Research Summary & Reflection on MVVM in Flutter

I. Research Summary:

- MVVM separates UI (View), state/business logic (ViewModel), and data (Model).
- Improves maintainability, testability, and reusability.
- Design patterns like Singleton, Factory, and Builder are useful in structuring scalable apps.

II. Reflection

During this project, I implemented a simple to-do list app using the MVVM pattern in Flutter. This structure made the code more organized. The View was kept lightweight, only handling UI concerns, while the ViewModel managed state changes and interactions with the repository. Implementing ChangeNotifier and using Provider was an insightful experience—it allowed the UI to react to updates automatically.

Initially, it was tempting to place logic in the View, but sticking to the MVVM approach encouraged me to separate responsibilities properly. This not only made the code more readable but also showed how unit testing the ViewModel independently becomes easier.

I also gained an appreciation for design patterns like Singleton (for central resources), Factory (for dynamic object creation), and Builder (for constructing complex objects), and how they relate to mobile app architecture. One challenge was keeping the repository logic simple while simulating data persistence with a local list.

Overall, this assignment helped me learn how to write scalable, clean Flutter code using MVVM.

III. Conclusion

Working on this assignment gave me my first real hands-on experience with the MVVM (Model-View-ViewModel) design pattern in a Flutter project. At first, I found it a bit confusing to separate the responsibilities between the View, ViewModel, and Model. It was tempting to put logic directly into the UI, especially because it feels quicker at the beginning. But as the app started to grow—even for a simple to-do list—I quickly realized how helpful MVVM is in keeping the code clean and easy to manage.

The **ViewModel** layer turned out to be the most valuable part. It acted as the middleman between the UI and the data, allowing me to write logic in one place and simply "reflect" those changes in the UI using Provider. It was satisfying to see how just calling **notifyListeners()** could update the entire screen without writing a lot of boilerplate code.

Another thing I learned was how useful **ChangeNotifier** is when paired with **Provider**. It helped me manage state efficiently while keeping the UI reactive. Although it took some trial and error to structure things correctly, once I understood the pattern, everything started to make more sense.

One of the challenges I faced was figuring out where to place certain methods—should it live in the ViewModel or in the Repository? I eventually learned that the Repository should only handle raw data operations (add, delete, update), and the ViewModel should control when and how those operations are triggered.

In the end, this assignment didn’t just help me understand MVVM—it helped me write Flutter code that’s more modular, testable, and ready to scale. I can now see why MVVM is considered a best practice, especially in mobile development where things can get messy quickly without clear structure.