Reflective report on evolution of your design architecture throughout the project. Consider the maintainability of your system (including explicit examples of the impact of future development) and future professional practices

Throughout the project the structure of our high-level architecture changed as we added more layers and patterns to make our system more maintainable and have better performance.

In the presentation layer, we changed the servlets to controller-view structure with the template view pattern and the front controller pattern. The addition of these two patterns makes the structure of the presentation layer clearer, makes the system more expandable, and increases the flexibility of system performance. This increased the maintainability of our system as future features which we would want to implement can be easily implemented by adding a view of the feature and corresponding command classes. Previously with servlets they had the difficulty of adding new features and also repetitive code structure within servlets such as error handling.

In order to ensure secure access to the data within our system, we add security patterns into the system, including authentication enforcer pattern, authorisation enforcer pattern, and secure pipe pattern. These security patterns are used to handle the authentication and authorization of users, and confidentiality of the users' requests. They improve the maintainability of the system as the code has been encapsulated into a single place to support reusability, and also decrease the likelihood of a successful attack from the opponents. With future development this means that the security component can be easily modified with other libraries without affecting the overall performance of the system while still ensuring that the overall system is secure.

The maintenance of session state in our system is supported by implementing the client session state into the system. The reason we choose the client session state is because it is easy to implement and supports stateless server objects well by giving them maximum clustering and failover resilience. However, there may be issues when dealing with large amount of data as the cost of where the data is stored and the time it takes to transfer all of it on each request may become very high. It also has a security issue as the data sent to client is vulnerable to being monitored and changed. Although the security issue can be covered by security patterns, there are still some risks that sensitive data can be attacked. Hence, the server session state can be used to avoid these issues in the future. Server session state stores all state in a replicated server side datastore, which can fully control the sessions without considering the issue of data size and hidden the implementation and user details. Due to the weaknesses of client session states within our system this could impact future development as the session state may needed to be modified if the traffic of our system increases if we were to commercialise the enterprise system as the system may be unable to handle significant data.

For service layer, domain logic layer and data mapper layer, there was not much changes over the duration of our project as we only added single classes to implement the feature B which were AdminService class, AdminUser class, and AdminMapper class. The use of model-view-controller pattern showed that our system was easily maintainable and that additional features could be added without impacting the overall system or having to change other classes to

accommodate these new features as every layer is decoupled from each other. If we were to implement additional features such as guest users or other features within existing users our model-view-controller will allow us to extend our current architecture easily without significant risks of the overall system breaking or requiring significant modification.

In order to deal with concurrency, we implemented pessimistic offline lock pattern and implicit lock pattern into the system. Implicit lock is quite important as it aims to prevent forgetting the lock by putting the code for acquiring and releasing the lock in the mapper layer. Pessimistic locking locks a record for exclusive use until completion resulting in better integrity but increases the risks of deadlocks for the application. It also needs a direct connection to the database to complete the lock. If someone who reads a record and intending to update it placing an exclusive lock on the record, the user may be locked out for a long time if no one release the lock, slowing down the response of the entire system and causing frustration. For our enterprise application system, optimistic lock might be a better choice as it is faster and it is not necessary to maintain a connection to the database for the session, which is suitable for the system have many layers of architectures. However, we also have to take consideration that due to the nature of our system an optimistic lock could also result in increased application latency in the scenario of multiple administrators editing the system. Therefore, future choices in concurrency patterns needs to be considered carefully by weighing the pros and cons of each locking pattern. From the perspective of architecture design, the implementation of lock also has some disadvantages. It increases the instability of the system as the best balance between lock overhead and lock contention is unique to the application and is sensitive to design, implementation, and even underlying system architecture changes. These balances can change over the life of an application and can lead to dramatic changes in updates. hence, implementation of lock patterns needs to be considered carefully in the future when design the architecture as it may impact a lot on maintainability of our system.

Overall, our system has achieved some improvements in overall performance, but the simplistic nature of the system meant that some of the patterns used increased the performance latency while principles of system performance that we did not consider could be beneficial for our system performance. Our system exhibits some maintainability and extensibility for future development through mode-view-controller patterns but features such as choices in concurrency locks and maintenance of client session state may need to be reviewed and potentially remodified for better maintenance and extensibility.