Architectural Requirements

1.1 Architectural Design Strategy

Strategy: Design based on quality requirements

Our strategy emphasises the importance of the system's quality requirement. This strategy ensures that our design decisions will lead us to deliver a system that upholds our quality requirements and guarantees that the system operates as expected.

1.2 Architectural Styles

1. Three Tier Architecture:

- Security
- Simplifies design
- Maintainability and portability

This architecture partitions our system into three different layers, namely the presentation layer, service layer, and database layer. By separating concerns of the user interface from our backend services and database, we can provide the users with a nice, easy-to-use interface that will also provide added security since the users cannot access the database directly. This separation of layers also makes the application more maintainable and scalable since changes to one layer should not affect another.

2. Component-Based Architecture:

- Robustness
- Reliability (via thorough testing)
- Reusability

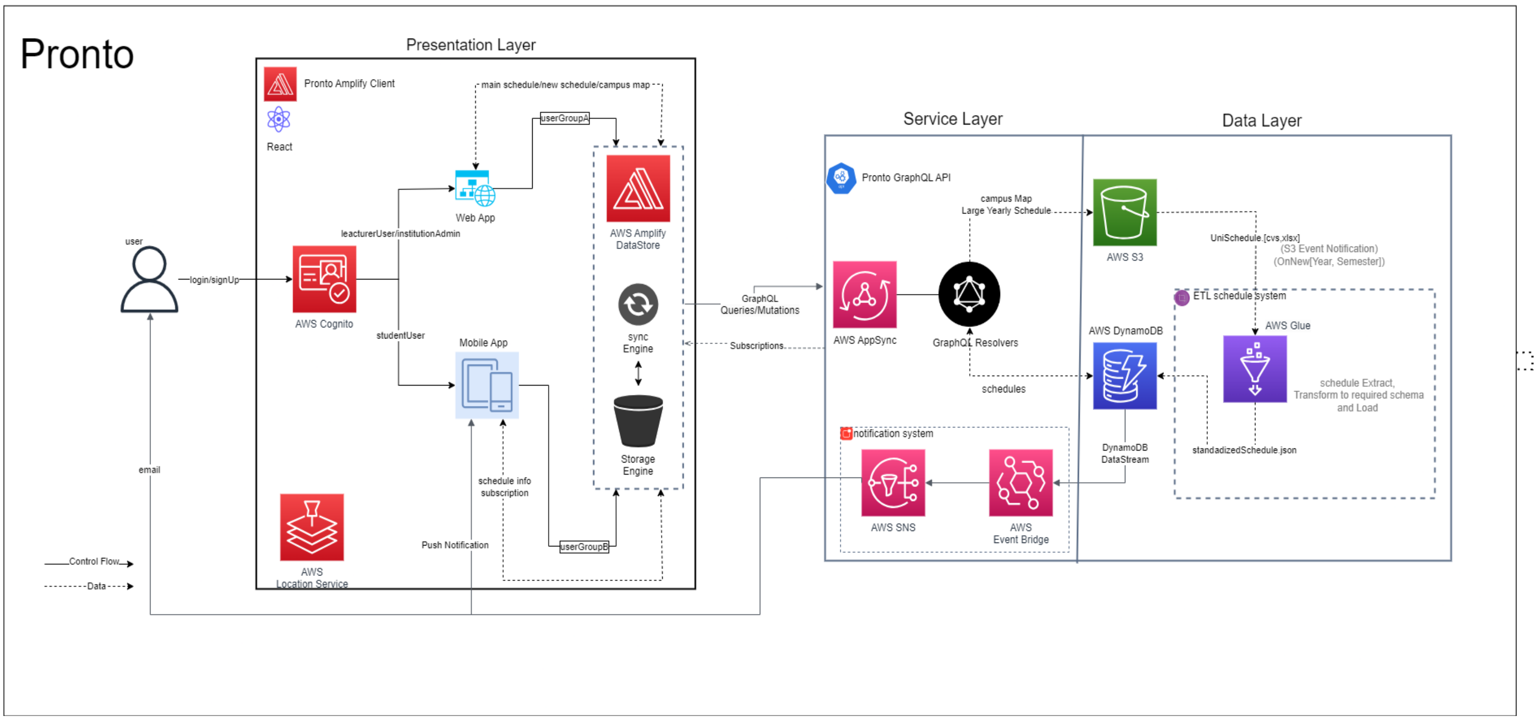
By composing our UI of several components, we can encapsulate related data and functions in each component, which will aid in testing as well as organise backend functionality to improve the overall quality of the app. This architecture also allows us to reuse certain useful components in multiple instances, reducing our production time and increasing overall cost/time efficiency.

3. Serverless Computing Architecture:

- Reliability
- Scalability
- No server management

This architecture will allow us to achieve our required uptime/reliability guarantee without having to worry about infrastructure maintenance or scaling. The AWS platforms we are using will allow us to automatically scale our app depending on the incoming workload, and make integration between several components and services simpler, improving overall app quality, maintainability and production time.

1.3 Architectural Design: Diagram



1.4 Architectural constraints

- **Scalability:** The system needs to be capable of seamlessly scaling to accommodate a large number of students and lecturers accessing our resources automatically.
- **Security:** The system must ensure the security of user credentials, such as passwords used for creating and signing into accounts, by implementing robust protective measures.
- **Maintainability:** The system should be well-documented to facilitate easy maintenance, and its design should follow modular principles, allowing for efficient updates and modifications.
- **Portability/Tech stack:** Since the project requires both web and mobile applications, it is important to select the appropriate frameworks that meet the project requirements. Additionally, the system should have limited reliance on libraries and AWS services to ensure flexibility and reduce dependencies.
- **Testability:** The architecture should support automated unit testing and integration testing, enabling comprehensive and efficient testing of system components.

2. Quality Requirements

2.1. Performance Requirements

A. Scalability

Definition

The product should be at first able to scale horizontally to accommodate for typical traffic that would be expected by an institution during peak times, then this should extend out to multiple institutions per region.

Acceptance Criteria

- Autoscaling of the services used will be able to accommodate changing traffic on a regular basis.

B. Performance

Definition

The different clients of the product should be fast and highly responsive.

Acceptance Criteria

- End-to-End testing will be conducted regularly to test server response times and ensure that a user can quickly perform each use case or multiple use cases in an acceptable time frame.
- Different instances of the back-end will be deployed per major region to decouple scaling, allowing for different regions and resources to scale up accordingly.

2.2. Availability

Definition

The tool should always be available for all users and any unavailability communicated.

Acceptance Criteria

- The website and application should have an uptime of at least 99% as it is critical for a user to be able to access and use their timetables at any given time.
- Any planned maintenance will occur outside of normal university hours or schedules
- Planned downtime will be indicated or communicated with clear UI changes.

2.3. Security

Definition

User data should be secured at rest and in transition in order to prevent unauthorised access and comply with local laws and regulations (POPI).

Acceptance Criteria

- User data will be Encrypted in transition and Encrypted at rest. (AES 256 is used by Cognito, with salt added)
- Users will be grouped into user pools during sign up
- This will provide limited access to data modification and access as per use case.

2.4. Maintainability

Definition

The software should be easily maintainable by both internal, or new external party(s), with clear concise documentation and code.

Acceptance Criteria

- Code linters and standard linting rules will be applied to ensure code uniformity.
- A 3-person review procedure will be used to ensure that all solutions provided are efficient and employ the best procedures.
- New features implemented will be explained thoroughly with each Pull Request, which will also be tested and evaluated by at least 3 parties prior to deployment.
- The application will be serverless in it's core, which means that other than optimising implementation, there won't be a server to maintain.

2.5. Reliability

Definition

The software should be stable and present reliable and verified data, with low downtime and errors.

Acceptance Criteria

- A lecturers account will be confirmed based on what the university has provided
- Students will be asked to confirm their courses after the selection

Error logging will be performed so that its nature can be assessed and prevented in the future.

3. Technology Choices

Front end - **Mobile Application**:

We chose React Native as our mobile front-end technology. It's an open-source framework that allows JavaScript code to create native-like applications for iOS and Android platforms. React Native's use of native components and APIs ensures a rich and performant user experience.

Pros and cons of React Native:

- Code Reusability: React Native enables code reuse, reducing development time and cost by sharing code between web and mobile apps.
- Time and Cost Efficiency: With React Native, one codebase serves both iOS and Android platforms, saving development time and cost.
- Native-like Performance: React Native achieves smooth and responsive user interfaces, although it may have performance limitations in complex animations or graphics-intensive tasks. However, these limitations don't affect Pronto as it doesn't rely on heavy animations or graphics.

Alignment with the Selected Architecture:

- 3-Tier Architecture: React Native fits the presentation layer, separating the user interface and backend services with clear boundaries.
- Component-Based Architecture: React Native's component-based approach aligns with the chosen architecture, promoting code organisation, modularity, and reusability.
- Serverless Computing Architecture: React Native integrates well with serverless computing, enabling secure and scalable communication with selected AWS backend services like Lambda, DynamoDB, and GraphQL.

Other similar technologies:

While Flutter and Xamarin can achieve cross-platform, native applications, React Native has advantages:

- Larger Ecosystem: React Native has a broader ecosystem and community support, providing more resources for development.
- JavaScript: React Native's use of JavaScript aligns with our team's existing expertise, making it more accessible compared to Flutter, which requires learning Dart.
- Platform Maturity: React Native's longer existence has resulted in a more mature and stable platform compared to Flutter.

Front-end - **Web Application**:

ReactJS is the choice we made for the web application of the Pronto project. This is due to its component-based architecture, virtual DOM, rich ecosystem, reusable components, and JSX syntax.

Pros, cons, and alignment with architecture:

With ReactJS, UIs can be broken down into reusable components that efficiently update and render when data changes.

The creation of reusable components in ReactJS improves productivity and code quality, reducing duplication and making maintenance and extension easier. This aligns with our chosen component-based architecture.

ReactJS aligns well with the selected 3-tier architecture by separating the user interface from backend services, maintaining a clear separation of concerns. It is also compatible with serverless computing architecture, consuming APIs efficiently.

Other similar technologies:

Compared to similar technologies, ReactJS is considered lightweight with a smaller learning curve than Angular, and it has a larger community and ecosystem. When compared to Vue.js, ReactJS shares similar architectural principles and performance but benefits from a larger community and more available resources.

For styling and components, we opted to use material UI in combination with Bootstrap. This is because these technologies are well integrated with ReactJS and have a visually appealing and consistent UI. Bootstrap offers a comprehensive design system with a wide range of customizable components and a responsive grid system. Both frameworks facilitate rapid development with ready-to-use components.

Back-end - **(Database)**:

Pronto has chosen Amazon DynamoDB as the backend database for scheduling data storage and retrieval. DynamoDB's scalability, performance, alignment with the chosen architecture, and comparisons to similar technologies are the main reasons for this decision.

Pros, cons, and alignment with architecture:

DynamoDB's scalability enables handling large volumes of scheduling data and adapting to workload demands. It provides seamless scaling in terms of storage and throughput, ensuring performance and avoiding downtime. DynamoDB aligns well with Pronto's chosen architecture, especially in a serverless computing environment, integrating smoothly with AWS Lambda and GraphQL.

DynamoDB's performance is excellent, delivering low-latency, high-throughput access for real-time scheduling applications. Its distributed nature and data replication enhance reliability and availability.

However, DynamoDB has limitations, such as the lack of support for complex relational queries. Data modelling and query design require careful consideration.

Other similar technologies:

Compared to Apache Cassandra, a competitor, DynamoDB stands out due to its managed service approach, eliminating database administration tasks. DynamoDB's integration with AWS services like CloudFormation and IAM enhances ease of use and manageability.

Back-end - **(API)**:

Pronto has decided to implement a GraphQL API using AWS AppSync or AWS API Gateway, as it believes that GraphQL is the ideal technology to enable efficient data retrieval and manipulation. GraphQL's flexible and precise query language provides several advantages that align with Pronto's chosen architecture and communication requirements between the frontend and backend components of the application.

Pros, cons, and alignment with architecture:

One of the key benefits of GraphQL is its ability to allow clients to request only the specific data they need. With GraphQL, clients can send queries that define the exact data requirements, avoiding over-fetching or under-fetching of data. This minimises network payload and reduces the number of round trips between the client and server, resulting in improved performance and reduced bandwidth consumption.

Additionally, GraphQL enables clients to retrieve multiple resources and related data in a single request. Through its concept of "resolvers," GraphQL consolidates data from various sources into a single response, enhancing efficiency and reducing the number of API calls. This capability is particularly valuable for Pronto, as it aims to retrieve and manipulate scheduling data efficiently.
One potential downside is the increased complexity of setting up and managing the GraphQL server and schema. It requires defining and maintaining the schema, resolvers, and relationships between data entities.

Other similar technologies:

When comparing GraphQL to similar technologies, such as RESTful APIs, GraphQL stands out in several aspects. RESTful APIs often require multiple endpoints to retrieve specific data, leading to over-fetching or chaining of requests. In contrast, GraphQL allows for more precise querying, eliminating the need for multiple round trips and reducing unnecessary data transfer. Additionally, with RESTful APIs, versioning and managing different versions of the API can become complex, whereas GraphQL's schema evolution capabilities make it easier to introduce changes without breaking existing clients.

Other:

Blob Storage:

The scheduling application shall utilise Amazon S3 for storing and serving files such as campus maps, logos, or any other blob-like data associated with the scheduling system. S3's durability and scalability will ensure reliable storage and retrieval of files while providing a secure access mechanism.

Authentication:

Pronto shall utilise Amazon Cognito for user authentication and authorization. This service will handle user sign-up, sign-in, and management of user credentials, ensuring secure access to the application's functionalities.

AWS Cognito