**Decision: Domain Modelling and Classes**

I chose to create a generic set of classes where all inspections sit under one main Inspection class, whether it’s a facility check or a machine safety check. I also added classes like User, Site, Zone, SubCheck, InspectionResult, and Attachment.  
I did this because I wanted the system to be flexible and avoid duplicating work for different inspection types. It also means that if new inspections are introduced later, they can easily fit into the same structure.

**Report version:**

* **What:** Designed a generic class model with an overarching Inspection superclass, supporting both facility inspections and machine safety inspections. Key entities included User, Site, Zone, SubCheck, InspectionResult, Reading, and Attachment.
* **Why:** This unified structure avoids duplication, increases reusability across inspection types, and ensures auditability. The separation of domain logic from implementation logic followed TM354 guidance, ensuring adaptability before committing to technologies

**Decision: Requirements Engineering**

I wrote down clear functional requirements such as login and role-based access, the ability to pick sites and zones, load inspections dynamically, record pass/fail checks with photos and comments, and let managers review results. I also set non-functional requirements around usability, security, and performance.  
The reason I did this was to make sure I captured exactly what engineers and managers need from the system. Having requirements defined up front also meant I could design with flexibility in mind, like loading new subchecks from the database without having to touch the code.

**Report version:**

* **What:** Defined functional requirements (FR1–FR6) covering user management, role-based access, inspection workflows, dynamic sub checks, photo attachments, and manager reporting. Also established non-functional requirements covering usability, performance, security, reliability, and maintainability.
* **Why:** These requirements ensure the system directly reflects operational needs. For instance, dynamic loading of inspections from the database reduces future maintenance effort, while role-based access enforces security and compliance.

**Decision: Database Design**

I decided to use a relational database design, with tables based on my classes (Users, Sites, Zones, Inspections, SubChecks, Results, Attachments, etc.). Although SQL Server will be implemented for the final system the prototype I am building as part of the TM470 will include SQLite database for simplicity and ability to embed the database with the application and make it operational in offline mode.   
The reason for this is that relational databases give me strong data integrity, which is really important for inspections and audits. SQL Server also matches what the company already uses, so it will be easier to maintain and support in the long run.

**Report version:**

* **What:** Adopted a relational model implemented through SQL Server tables reflecting conceptual classes (Users, Sites, Zones, Inspections, SubChecks, Results, Attachments, Parameters, Readings).
* **Why:** Relational models enforce referential integrity and data consistency, which are critical for compliance-heavy environments. A bottom-up approach starting from the database was advised by an experienced engineer to ensure long-term stability and alignment with enterprise IT policies

**Decision: Technology Selection**

For the prototype, I used React Native with Expo. It was agreed that the final system will be created as a desktop application in C# with SQL Server database, but this is out of the scope for TM470 project.   
The reason I picked React Native at the start was because it’s fast to build prototypes and let me test workflows with stakeholders without spending too much time or money. For the final version, C# and SQL Server made more sense because they align with what the IT department already supports, and it’s easier for them to look after.

* **What:** Used React Native with Expo for prototyping, then selected C# with SQL Server for final implementation.
* **Why:** React Native allowed rapid prototyping and validation of workflows at low cost. However, for the final system, C# and SQL Server aligned with the company’s IT infrastructure and support model. Alternatives (.NET MAUI, Xamarin, MySQL, NoSQL) were evaluated but rejected due to limited support, weaker compliance guarantees, or lack of organisational fit.

**Decision: Prototyping and User Interface**

I made wireframes and then built an interactive prototype in React Native.  
The reason I did this was to show engineers and managers how the system would look and work. This helped me collect feedback early and confirm I was on the right track before building the actual system.

**Report version:**

* **What:** Created wireframes and an interactive prototype in React Native to validate workflows and collect stakeholder feedback.
* **Why:** Prototyping ensured usability and requirement validation before significant resources were committed. React Native was chosen because it allowed quick, cost-effective iterations, even though it was not intended for final deployment.

**Decision: Scope Management**

At first I thought about adding more advanced features like predictive maintenance and cloud access. In the end, I narrowed the scope to focus only on the core functions: digital inspections, role-based login, and storing/retrieving data.  
The reason I scaled back was to keep the project realistic for the timeframe, while still solving the most important problems.

**Report version:**

* **What:** Reduced project scope to focus on core features (digital inspections, role-based access, data storage/retrieval), while deferring advanced features such as predictive maintenance and cloud-based access.
* **Why:** This decision kept the project realistic within the academic timeframe while still meeting the critical operational needs of engineers and managers.

**Decision: Application Architecture**

I decided to go with a three-tier architecture. This way I will be able to separate the user interface, the business logic, and the database.  
The reason I made this choice is because it’s a common way of building modern applications, and it makes the system easier to maintain and scale. For example, if I later change the database from SQLite to SQL Server or MySQL, the rest of the app doesn’t need to be rewritten. This was also backed up by advice from a senior technical advisor.

**Report version:**

* **What:** Adopted a three-tier architecture (presentation, application, and data management tiers) separating user interface, business logic, and database.
* **Why:** This approach is standard in modern data-driven applications and was confirmed by the senior technical advisor as the most scalable and flexible option. It ensures that changes in one layer (e.g., UI or database migration from SQLite to MySQL) do not disrupt others, supporting maintainability and long-term extensibility.

Implementation decisions:

Data object or string to store the current data of the inspection form.

Decided to use the string data type to hold the Data instead of using the TypeScript Data object as it