**Agri-Energy Connect Platform**  
 **Non-Functional Requirements, Design Patterns, and Architecture Overview**

Programming 3A

Prog 7311

POE Part 1

ST10365374

Table of Contents

1. Introduction
2. Critical Non-Functional Requirements  
    2.1 Scalability  
    2.2 Security  
    2.3 Usability  
    2.4 Performance
3. Influence of Non-Functional Requirements on Development
4. Relevance of Design and Architecture Patterns
5. Recommended Architecture
6. Conclusion
7. References

Programming 3A

Prog 7311

POE Part 1

ST10365374

‌

1. Introduction

In response to South Africa's growing demand for sustainable agricultural and renewable energy integration, the need for innovative platforms and solutions is becoming increasingly vital. The Agri-Energy Connect platform has been conceptualized as a digital ecosystem. It aims to support farmers, green energy providers, and sustainability advocates by offering a unified space for collaboration, education, and access to technology. The platform will host sustainable farming resources, a green energy marketplace, educational content, and tools for collaborative project development and funding. The platform’s effectiveness depends on carefully selected non-functional requirements and the implementation of suitable design and architecture patterns.

2. Critical Non-Functional Requirements

2.1 Scalability

Scalability is vital to ensure the platform can grow with increasing user demand. The use of cloud-based infrastructure, such as Microsoft Azure or AWS, allows for horizontal scaling—adding more resources dynamically based on load. This ensures that the platform remains responsive and accessible as adoption grows (Gluo, 2023).

2.2 Security

Given the potential handling of personal and financial data, robust security measures are required. Secure authentication via OAuth 2.0, encryption of all data, and role-based access control (RBAC) will be used to safeguard user information and comply with data protection laws like South Africa’s POPIA. Measures will also include auditing, access logging, and user data control to further ensure compliance (Gluo, 2023).

2.3 Usability

To serve a wide audience with varying technical skills, the platform must prioritize usability. This includes designing an intuitive user interface, using plain language, and supporting mobile access. Adopting user-centered design principles and conducting usability testing will ensure that the system remains accessible and effective. (Gluo, 2023).

2.4 Performance

The platform should perform efficiently, especially during real-time interactions in forums, training sessions, and project collaborations. This can be achieved through caching mechanisms, optimized backend queries, and content delivery networks (CDNs) to speed up content access (Gluo, 2023).

3. Influence of Non-Functional Requirements on Development

These non-functional requirements influence multiple aspects of the development lifecycle. Security considerations require a DevSecOps approach to embed protection into every development phase. Scalability demands modular development even within a monolith. Usability influences UI/UX design priorities. Performance goals shape technology stack decisions, favouring fast and reliable frameworks like ASP.NET Core. A CI/CD pipeline will support efficient, secure, and continuous delivery, even in a monolithic structure, by enabling automated testing, build, and deployment workflows (Leffingwell, D. 2025).

4. Relevance of Design and Architecture Patterns

Importance of Patterns

Design and architecture patterns offer proven solutions to recurring development challenges. They promote code reusability, consistency, and scalability—key traits for enterprise systems like Agri-Energy Connect. By using these patterns, the platform can evolve as new features are added, without compromising system integrity or performance (Vpadmin, 2024).

Proposed Design Patterns

* Model-View-Controller (MVC): This pattern separates concerns between data (Model), interface (View), and control logic (Controller), supporting clean code structure and easier testing. It is particularly useful for web applications (Amin, R., 2024).
* Repository Pattern: This provides a layer between the application and the data access logic, enabling easier maintenance and testing of the data layer (Amin, R., 2024).
* Singleton Pattern: Ensures that only one instance of services such as logging or configuration exists, reducing memory consumption and avoiding resource conflicts (Amin, R., 2024).
* Observer Pattern: Useful for implementing real-time alerts and updates in the forum and project collaboration sections (Amin, R., 2024).

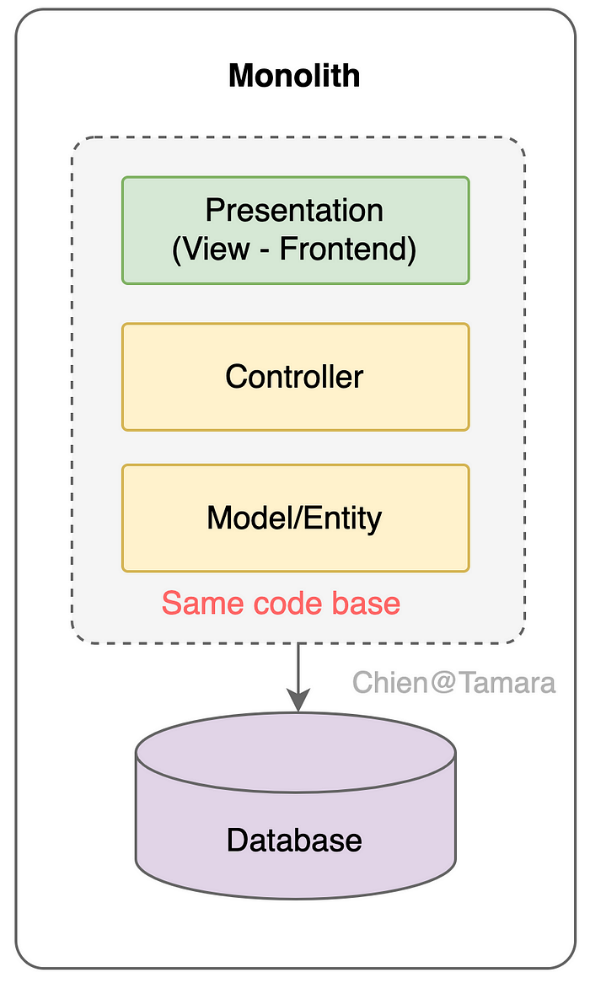
5. Recommended Monolithic Architecture

A Monolithic Architecture is selected as the prototype due to its simplicity, reduced overhead, and fast development lifecycle. All modules (Marketplace, Training, Collaboration) will reside within a unified application. The modular monolith approach enables future transition to microservices, aligning short-term efficiency with long-term scalability (Hoang, C, 2025).

Advantages:

* Fast Development and Growth: Because of the single codebase, monolithic applications can be created and implemented swiftly (Hoang, C, 2025).
* Code Reuse: Efficiency can be increased by reusing shared models or entities across the program (Hoang, C, 2025).
* High Degree of Uniformity: A consistent coding style is encouraged by this architecture (Hoang, C, 2025).

Monolith Architecture Pattern (Hoang, C, 2025).



All of the application's components, including the data access code, business logic, and user interface, are developed and assembled into a single database and repository in a monolithic architecture. Typically, this architecture makes use of ideas like the Model-View-Controller (MVC) design pattern and templates/themes. (Hoang, C, 2025).

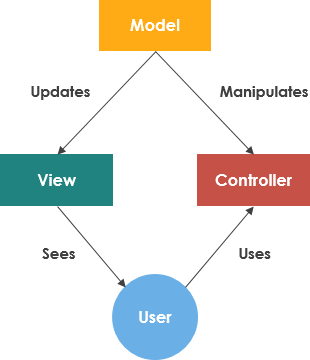
Monolith System Architecture:

A screenshot of a computer screen

AI-generated content may be incorrect.

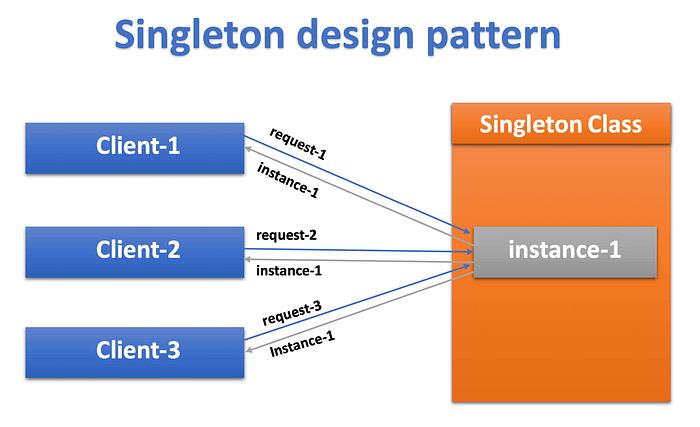
The Agri-Energy Connect Platform is built using a monolithic architecture structured into three main layers: the UI layer, the business logic layer, and the data access layer. The UI layer uses the Model-View-Controller (MVC) pattern to separate concerns and includes modules like the homepage, marketplace, training, and collaboration portal for user interaction. The business logic layer handles core operations such as user authentication using the Singleton pattern, real-time notifications using the Observer pattern, and manages projects and resource allocation. The data access layer implements the Repository pattern to interact cleanly with a centralized SQL database, which stores essential data like user profiles, products, and projects. This layered and pattern-driven approach ensures the system is organized, maintainable, and scalable while remaining within a single deployable unit.

Model View Controller Design (Visual Paradigm, 2024).



The Model-View-Controller (MVC) software framework encourages an organized approach to application development by dividing concerns into three separate components: a model for data and business logic, the view for the user interface, and the controller for input handling. This decoupling promotes loosely linked architecture, which allows developers to handle complexity by concentrating on certain areas of the program. This separation enables simultaneous development, increased reusability, scalability, and lower coupling between components (Visual Paradigm, 2024).

Singleton Pattern (Jamdade, P. 2022).



The Singleton Pattern is a software architecture technique that ensures a class has only one instance and provides a centralized access point to that object. For example, when a user logs in, the application can consistently retrieve their details via a Singleton, ensuring shared services like logging, caching, or configurations are optimized for performance (Jamdade, P. 2022).

Observer Pattern (‌Durán, E, 2019)

A diagram of a diagram

AI-generated content may be incorrect.

The Observer pattern in the Agri-Energy Connect platform facilitates real time updates between system components with minimal coupling. In this implementation, the Collaboration Module acts as the Subject, managing the state of posts, project updates, and user interactions. Observers are the registered users or services that have subscribed to specific threads, projects, or content updates. When the subject’s state changes such as, when a new comment is added to a project discussion or a project status is updated, the subject automatically notifies all subscribed observers. This is accomplished using a subscribe/notify mechanism, ensuring that only interested parties receive updates. This decoupled approach promotes scalability and responsiveness, especially in dynamic sections like forums and collaborative workspaces (Durán, E., 2019).

6. Conclusion

To summarize, the Agri-Energy Connect platform’s success relies on solid non-functional foundations such as scalability, security, usability, and performance. Integrating design patterns like MVC, repository, singleton, and observer within a monolithic architecture ensures a robust, maintainable, and user-friendly solution. This architecture allows for faster prototyping and future adaptability, aligning the project with its vision of connecting the agriculture and green energy sectors in meaningful and scalable ways.

7. References

Gluo, 2023. *Non-functional requirements: Why are they important?* [online] Available at: <<https://www.gluo.mx/en-US/blog/requisitos-no-funcionales-por-que-son-importantes>>[Accessed 2 Apr. 2025].

‌Leffingwell, D. 2025. *Nonfunctional Requirements*. [online] Available at: <<https://framework.scaledagile.com/nonfunctional-requirements/>> [Accessed 2 Apr. 2025].

Vpadmin, 2024. *The Importance of Patterns in Enterprise Architecture: What, Why, and How - Visual Paradigm Guides*. [online] Available at: <<https://guides.visual-paradigm.com/the-importance-of-patterns-in-enterprise-architecture-what-why-and-how/>>[Accessed 2 Apr. 2025].

Amin, R.,2024. *After the last post about the Command Design Pattern, a few questions were asked. One important question was: Why should we follow Software Design Patterns, or what are the benefits? Another obvious question was: Which Design Pattern is best suited for a specific type of application? Instead of sepa*. [online] Available at: <<https://www.linkedin.com/pulse/design-patterns-top-10-software-best-use-cases-recommended-ruhul-amin-dibqc/>> [Accessed 7 Apr. 2025].

Richardson, C, 2020. *Microservice Architecture pattern*. [online] Available at: <<https://microservices.io/patterns/microservices.html>> [Accessed 7 Apr. 2025].

Visual Paradigm, 2024. *What is Model-View and Control?* [online] Available at: <<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-model-view-control-mvc/>> [Accessed 7 Apr. 2025].

Jamdade, P, 2022. *Singleton Design Pattern in Android Explained*. [online] Available at: <<https://medium.com/@Prachi-Jamdade/singleton-design-pattern-in-android-explained-5c46f6cd3448>.> [Accessed 7 Apr. 2025].

‌Durán, E, 2019. *Diving into the great observer pattern in javascript*. [online] Available at: <<https://enmascript.com/articles/2019/03/09/diving-into-the-great-observer-pattern-in-javascript/>> [Accessed 7 Apr. 2025].

‌

Sapna Nair, H, 2025. *Top Microservices Design Patterns for Microservices Architecture*. [online] Available at:<<https://www.lambdatest.com/blog/microservices-design-patterns/>> [Accessed 7 Apr. 2025].

Hoang, C. 2023. *Monolith Architecture*. [online] Available at: <<https://tech.tamara.co/monolith-architecture-5f00270f384e>> [Accessed 8 Apr. 2025].