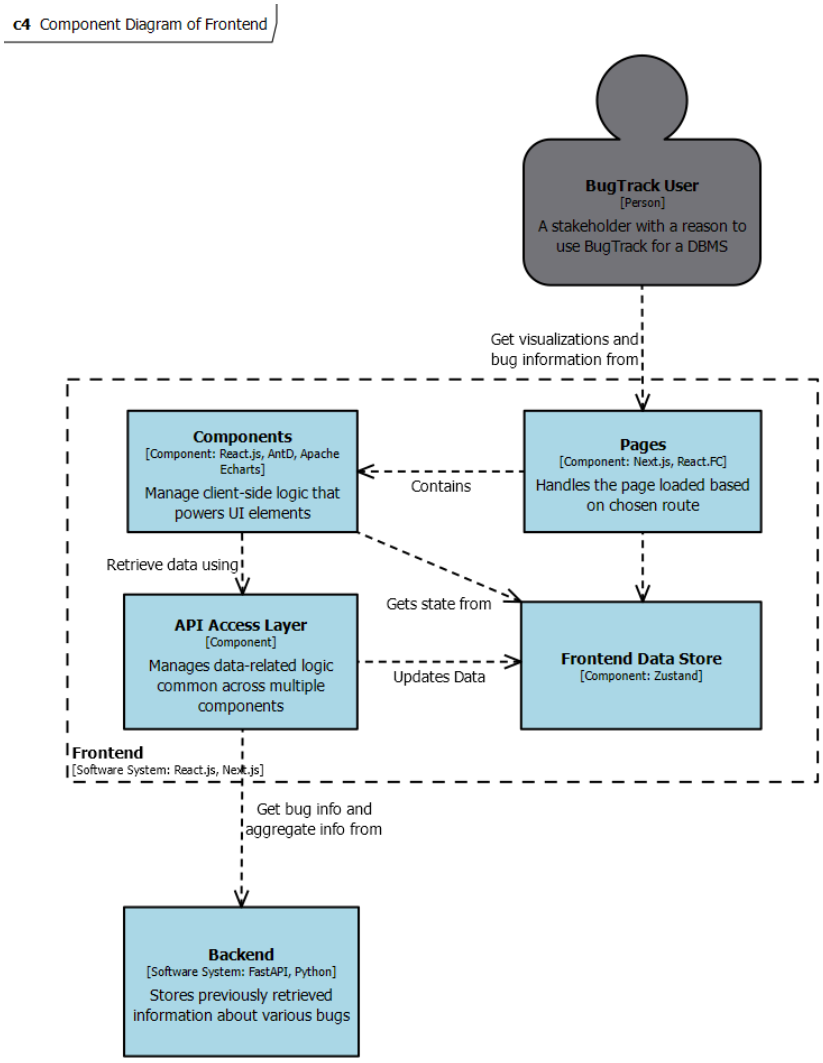
3. Frontend

3.1. Overview of Frontend

Our Frontend uses React with Next.js. With Next.js, the relative directories of files in the pages folder directly corresponds to the different relative urls that the user's browser is expected to call GET requests to, and for each, a React Component is expected to be rendered. No matter which page is rendered, however, the component is rendered as a child of the universal parent component in _app.tsx.

These pages can themselves contain UI definitions in the form of JSX, but they can also call reusable UI components called React Components.

We also have an API access layer, that abstracts functionality unrelated to the UI, such as storing and retrieval of information, and direct communication with the backend through HTTP requests.



3.2. Dashboard

3.2.1. Bug Explore and Bug Search

Progressive fetching: With the sheer amount of bug reports, we have decided to not load all the bug reports into the website on the first fetch. But rather, we would load a file bug report for each category, and allow users to trigger fetches by category. To track the number of bug reports currently loaded on the website, we maintained a **distribution.** When we want to fetch more bug reports from a certain category, say `Incorrect Query Result`, we will apply the category filter and thereafter offset the search by the number of `Incorrect Query Result` bug reports already loaded on the website.

Ant Design (Antd) UI framework: We have utilised the antd design framework across our application, but in particular, it helped bring our bug explore and bug search component to the

next level. With the ability to inject our own custom written components into parts of antd pre-built components, we were able to augment the components to our very own needs. For the bug explore and bug search module, we have adopted the Tabs, Card and List component, all working seamlessly in cohesion to form the robust component.

Filtering: To further enhance search capabilities, we have integrated a filtering feature, which allows users to filter bug reports by attributes such as category.

Let's take a look at a sample endpoint, `router.get("/{dbms_id}/bug_search")`, which takes in query parameters: `start`, `limit`, `search`, and `category_id`. This has allowed us to reuse this endpoint for searching by field and filtering, and also allows for potential extension should we allow multiple categories to be specified.

Autocomplete with debounce: A particularly noteworthy component is our autocomplete search bar, which performs matching across indexed bug titles. This is backed by a debounced input handler — ensuring that API requests are only fired when the user pauses typing, thus reducing backend load and avoiding unnecessary re-renders.

3.2.2. Chart components - Daily bug trend and Bug Distribution by Category

To provide an at-a-glance understanding of bug occurrences, we built two key visualizations: a Daily Bug Trend (line chart) and a Bug Distribution by Category (bar). We employed Apache ECharts due to its rich interactivity and extensibility. Tooltips, hover effects, and click-to-filter logic were all added with minimal overhead. Due to time limitations we were not able to implement server-side rendering, but this will greatly help the speed to our first browser paint.

3.2.3. AI DBMS Summary

To assist users in quickly grasping the state of the database systems, we integrated a natural language summary powered by OpenAI's GPT-4 API. The prompt is dynamically constructed using a significant sample of recent bug reports, ensuring decent coverage of the DBMS state.

3.2.4. Integration of periodic fetching details

To locate the categories of the bug reports that were fetched today, we utilised the speed and robustness of SQL itself to perform joins and aggregate querying, abstracted out into a modern Object-relational mapping (ORM). Querying with the SQLAlchemy ORM boasted great readability and robustness:

```
categories = tx.exec(
        select(
            BugReport.category_id.label("id"),
            BugCategory.name.label("name"),
```

```
            func.count(BugReport.id).label("count"),
        )
        .where(
            BugReport.dbms_id == dbms_id,
            func.date(BugReport.created_at) == today,
        )
        .join(BugCategory, BugReport.category_id == BugCategory.id)
        .group_by(BugReport.category_id, BugCategory.name)
    ).all()
```

3.2.5. Multitenancy (multi-DBMS) refresh logic

With support for multiple DBMS sources, we implemented multitenancy-aware refresh logic. Rather than freezing the entire dashboard during data refreshes, each component — whether it's a chart, list, or summary — has its own loading state and fetch cycle.

This granular control prevents blocking and provides a smoother user experience, especially during high-load operations like data aggregation. Components can independently retry failed fetches without affecting others, ensuring the dashboard remains partially usable even in the face of intermittent API failures.

3.3. Bug Report Page

3.3.1. AI bug report summary

To generate summaries for bug reports, we leveraged OpenAI's `gpt-4o-mini` model through the ChatCompletion API. Each bug report is processed by constructing a prompt that includes the name of the affected DBMS and the full bug description. The prompt is carefully designed to be concise and direct, ensuring consistent output quality and relevance across varying bug types.

To maintain factual and focused responses, we set the temperature to 0.2, reducing randomness and encouraging deterministic behavior. We also limit the max_tokens to 200 to enforce brevity and prevent overly verbose outputs.

Error handling is implemented to catch and flag failures in summary generation, ensuring that missing or malformed responses do not silently propagate through the system. In the scenario where the summary cannot be generated, an error message 'Error fetching AI summary, please try again' will be displayed to the user and a reload button allows users to retry generating the summary.

3.3.2. Similar bugs

To surface similar bug reports, we implemented a content-based similarity search using vector representations of bug reports. The details of generating vector representations can be found in section Bug Vectorizer below. When a user queries for similar bugs, the system fetches the vector for the target bug report and computes cosine similarity between it and all other available bug report vectors.

This similarity is calculated using the formula `1 - cosine(target, other)`, where a higher value indicates greater similarity. After computing pairwise similarities, we sort the reports in descending order of similarity and return the top N most relevant ones.

This approach ensures that suggestions are semantically relevant based on the language used in the title and description, rather than relying solely on metadata or manually defined rules. The vectors capture latent features such as bug behavior, affected components, and error context, which enhances the relevance of the retrieved results.

3.3.3. Comment section

The `CommentSection` feature is implemented as a modular, state-managed React component that allows users to view, add, and reply to bug report discussions. It integrates with an API layer using typed DTOs to fetch and post comments (`fetchDiscussions`, `addBugReportComment`, `addDiscussionReply`) and maintains authentication context to ensure only logged-in users can post. Comments and replies are rendered using reusable `Comment` and `Thread` components, with UI interactivity enhanced by modals for input,

loading skeletons, scroll-into-view effects, and animated highlights for new entries. This structure ensures separation of concerns, responsive feedback, and smooth user experience.

3.3.4. Bug report details

There are various details of the bug report that are reflected on this page. These include:

- Title
- Description
- Status
- GitHub issue url
- GitHub repository url
- DBMS
- Category
- Priority
- Versions affected
- Issue created at

These details can be fetched from the endpoint:

- GET /api/v1/bug_reports/{bug_id}

In addition, the category, priority and versions of the DBMS affected field can be edited. This is done by making a patch request to the respective endpoints:

- PATCH /api/v1/bug_reports/category
- PATCH /api/v1/bug_reports/priority
- PATCH /api/v1/bug_reports/versions_affected

More details regarding the API endpoints can be found in our API documentation here.

3.4. Authentication

This authentication system uses JWT (JSON Web Token) with a dual-token approach (access and refresh tokens) to secure API endpoints.

Token System

Token Types

1. Access Token: Short-lived (15 minutes) token used to access protected resources
2. Refresh Token: Longer-lived (1 day) token used to obtain new access tokens

Token Payload

- `sub`: User email

- `id`: User ID
- `name`: User name
- `token_type`: "access" or "refresh"
- `exp`: Expiration time

Authentication Flow

Registration/Signup

1. User submits email, password, and name
2. System checks if email is already in use
3. If email is available, password is hashed and user is created
4. JWT pair (access + refresh tokens) is generated and returned

Login

1. User submits email and password
2. System verifies credentials
3. If valid, JWT pair is generated and returned

Token Refresh

1. Client sends refresh token
2. System validates the refresh token
3. If valid, a new access token is generated (refresh token remains the same)
4. New access token and existing refresh token are returned

Accessing Protected Resources

1. Client includes access token in the Authorization header (Bearer {token})
2. Middleware validates the token
3. If valid, request proceeds to the controller
4. If invalid or expired, an error response is returned

Middleware (secured_endpoints.py)

Protects all /api/v1/* endpoints by validating the access token in each request:

1. Extracts token from Authorization header
2. Verifies token signature and expiration
3. Checks token type is "access"
4. Allows request if valid, returns error if not

Auth Endpoints

- POST /public/api/v1/auth/login: Authenticate and get token pair
- POST /public/api/v1/auth/signup: Create account and get token pair
- POST /api/v1/auth/refresh: Use refresh token to get new access token

Implementation Notes

- Passwords hashed with bcrypt
- JWT signed with HS256 algorithm

3.5. Testing

Frontend tests are managed with Vitest, and rendering, interacting and asserting properties of the DOM are done through React Testing Library.

The vast majority of unit tests on user-facing React elements follow a similar structure:

- Mocks are first created, to replace upstream and downstream components to test only incoming and outgoing interactions for the component being tested
- Tests are written at the bottom, and each test case starts with a description of a property that is expected out of the component
- Each test is the bare minimum assertions required to conclude that the property holds, assuming that all previously tested properties hold

Since frequent changes on the frontend is expected, it is important that tests are not dependent on any specific feature, value or even frequency. Instead, we favor assertions on mocks that assert the presence of certain key information anywhere in the calling arguments (see the assert helper functions in MockFnTestUtil.ts), and tests based on valid mock data that is shared across tests. If this is not possible, we make assertions on the DOM but attempt to avoid any direct assertion of html element type or css classes.

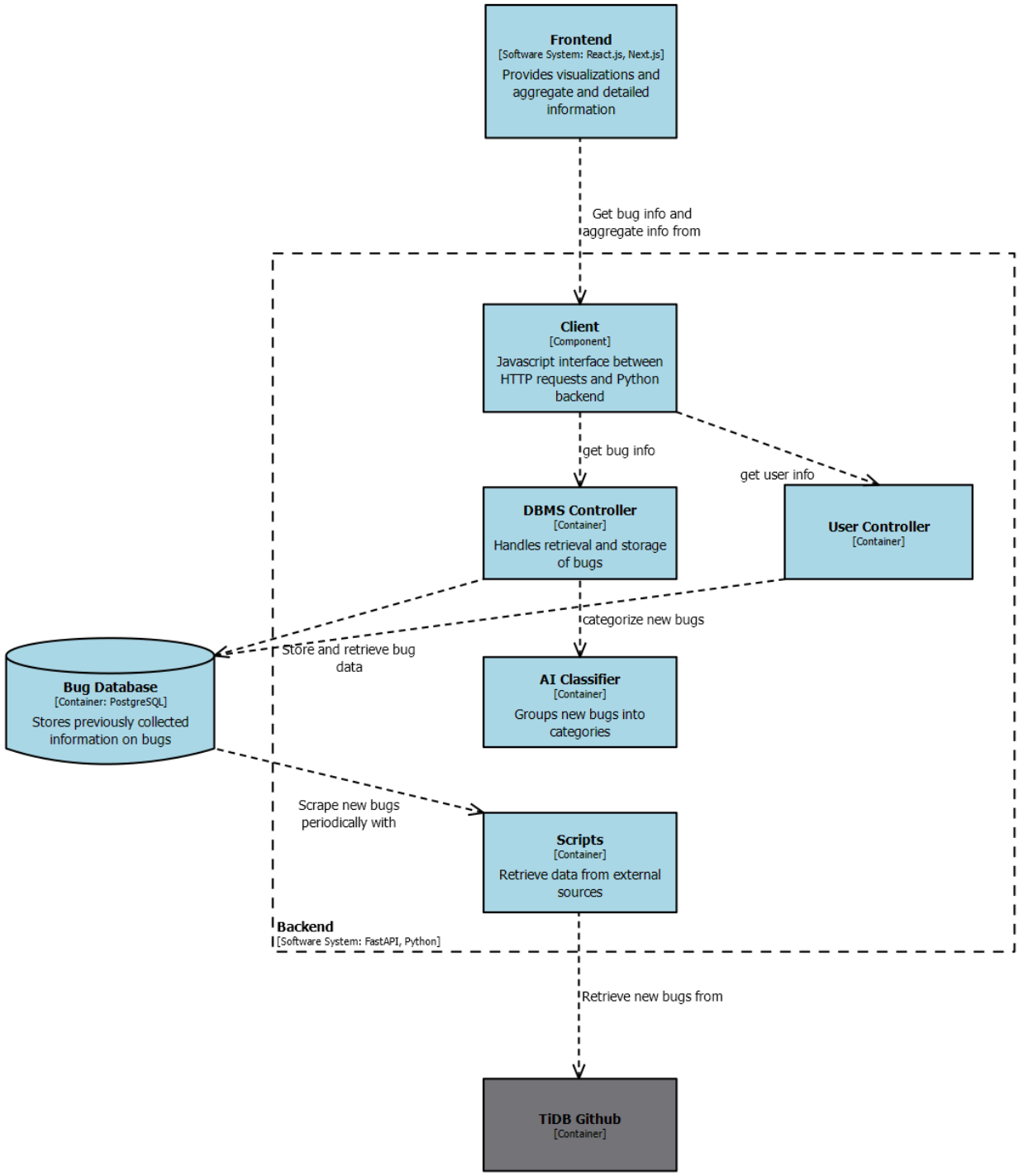
4. Backend

4.1. Overview of Backend

Our backend is built with FastAPI, using SQLAlchemy/SQLModel for ORM and database interactions. It features JWT-based authentication for secure access, and leverages Celery with Redis as the message broker for asynchronous task handling. We integrate OpenAI to generate AI-powered summaries of bug reports, and implement custom middleware for securing endpoints and managing error response.

The api/src folder contains the core backend logic for the Bug Track project, organized into key subfolders: **controllers** for defining FastAPI endpoints (e.g., bug reports, DBMS, discussions), **services** for implementing business logic and database interactions, **domain** for database models and schema definitions, **views** for data transfer objects (DTOs) used in API responses, and **internal** for shared utilities like error handling and middleware. This structure ensures a clean separation of concerns, making the backend modular, maintainable, and scalable.

c4 Component Diagram of Backend



4.2. Database

Our database schema is designed to support multi-tenancy by logically separating data across key entities such as bug_reports, dbms_systems, users, and comments, while maintaining normalized relationships through foreign keys. Each bug_report is linked to a dbms_system via a `dbmd_id`, allowing us to isolate and fetch data per DBMS instance

4.3. Periodic fetching

This document describes the system for automatically fetching bug reports from GitHub repositories and classifying them using machine learning techniques. The system runs as background tasks that periodically collect new issues and assign them to appropriate categories. At the same time, bug reports are vectorized and stored together in the database, and it is used to find similar reports.

4.3.1. System Architecture

The system is built on two primary components:

- GitHub Issue Fetcher: Retrieves bug reports from configured repositories
- Bug Classifier: Uses machine learning to categorize the collected bug reports

These components are implemented as Celery tasks that operate independently but coordinate with each other through a shared state manager.

4.3.2. Task Coordination

The TaskCoordinator class manages shared state between tasks, preventing them from overlapping and enabling them to communicate their status. It implements a singleton pattern to ensure consistent state across imports.

Key features:

- Tracks whether fetcher and classifier are currently running
- Provides methods to check and set running states
- Uses thread-safe locking mechanisms

4.3.3. Issue Fetcher

The GitHub Issue Fetcher runs on a scheduled basis to retrieve new issues from configured repositories.

Key capabilities:

- Fetches issues tagged with specific labels (e.g., 'fuzz/sqlancer')
- Tracks the latest issue timestamps to avoid duplicates

- Handles GitHub API rate limits with appropriate retry mechanisms
- Stores new issues in the database
- Triggers the classifier and vectorizer when new issues are found

4.3.4. Celery Configuration

The system uses Celery for task scheduling and execution:

- A Redis server acts as the message broker
- Periodic tasks are scheduled using crontab syntax
- Worker and beat processes are started in separate threads
- Task retries are configured with appropriate backoff strategies

4.3.5. Execution Flow

1. The Celery beat scheduler triggers the fetcher at configured intervals
2. The fetcher retrieves new issues from GitHub and stores them in the database
3. When new issues are found, the fetcher triggers the classifier and vectorizer
4. The classifier and vectorizer processes bug reports and updates the database
5. The tasks are coordinated to prevent overlap and ensure all issues are processed

4.3.6. Error Handling

The system implements comprehensive error handling:

- Both tasks use retry mechanisms with exponential backoff
- Special handling for GitHub API rate limits
- Database transaction management
- Detailed logging of progress and errors

4.3.7. Integration with FastAPI

The system integrates with a FastAPI application that provides REST endpoints for viewing and managing bug reports and categories. The Celery workers are started when the API application initializes.

4.3Bug Classifier

The Bug Classifier processes and categorizes bug reports that haven't been classified yet.

Key capabilities:

- Can be triggered by the fetcher or run independently
- Can classify specific bugs by ID or all unclassified bugs
- Waits for the fetcher if no bugs are available to process

Bug Vectorizer

The Bug Vectorizer vectorizes bug reports that do not have a vector representation. Each bug report is vectorized by:

1. Using an NLP model to process the bug's title and description
2. Extract word vectors from the processed text and compute their average
3. Serialize the vector and storing it into the database

4.4. Hybrid ML classification

The BugClassifierService implements a sophisticated classification algorithm that combines:

1. Text Preprocessing:

   Removes noise from bug descriptions

   Handles Markdown formatting

   Lemmatizes text and removes stopwords

2. Multi-strategy Classification:

   Keyword-based matching (highest priority)

   Fuzzy matching for handling typos and variations

   Semantic similarity using word vectors

   Machine learning prediction as a fallback

3. Confidence Handling:

   Uses confidence thresholds to handle uncertain cases

   Falls back to an "Others" category when confidence is low