DEVX360

Sipho Sehlapelo
Kesley Hamann
Sibusiso Mngomezulu
Lwando Msindo
David Musa-Aisien

DevX360

Introduction

This document outlines the purpose, scope, and technical requirements of DevX360. The SRS serves as a shared reference point for all project stakeholders — including developers, designers, testers, and clients — to ensure a consistent understanding of the system's objectives, features, and constraints throughout the development lifecycle.

Business Need

The South African software development industry, valued at USD 1.2 billion in 2023 and projected to reach USD 3.4 billion by 2030, is under increasing pressure to enhance team performance and deliver high-quality solutions efficiently. Engineering managers need data-driven insights to monitor and improve productivity, particularly through the adoption of industry-standard DORA metrics.

Project Scope

DevX360 is an AI-powered DevOps analytics platform designed to automatically track, analyze, and visualize key DORA metrics—**Deployment Frequency, Lead Time to Change, Mean Time to Recover**, and **Change Failure Rate**. By integrating with existing development tools, DevX360 provides actionable insights through real-time dashboards, automated reports, and AI-based recommendations to help engineering teams optimize workflows and reach elite performance levels.

User Stories

User Authentication

- **US1:** As a new user, I want to register via email/password or third-party providers

(GitHub, Google) so that I can access the system securely.

- **US2:** As a security-conscious user, I want to enable Multi-Factor Authentication (MFA) during login so that my account remains protected.
- **US2.1:** As a security-conscious user, I want to be automatically logged out after 30 minutes of inactivity so that my session isn't left vulnerable.
- **US3:** As a manager, I want to generate invite-only sign-up links for team members to ensure controlled access to the platform.

User Roles

- **US4:** As a manager, I want to view team performance metrics and detailed reports so that I can identify bottlenecks and optimize workflows.
- **US5:** As a team member, I want to view my personal metrics and receive AI-driven improvement suggestions so that I can enhance my contributions and commits.
- **US5.**1: As an admin/leader I want to be able to change roles of members to manage access as team structures evolve.

DORA Metrics Collection

- **US6:** As a DevOps engineer, I want the system to automatically collect deployment frequency and lead time from GitHub and CI/CD tools so that I can monitor delivery performance without manual effort.
- **US7:** As a team lead, I want real-time updates on Mean Time to Recover (MTTR) so that I can address incidents promptly.
- **US7.1**: As a developer, I want to see visual comparisons of current vs previous DORA metrics so I can track improvements.

Team Individual Metrics

- **US8:** As a manager, I want to view team-level metrics while respecting privacy so that I can foster collaboration without encouraging harmful competition.
- **US9:** As a developer, I want to see my individual contribution metrics privately so

that I can self-assess and improve.

AI Features

- **US10:** As a developer, I want AI to analyze my pull requests and suggest code quality improvements so that I can reduce technical debt.
- **US11:** As a team member, I want AI-generated commit messages to standardize my workflow so that commit logs are clear and consistent.
- **US11.1:** As a manager, I want AI to summarize team-wide performance trends from metrics so I can identify patterns without digging through raw data.

Compliance Tracking

- **US12:** As a manager, I want alerts when no commits are made on a workday so that I can ensure consistent progress and avoid code loss.
- **US13:** As a developer, I want reminders to commit code following a schedule so that I stay aligned with team expectations.

Dashboard Reporting

- **US14:** As a user, I want a role-based dashboard showing real-time DORA metrics so that I can track performance at a glance.
- **US15:** As a manager, I want automated weekly reports emailed to stakeholders so that I can share progress without manual effort.
- **US15.1**: As a team lead, I want to bookmark or pin specific metrics on my dashboard for easy access.

Integrations

- **US16:** As a developer, I want seamless integration with GitHub and Jira so that data flows automatically into DevX360 without duplication.
- **US17:** As an admin, I want API access to connect custom tools so that the system adapts to our existing workflow.

Security Compliance

- **US18:** As a user, I want RBAC to restrict access to sensitive data so that only authorized roles view specific metrics.
- **US19:** As a compliance officer, I want GDPR-compliant data handling so that user privacy is legally protected.

Non-Functional Needs

- **US20:** As a user, I want the dashboard to load within 2 seconds so that I can work efficiently without delays.
- **US21:** As a global team, I want the system to support 10,000 concurrent users so that scalability is ensured during peak usage.

User Characteristics

User Roles

- **Team Leader:** Oversees team performance, reviews DORA metrics, receives reports, and configures alerts. Responsible for inviting new Team Members via the invite feature.
- **Team Member**: Contributes code, receives feedback on pull requests and views personal metrics and AI suggestions. For privacy metrics are hidden from other users
- **Admin:** In charge of the initial setup and system configuration. Can select the initial Team Leaders.

Age Range

- Most users are between 21 and 45 years old, typically working as professional developers, team leads, or engineering managers.

User Skill Levels

- **Technical Understanding:** Most users are expected to have experience with GitHub, CI/CD pipelines and other Git-based workflows
- **AI Understanding:** Users are not expected to understand the internals of an LLM. The setup and internal functioning of the LLM will be abstracted from the user and will be handled by the developers
- **Web Interface Understanding:** Users are assumed to be comfortable with modern web interfaces and are expected to be able to interact with web pages (dashboard usage, report downloads).

Internal vs. External Users

- DevX360 is designed for internal use within software development teams.
- All users are authorized, authenticated members of an organization; there are no anonymous or public-facing access points. There is no ambiguity regarding users of the system.

Accessibility Needs

- Some users may need keyboard-only navigation or screen reader support.
- WCAG 2.1 AA compliance is a target to ensure accessibility for users with visual or motor impairments.

Availability & Usage Patterns

- Users access the platform during typical work hours, but some features like notifications or reports may be used asynchronously.
- Developers may log in multiple times a day to check metrics and PR feedback.
- Managers tend to review dashboards and reports on a weekly or sprint basis.

Security Awareness

- Users understand the need for strong authentication, access control, and privacy. Features like MFA, RBAC, and session timeout are expected.
- Compliance with GDPR/POPIA is necessary for both trust and legal alignment.

Use Case Diagrams

This section presents the use case diagrams for DevX360, illustrating the interactions between users and the system. These diagrams provide a visual overview of the main functionalities and workflows. The detailed diagrams can be found here.

Functional Requirements

FR1: User Authentication

- **FR1.1**: The system shall support user registration via email/password and third-party providers (e.g., GitHub, Google).
- **FR1.2**: The system shall implement Multi-Factor Authentication (MFA) for enhanced security.
- **FR1.3**: The system shall support invite-only registration to ensure controlled team access.

FR2: User Profile Management

- **FR2.1**: The system shall provide a user profile page displaying user details including name, email, role, and join date.
- **FR2.2**: The system shall allow users to upload and update profile pictures/avatars.
- **FR2.3**: The system shall support JPEG, PNG, and GIF formats for profile pictures.
- **FR2.4**: The system shall automatically resize and optimize uploaded profile images for performance.

FR3: User Role Management

- **FR3.1**: The system shall implement Role-Based Access Control (RBAC) for "Manager" and "Team Member" roles.
- **FR3.2**: The system shall restrict access to data based on user roles and privacy settings.
- **FR3.3**: The system shall allow managers to invite team members and assign

appropriate roles.

FR4: DORA Metrics Collection

- **FR4.1**: The system shall automatically collect **Deployment Frequency** metrics from CI/CD tools.
- **FR4.2**: The system shall calculate **Lead Time to Change** using data from version control systems.
- **FR4.3**: The system shall track **Mean Time to Recover (MTTR)** from incident and recovery data.
- **FR4.4**: The system shall compute **Change Failure Rate (CFR)** from deployment and rollback information.
- **FR4.5**: The system shall update DORA metrics after data ingestion.

FR5: Team and Individual Metrics

- **FR5.1: The system shall display team-level DORA metrics accessible to users.**
- **FR5.2**: The system shall provide individual performance metrics with configurable privacy controls.
- **FR5.3**: The system shall support filtering of metrics by time periods and individual team members.

FR6: AI Code Analysis

- **FR6.1**: The system shall analyze pull requests to detect code quality issues.
- **FR6.2**: The system shall provide actionable improvement suggestions based on code analysis.
- **FR6.3**: The system shall identify patterns that may negatively affect DORA metrics performance.

FR7: AI-Assisted Commit Messages

- **FR7.1**: The system shall suggest standardized commit messages for developers.
- **FR7.2**: The system shall improve the consistency and clarity of the commit log.

FR8: Commit Compliance Tracking

- **FR8.1**: The system shall track scheduled commit activity per developer.
- **FR8.2**: The system shall generate alerts when no commits are made on scheduled workdays.
- **FR8.3**: The system shall send scheduled commit reminders to team members.
- **FR8.4:** The system shall provide an activity history interface for users to view their

most recent action.

FRG: Performance Dashboard

- **FRG.1**: The system shall provide role-based dashboards showing real-time DORA metrics.
- **FRG.2**: The system shall allow users to customize their dashboard views.
- **FRG.3**: The system shall support drill-down features for in-depth performance analysis.

FR10: Automated Reporting

- **FR10.1**: The system shall generate automated performance reports for stakeholders.
- **FR10.3**: The system shall allow users to schedule and configure custom reports.

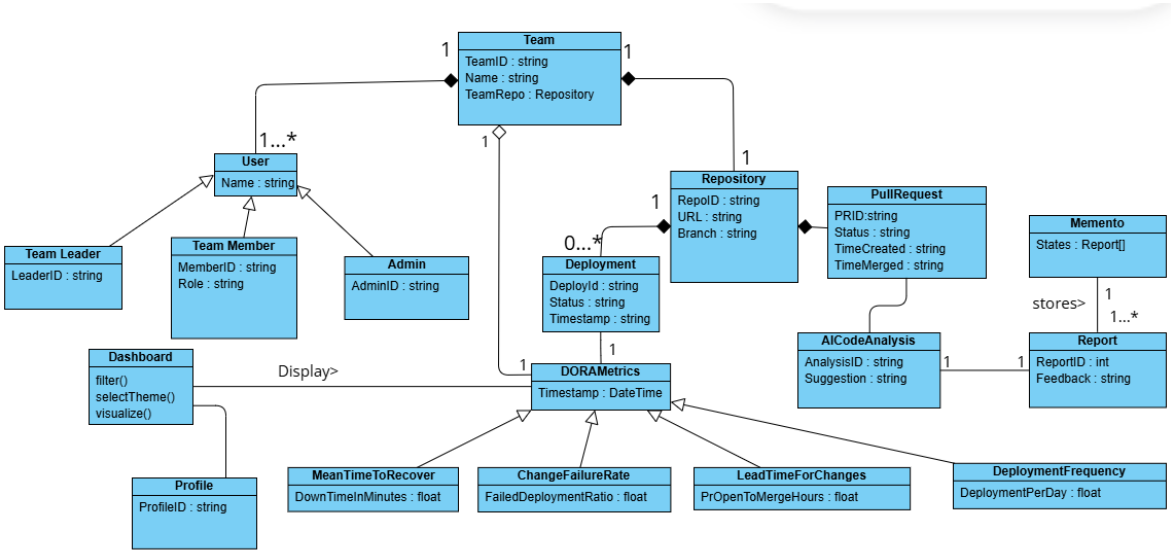
FR11: Tool Integrations

- **FR11.1**: The system shall integrate with GitHub to collect repository and commit data.

Service Contracts

This defines the service contracts for the DevX360 platform, outlining the expected behavior, inputs, outputs, and error handling for each API endpoint. The detailed document can be found here.

Domain Model



Architectural Requirements

Performance Requirements

1.1 **Dashboard Load Time**

Requirement: All dashboard pages must load within 2 seconds under normal network conditions.

Justification: Quick dashboard response times enhance user experience and usability. A load time of 2 seconds was determined to be sufficiently responsive as well as align with industry standards/usability benchmarks and reduce user frustration when using the system.

Implementation: CSS files were modularized to load only the styles required for each component. Unused styles and libraries were eliminated to minimize render-blocking resources.

1.2 **Data Processing**

Requirement: DORA metrics calculations must be completed within 5 seconds of data ingestion from integrated tools. This may be achieved using asynchronous event-driven components to process data in parallel.

Justification: Ensuring rapid processing of the repository information allows for real-time feedback loops for developers and project managers. Users can quickly view their metrics and adjust accordingly.

Implementation: Use of asynchronous processing so that events in the backend can occur at the same time. For example AI-analysis as well as API requests and DORA metrics calculations would all occur simultaneously to reduce overall response time and improve system efficiency.

1.3 **API Response**

Requirement: API endpoints must respond in the range 500ms to 2000ms for 95% of requests under typical load.

Justification: The API acts as a central daemon for our system. It is essential that if all communication passes through the API, it must forward requests and responses quickly to prevent any noticeable delay between interactions of our components. A 500 ms response time also reasonably aligns with industry standards. 2000ms represents the uppermost percentile for our API response time under the highest possible load.

Implementation: We made use of lightweight Express.js functionality and avoided unnecessary nesting or overuse of synchronous blocking operations. Requests from the Github API were also pre-processed in the form of filtering to reduce the time required to gather relevant data.

Scalability Requirements

2.1 **User Capacity**

Requirement: Support up to 50 concurrent users with no degradation in performance.

Justification: After discussion with the clients we were informed average team sizes would likely not grow larger than 10+ members. This user limit allows for the support of 4–5 teams with no drop in performance which aligns with client needs.

Implementation: Our API architecture is RESTful and stateless, this means each request is independent and does not require storing server-side session data. This allows multiple instances of the API to run in parallel across different servers enabling the system to support more concurrent users without performance loss. We made use of “express-rate-limit” to prevent monopolization by clients and enforce fair-use. This reduced overload during traffic spikes.

Reliability Requirements

3.1 **Uptime**

Requirement: Achieve 99.5% annual uptime, excluding scheduled maintenance.
Justification: This threshold ensures that the system is available throughout a general working-week with minimal downtime. This allows for developers and project managers to competitively view their metrics at any time.

Implementation: Implemented the system with graceful error handling on both frontend and backend to avoid complete system crashes during partial failures. Deployed using MongoDB Atlas as our database service which automatically backed up our data as well provided **replica sets.** This is a group of database instances that maintain the same data set on different servers. This ensured our database infrastructure maximized its uptime.

3.2 **Real-Time Updates**

Requirement: Ensure metric updates reflect in the dashboard with a maximum latency of 300 seconds after data ingestion.

Justification: A 300 second window balances the cost of data polling and system performance while still supporting near-real time visibility of project health.

Implementation: The system checks GitHub for new data every 5 minutes using the Octokit library. It looks at recent commits, releases, pull requests, and issues from the last 30 days. After getting this data, it recalculates the DORA metrics to keep them up to date. The updated metrics are then sent to the dashboard so users can see the latest information. This can happen either by the dashboard asking for updates regularly (polling) or by the server sending updates automatically

---

Security Requirements

4.1 **Access Control**

Requirement: Implement role-based access control (RBAC) for Managers and Team Members.
Justification: RBAC prevents unauthorized access to sensitive project data. It also prioritizes the principle of least privilege, promoting secure collaboration amongst developers and project managers.

Implementation: The system records the creator's ID when a team is created and logs the IDs of users as they join. When a dashboard view is requested, team creators are provided with full team details, while regular members only see limited information. An admin role is implemented to allow management of all users and teams, but safeguards are in place to prevent any admin from deleting another admin.

Usability & Accessibility

5.1 **Accessibility Compliance**

Requirement: Ensure the dashboard meets WCAG 2.1 AA accessibility standards.

Justification: Meeting this standard ensures inclusivity for users with disabilities and fulfills ethical and legal responsibilities for accessible design.

Implementation: We made the dashboard accessible by using screen reader-friendly HTML, adding text alternatives for images, ensuring keyboard navigation, and styling focus indicators. This ensures the app works well for users with visual or motor impairments. We also implemented a high contrast toggle to improve visual clarity for users with reduced vision/colour sensitivity.

Compatibility Integration

6.1 **Tool Integration**

Requirement: Support integration with GitHub.
Justification: GitHub and its API will be the core technology that will be interacted with in order to gather repository information. It is essential that that the system maintains seamless and reliable integration with GitHub to extract accurate data for DORA metrics calculation.

Implementation: Integration with GitHub was achieved using the official Octokit REST API client, which provided a reliable and well-documented interface to interact with GitHub repositories. Octokit was configured to authenticate using a personal access token stored securely in environment variables, ensuring secure access to private data.

Maintainability Support

7.1 **Documentation**

Requirement: Provide detailed API documentation and architectural guidelines.

Justification: Good documentation speeds up onboarding, simplifies integration, prevents ambiguity in the development stage, allows for a historical tracking of development, and ensures maintainability for continued development.

Implementation: This was achieved through a series of structured write-ups that detailed the planning, implementation, and progress of each system component. Additional guides and manuals were created to explain how these components are used. Reports were compiled on supporting aspects of the system, such as enforcing and adhering to coding standards. Version control was also prioritized to support accurate historical tracking via our dedicated team GitHub repository.

7.2 **Code Quality**

Requirement: Maintain a codebase with at least 80% unit test coverage and enforce consistent linting rules via CI/CD pipeline.
Justification: High test coverage and consistent linting practices help ensure a robust and maintainable codebase. This reduces the risk of introducing bugs during future changes, minimizes unexpected failures, and lowers long-term maintenance costs.

Implementation: Before committing code to the main or dev repository braches, developers ran tests and applied linting rules locally using tools like **Jest** for unit testing and **ESLint** and **Prettier** for code consistency. This approach ensured that only tested, clean code was pushed to the repository, even though formal CI/CD enforcement was not used.

Legal Compliance

8.1 **Data Privacy**

Requirement: Comply with GDPR and POPIA regulations for user data protection.
Justification: Legal compliance is mandatory for handling personal data. This not only reduces legal risks but builds user trust as well.

Implementation: User data such as names and email addresses are securely stored using MongoDB Atlas, with sensitive operations handled through authenticated API routes. No personal information is exposed in logs or frontend responses. Access to user data is restricted based on authentication and role-based permissions. We also

avoided collecting unnecessary data that falls outside the scope of the system's purpose.

8.2 **User Consent**

Requirement: Obtain explicit user consent for data collection and processing during sign-up.
Justification: Transparency in data collection respects user rights and aligns with best practices in ethical data management.

Implementation: During the sign-up process, users are required to agree to the platform's data handling policy via a clearly presented checkbox before their account is created. This consent is recorded and stored alongside their user profile. Users also have access to view or update their profile and revoke access if needed.

Architectural Patterns

N-layered Architecture

Description: Design the system using 4 different layers to handle different functionality required by the system as well as separate concerns.

Justification: This architecture was mainly used over others for its simplicity. The other architecture being considered (micro-services) would add unnecessary complexity for our current system's scope. After further research, we found that layered architectures also offered advantages such as simplified testing and debugging processes. The separation of concerns would allow for parallel development which would speed up the development process.

Architecture Implementation: The system features 4 layers, a frontend layer for displaying information to a user, an API containing multiple service routes, a backend layer that handles the immediate storage and calculation of DORA-metrics and other information and an AI analysis layer that will be used for further processing of fetched data**.** The implementation features a central API that serves as the sole

communication point for all system components. Each component interacts exclusively through this API.

Component-based Architecture

Description: The API follows a component-based architecture, with each component encapsulating its functionality and exposing only the required interfaces.

Justification: A component-based design allows the system to remain flexible, organized, and scalable. Components can be updated or extended without impacting other parts of the system. It also allowed for each component to be optimized individually.

Architecture Implementation: There are multiple components in the form of API service routes. The authentication component handles password hashing, token generation, and login/logout processes independently from the user management component, which manages user profiles, roles, and permissions. By isolating functionality into components, the API can safely expand with new endpoints or features without compromising existing functionality.

MVC Architecture

Description: The frontend is implemented using the Model-View-Controller to handle the storage of the UI state, the display of this state to a user and the handling of events and changes of these states.

Justification: MVC supports Maintainability (QR 7.2) by allowing independent updates to the UI, state, or control logic without affecting other parts. It improves Performance (QR 1.1–1.3) because UI changes can be handled efficiently through the View without unnecessary re-rendering. It enhances Reliability (QR 3.1–3.2) by isolating user input processing, and it contributes to Usability and Accessibility (QR 5.1) by enabling responsive, predictable, and accessible interface updates.

Architecture Implementation: The frontend's React components are organized according to MVC. The Model stores application state, the View renders components dynamically based on that state, and the Controller handles user interactions and updates the Model. This ensures modularity, maintainability, and predictable behavior, making it easier to implement new UI features and maintain a responsive, interactive interface.

Quality-Requirement-Driven Design

Description: Many of the design decisions early on were made to explicitly meet the Quality Requirements of the system.

Justification: This was done to prevent having to majorly redesign the system later on if we discovered upon further feasibility studies that a certain Quality Requirement couldn't be met.

Implementation: Detailed quality requirements were documented early on such that there was clear knowledge of the system expectations before development even began.

Decomposition

Description: The system was broken down into multiple components to allow for separation of concerns.

Justification: This made it much simpler to isolate parts of the system for review and allowed for parallel development across team members.

Event-Driven Processing

Description: Use asynchronous event handling for real-time metric updates and alerts.

Justification: This allows for backend processing (such as AI-analysis) to commence as soon as new data arrives instead of waiting for batch updates. It improves overall performance and responsiveness of the system, aligning well with our performance Quality Requirements.

Design Patterns

Pattern	Purpose

Singleton Pattern	**To ensure that the API makes one instance of the connection to the database**
Observer Pattern	**Enables real-time alerting and notification updates**
Strategy Pattern	**Allows multiple integration strategies for development tools//ASK**

Factory Pattern	**Facilitates creation of metric calculation services based on need**
Mediator Pattern	**API separates direct communication between any two high-level separate components**
Memento Pattern	**AI-analysis summaries may be saved and compared to view trends**

Constraints

- **GitHub Dependency:** The system must prioritize GitHub for sourcing metrics (e.g., commits, pull requests, CI/CD data).
- **Dashboard Functionality:** Dashboards must display at least 4 DORA metrics and support role-based privacy controls.
- **Hosting Limitations:** Deploy only on approved platforms.
- **Real-Time Data:** Metrics must reflect updates within 300 seconds of data ingestion.
- **Compliance Deadlines:** Adhere to GDPR and POPIA requirements from initial deployment.

Technology Requirements

Frontend Technologies

This refers to the technologies for the developing the parts of our system that are directly interacted with by a user

Framework:

-We made use of React (JavaScript). This is a component-based UI library that supports fast rendering. This already aligns with our desire for a responsive UI as documented earlier. It was also recommended to us by our client as they advised it would be easier to manage as our UI requirements grew in size and complexity. It also has widely available support making it more reliable and easier to learn.

Styling:

-We used CSS for the styling of our pages. Our team had good experience using this paradigm as was able to make stylish, appealing and easily modifiable designs in reasonable amounts of time

Visualization:

-We used JavaScript for the visualization of results. It would update the frontend based on receiving new data or based on scheduled time intervals. This would allow for real-time metric analysis and

clarity when interpreting DORA metrics which aligned with both our reliability and performance QRs.

Backend Technologies

This refers to the technologies used to make our system function. It deals with our system structure and processing behind the scenes.

Runtime/Framework:

-Node.js with Express. Node.js is lightweight and asynchronous, supporting scalable operations which aligns with our scalability requirements. Express provides a minimalist framework for building performant REST APIs with fast response times. This made it a suitable choice as stated earlier we hoped for an API response time faster than 500ms.

Database:

-The technology we used was MongoDB Atlas It worked well with our decision to use only open-source software. It provided managed cloud hosting which improved scalability as well as reliability. Reliability was further improved as cloud hosting provided inbuilt backup support.

AI-Analysis:

- The system uses **OpenAI** to perform AI-powered repository analysis. This enables fast DORA metrics calculations and insight generation, supporting near real-time updates to the dashboard (QR 1.2, 1.3, 3.2). OpenAI's secure infrastructure ensures compliance with access control and data privacy requirements (QR 4.1, 8.1) while maintaining system performance and uptime.

Integration Technologies

This refers to the tools that will be used to speed up the deployment and management of our system.

Version Control:

-The selected technology was GitHub API. This is the system's main data source. Integration with its API enables real-time updates on development metrics (QR 3.2) and aligns with compatibility requirements (QR 6).

CI/CD Pipeline:

-The obvious choice was GitHub Actions. It allowed automated testing and deployment pipelines. It was easy to setup as well as support for installing self-contained external testing libraries. This improved maintainability and provided a simple way for us to measure our test coverage as well as aligned with our compatibility requirements.

Hosting:

-We have currently decided on the use of Vercel. Vercel offers fast global content delivery,

serverless backend support, and CI/CD integration. It contributes to high uptime (QR 3.1), responsive UI performance (QR 1.1), and scalability (QR 2.1).

Development Tools

This refers to the tools we use to efficiently ensure that our system meets industry standards.

Code Quality:

-We will make use of ESLint and Prettier. Prettier simplifies code formatting with minimal configuration, while ESLint offers deep analysis and bug detection and can enforce consistent coding styles. Using both tools together ensures high-quality, consistent code, with Prettier handling formatting and ESLint enforcing best practices. The use of these tools would promote code maintainability and minimize unexpected failures during development. This would help us achieve QR 7.2 as well QR 3.1

Testing:

-We are using Jest for our testing. It supports both unit and integration testing and allows for the easy use of mock data as well as mock connections that are invaluable in the testing of a system that is to be hosted online. It ensures system correctness, reliability (QR 3), and meets the requirement for automated test coverage (QR 7.2).

Deployment Model

The deployment model ensures that the frontend, API, and backend services are reliably delivered to end-users, with clear communication channels and well-defined responsibilities across all layers.

DevX360 is delivered as a cloud-hosted web application. The user-facing frontend is a static single-page app, and the server-side components (API and AI integration) run as managed services. Persistent data is stored in MongoDB Atlas and generated files are stored in object storage. We are currently considering using AWS hosting services for majority of our needs for its proven reliability and global availability, a broad portfolio of managed services and strong built-in security and compliance controls.

Mapping Of Layers

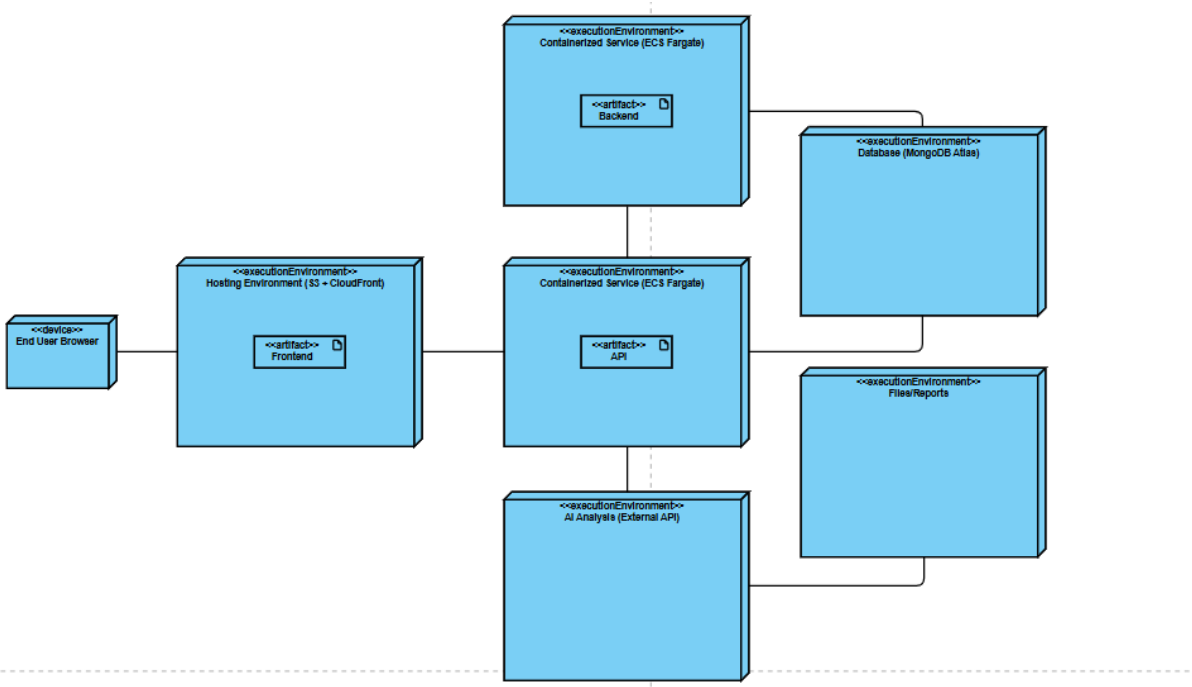
Frontend -- static site (S3 + CloudFront/CDN).
Backend / API -- containerized service (ECS Fargate or simple app service).
AI Analysis -- external API (OpenAI) called by the backend.
Database -- managed cloud DB (MongoDB Atlas).
Files/Reports -- object storage (S3).

Justification

- The frontend is a static React app, so hosting it as static files is cheap, fast and simple to operate.

- The API and AI logic are hosted as managed services to keep operational overhead low and allow straightforward scaling.
- Using a managed database (MongoDB Atlas) and a third-party AI service (OpenAI) removes the need to host and maintain DB or model infrastructure, simplifying security and maintenance.
- This approach keeps the architecture easy to explain, cost-effective for small deployments, and simple to scale if usage grows.

Diagram



[The detailed document can be found here.]

Live Deployed System