DEVX360

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

DevX360

Contents

Architectural Requirements

This section outlines the architectural design decisions and quality-driven rationale behind the development of DevX360, our AI-powered DevOps analytics platform. The architecture was carefully selected to balance performance, scalability, security, maintainability, and reliability, all while aligning with the project's functional goals, non-functional constraints and client recommendations.

Architectural Design Strategy

Architectural Design Strategy (N-Layer Client-Server)

- **Decomposition into distinct layers**

  Frontend (Presentation Layer): Displays information and collects input from users.

  API (Application Layer): Dispatches and processes requests from the JavaScript client, enforces security, and routes to backend services.

  Backend (Data Layer): Performs data retrieval, processing, and metric computation.

  (Note: API treated separately from backend because its responsibilities—request handling vs. data processing—differ significantly.)

- **Quality-driven approach**

Adopted QR-driven development — initial coding decisions were made to meet performance, reliability, and maintainability requirements from the start.

Chose mature open-source technologies to leverage existing optimization and reliability instead of building custom implementations.

- **Benefits of this strategy**

  Separation of concerns: Each component is modular, easier to isolate for testing and debugging.

  Parallel development: Different team members could work on layers independently without conflicts.

  Improved maintainability: Clear boundaries made it easier to identify and fix issues.

  Client transparency: Architectural decisions could be clearly explained in terms of feasibility and trade-offs.

Architectural Styles & Patterns

Frontend - MVC Pattern

We chose the Model-View-Controller (MVC) pattern for the frontend to organize the user interface and interactions in a modular and maintainable way. The Model manages application state and data fetched from the API, such as dashboard metrics, user profiles, and AI analysis results. The View handles rendering of UI components in React, dynamically updating displays as the state changes. The Controller processes user input, coordinates API requests, and updates the Model accordingly. This separation allows UI updates or new visual components to be implemented independently of state management or event handling, simplifying development and testing.

Quality requirements supported:

- Maintainability: Clear separation of concerns simplifies testing and ensures predictable interactions. Additionally, changes to UI or state logic can be made independently without affecting other layers.

Justification:
MVC enables a structured, organized frontend where each part can evolve independently, supporting maintainability, scalability of features, and easier debugging of user components. It would

N-Layered Pattern

The overall structure uses a 4-layered architecture, consisting of Frontend, API, Backend, and AI Analysis layers. Requests from the frontend, initiated by user interactions, are first sent to and handled by the API layer, which determines the appropriate service route. Depending on the request, the API either interacts with the backend layer — handling business logic, computations, and data access — or with the AI analysis layer for tasks such as processing

repository data. Importantly, no two layers communicate directly; all interactions pass through the API, ensuring a controlled and secure flow of data. This design isolates functionality, simplifies debugging, and enforces predictable and secure system behavior.

Quality requirements supported:

- Maintainability: Layers can be updated independently without affecting other parts of the system.
- Reliability & Security: Each layer can handle errors and enforce validation rules in isolation.
- Performance: Allows for individual optimization of each layer.

Justification:
A layered backend enforces separation of concerns, allowing modifications or extensions at any layer without breaking others, which improves maintainability, reliability, and ensures secure, predictable operations.

API – Component-Based Pattern

The API is designed following a Component-Based architecture, with modular components for authentication, user management, file uploads, and utility functions. Each component encapsulates its functionality and exposes only required interfaces. For instance, the authentication component handles password hashing, token generation, and login/logout independently of the user management component. This modularity allows components to evolve or scale independently while centralizing cross-cutting concerns like security and validation.

Quality goals supported:

- Maintainability**:** Components can be updated or extended independently.
- Security & Performance**:** Centralized control of authentication, rate limiting, and request validation ensures predictable and secure system behavior.

Justification:
A component-based API allows the system to remain flexible and organized while serving as the integration point between frontend and backend. Adding new API features or endpoints does not compromise existing functionality, maintaining system integrity and scalability.

Quality Requirements

There are numerous quality metrics that our system aims to satisfy, covering a range of attributes such as performance, scalability, maintainability, usability, and security. However, for each category, we identified and prioritized one core quality requirement that was most critical to the success of our project. These selected requirements guided our architectural decisions and implementation strategies:

Performance Requirements

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

Justification: This was heavily prioritized as we felt that it was of the main functionality of the system and thus should always respond quickly to the user to provide relevant information. A load time of 2 seconds was considered to be sufficiently responsive and align with common industry standards.

Implementation: Modular CSS files; eliminated unused styles/libraries;

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

Justification: The API was the central core of our system. All components attempting to communicate with one another required their requests/responses to be passed through the API, this made it a very large priority to make the API responsive to prevent any noticeable delay between interactions of components. 2000ms represents the uppermost percentile for our API response time under the highest possible load.

Implementation: Lightweight Express.js; avoided synchronous blocking; preprocessed GitHub API data to reduce response time.

Scalability Requirements

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

Justification: This was prioritized as the client stated the technology would often be employed for teams not exceeding sizes of 10+ people. This would allow for 4-5 teams to use the software simultaneously without a drop in performance allowing for parallel collaborations between teams.

Implementation: We used a RESTful, stateless API; parallelizable across servers; `express-rate-limit` to prevent overload.

Security Requirements

Requirement: Implement role-based access control (RBAC) for Managers and Team Members.

Justification: This was seen as a top priority as sensitive business information is being handled thus only authorized users should be able to access it. This as well promotes the least-privilege principle which only permits the minimum required access needed for each team member to do the assigned task.

Implementation:

Reliability Requirements

Requirement: Achieve 99.5% annual uptime, excluding scheduled maintenance.

Justification: This was important as it was emphasized a lot of work in the business is carried out through the use of Git-based workflows, it would thus be beneficial to have the system available to assist users and improve productivity be readily available at all times. This threshold ensures that the system is available throughout a general working-week with minimal downtime. This allows for developers and project managers to view and compare their metrics as frequently as needed.

Implementation: Graceful error handling; MongoDB Atlas with automatic backups and replica sets.

Maintainability Requirements:

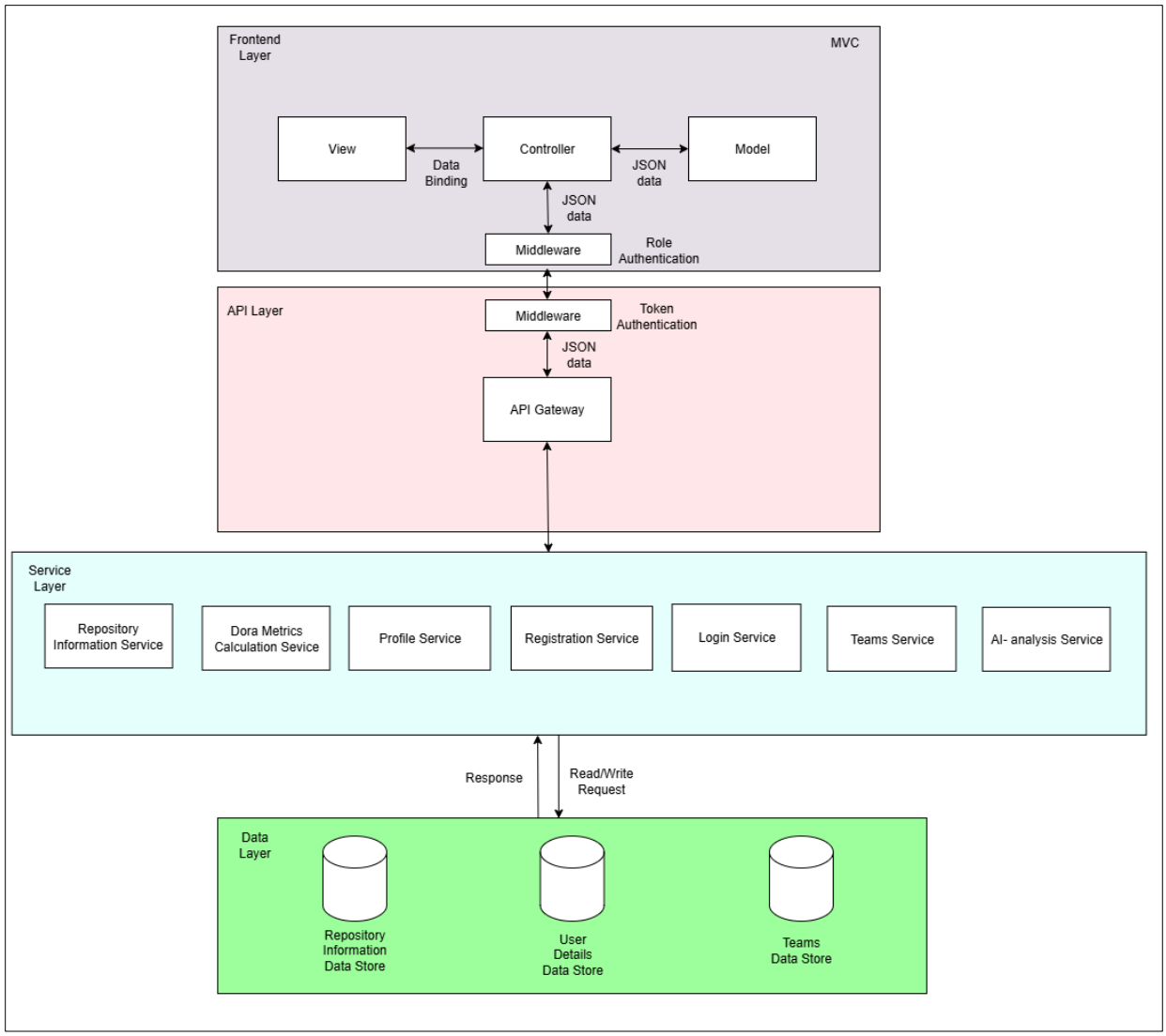
Requirement: Maintain a codebase with at least 80% unit test coverage and enforce consistent linting rules via CI/CD pipeline.

Justification: This was prioritized to ensure the code was well tested under all circumstances and to prevent any large scale system failures down the line. This reduced the risk of bugs during future changes as well as lowering long-term maintenance costs.

Implementation: Jest for testing; ESLint/Prettier for code consistency; tests and linting run locally before commits.

Architectural Design & Diagrams

A detailed high level diagram showing the interaction of our components:



Architectural Constraints

- To ensure accessibility for all team members, we made a deliberate decision to use open-source tools and frameworks exclusively. This ensured no software licenses etc. were required for development, deployment or testing. This helped us maintain a consistent development environment across all contributors.
- Our team was geographically dispersed, with members residing far away from one another, this meant physical meetings were often not an option. This necessitated on a strong emphasis on remote development and the use of remote collaboration tools. We used platforms such as Discord and WhatsApp to coordinate tasks and decisions.
- The client provided us with a large degree of design autonomy. The main concern was having a system that met their expectations and use cases. This however placed a

large responsibility on the team to set and meet appropriate and feasible architectural and technical standards.

Technology Choices

- ## Frontend – Login, View Dashboard, Profile Management

  We decided to use React for the design of our frontend as it was recommended by the client. Upon further research we discovered React offered a component-based architecture. It replaced the use of traditional DOM for handling state changes regarding our JavaScript.

  - This was advantageous as the declarative method of building UIs made development faster and easier to maintain. This aligned with our priority for maintainable code.
  - It was however disadvantageous due to have to learning a new technology under short notice.

  Another option considered was using traditional HTML and CSS for the frontend. It was the technology we had the most experience with as we had used it extensively for previous projects. Pages would be modelled using normal HTML and styled using CSS with the events/requests being listened to and handled by an accompanying JavaScript file for each webpage.

  - The main appeal of this approach was our familiarity with HTML and CSS. Each member of the team would be able to collaborate and contribute in the development of pages as well as consult in the event of errors.
  - A considerable drawback of this approach would be the manual DOM management as the pages grew increasingly complex. We planned for having pages that could dynamically change states upon receiving certain requests from a user and felt having to manually set up DOM and handling conflicts may prove challenging for a system of this scale.

  A third considered technology was angular. This option was considered due to our experience with it (more than React but less than traditional HTML/CSS) and because its framework supported typescript which was a technology that was considered for the development of our API. We considered using this technology as the entire framework for our project building it from the ground up.

  - This appeal of this specific approach was its All-in-one framework – we would completely integrated tools for testing, development, HTTP services, routing etc. all provided by one technology stack. This would improve reliability and align with our desire for maintainable code as we could use the in-house tools for all of our testing.
  - We were however swayed away by the steep learning curve of this technology. With limited experience using Angular we felt that the time investment required to fully understand and implement the framework would reduce our

development velocity and introduce unnecessary complexity during the early stages of the project.

- ## API

  We decided to develop the API using JavaScript. We made use of many of the packages and libraries made available in the JavaScript framework. Requests from the frontend received would be handled using the appropriate service route. We used modules from the npm package such as jsonwebtoken and bycrpt.js to create JWTs and to hash passwords respectively to improve security.

  - JavaScript was advantage our as it's a very widely used language. It has good performance and is lightweight. It was also appealing because it is generally an easy language to pick up and further learn considering the team was already familiar with it.
  - A drawback of this approach that we faced was handling of conflicts between npm modules. It was important that we only used the necessary modules and that including any didn't cause clashes

  A technology we briefly considered was Typescript. It was a superset of JavaScript with the addition of strong static typing for variables.

  - This addition would allow for better code quality and improved maintainability of code.
  - We were however swayed away from type script as we felt the added tooling overhead was unnecessary for the current scope and size of our project.

  The third considered technology was Python. It would've mainly used packages like Flask or FastAPI for development. From that point routes/services would be handled similarly as it would be in related JavaScript and Typescript programs.

  - An API designed in Python would boast great simplicity and readability. The use of well-established Flask and FastAPI packages would also ensure reliability in the development of our API
  - The main con of this was managing cross-language communication. It seemed unnecessary to make an API in Python and attempt to communicate with the underlying JavaScript from the frontend component.

- ## Backend – Fetch Repository Information, Calculate DORA metrics

  The function of our backend component was to scrape data (that would be used in the calculation of DORA metrics) from GitHub repositories.

  The technology we decided on was Octokit. This was the official GitHub SDK for JavaScript. Octokit provides a well-documented, actively maintained, and strongly-typed interface for accessing repository data.

- This was beneficial as it simplified making API requests, handling response and managing authentication. It eliminated the need for generating JSON web tokens manually. There is also always the added benefit of improved reliability when using official software
- There wasn't any major drawback to using Octokit over other technologies except for minor compatibility issues involving maintaining the system if there ever is an update to the GitHub API.

We also briefly considered the use of Python to interact with the GitHub API. It would make use of the requests package in the Python framework.

- As previously mentioned working with Python would provide improved simplicity and readability of code.
- The drawback would be similar to that previously mentioned. It was deemed unnecessary to try and manage the cross-language communication when there are alternatives that eliminate this issue.

The last technology we looked into would be using manual curl requests. We would need to generate an authenticated GitHub token before interaction can commence however.

- The advantage of curl would be its ability to function with any language. This would make it suitable for any environment.
- A large drawback of this approach would be the security risks. It was considered unsafe to directly pass tokens in command strings, shell scripts and other curl requests to the GitHub API.

AI Analysis

For our AI component we decided to use the OpenAI API. This cloud-based service allowed us to integrate powerful language models such as GPT-4 directly into our system and access them through our API without requiring any local setup.

- The API also provided faster and more in-depth analysis than what we achieved using locally run open-source models, making it easier to meet our performance and accuracy quality requirements. Its cloud-hosted nature ensured high availability and consistent performance regardless of user load.

- A drawback of this technology was concerns we had over the distribution of possibly limited API tokens to allow for sufficient use of prompts to the language model.

An alternative considered was LM studio. LM Studio is a desktop application that allows users to run and interact with open-source language models locally. It is built on the same underlying technology as Ollama—llama.cpp

- One of its key advantages is its ease of setup. It provides a user-friendly graphical interface that allows models to be downloaded and tested with minimal configuration

- It is however less suited for backend integration. For our system where AI functionality needed to be imbedded into the full-stack application and accessed via API calls, LM Studio's GUI based nature made it an impractical choice.

Ollama was considered as a second alternative. This tool allows running large language models locally on developer machines without requiring any cloud infrastructure

- One of its main advantages is privacy: since models are executed locally, no sensitive user data needs to leave the system, aligning with our security and confidentiality requirements. It also supports fully open-source models, which aligns with our decision to avoid proprietary software dependencies.
- However, running models locally can introduce significant hardware requirements, which may impact performance on less powerful machines. For production environments, scaling this approach would be more complex compared to cloud-based solutions.

• Deployment

We decided to host the DevX360 API as serverless functions on **AWS Lambda** behind **API Gateway (HTTP API)**, manage secrets with **AWS SSM Parameter Store**, use **DynamoDB** as our primary cloud datastore, and use **CloudWatch** for logging and basic monitoring. CI/CD is handled with **GitHub Actions** using OIDC and **AWS SAM** templates for repeatable infrastructure deployments. This stack was chosen to minimise operational overhead, remain cost-conscious (fit the AWS Free Tier where possible), and provide an easily reproducible deployment workflow that integrates with our GitHub-based development process.

- Our chosen serverless deployment stack (AWS Lambda, API Gateway, DynamoDB, SSM, CloudWatch, and GitHub Actions) provided several advantages: low operational overhead, automatic scaling, secure secret management, and cost-efficiency within the free tier, all while integrating seamlessly with our GitHub-based CI/CD.
- Our deployment stack however introduced challenges such as potential Lambda cold start delays, the complexity of NoSQL data modelling in DynamoDB, and more difficult debugging compared to traditional servers. Overall, the trade-off favored simplicity, scalability, and affordability for our project scope.

An alternative considered was using traditional compute or container hosting (EC2/ECS/Fargate) and RDS for relational storage.

- Favourably considered as I provided full control over runtime environment, easier local-like debugging, and natural support for relational schemas and complex queries.
- It was disadvantageous because it introduced Higher operational overhead (servers/containers to manage), likely higher cost (less free-tier friendly), and added deployment complexity.