NANYANG TECHNOLOGICAL UNIVERSITY

CZ3003 - Software System Analysis & Design

Architecture Design

Project Name: Food Wars

Group Name: Team 1

Lab group: TDDP3

Date of Submission: 03 October 2021

Group Member	Matric No.
David Tay Ang Peng	U1910603L
Grace Ong Yong Han	U1721575H
Jordon Kho Junyang	U1920297F
Lim Wei Rong	U1921791D
Ryan Tan Yu Xiang	U1922774F
Joy Cheng Zhaoyi	U1922716L
Tang Hoong Jing	U1721417E
Guo Wan Yao	U1822530E
Ng Wee Hau, Zaphyr	U1822044D
Lee Kai Jie, John	U1921862J
Chio Ting Kiat	U1720465K

Table of Contents

1. Introduction **4**

2. Potential Architectures **5**

2.1 Microservices Architecture 5

2.1.1 Description 5

2.1.2 Benefits 5

2.1.3 Trade-offs 6

2.1.4 Key usage scenarios 6

2.2 Client Server Architecture 8

2.2.1 Description 9

2.2.2 Benefits 9

2.2.3 Trade-offs 9

2.2.4 Key usage scenarios 10

2.3 Model-View-Controller Architecture 11

2.3.1 Description 11

2.3.2 Benefits 12

2.3.3 Trade-offs 12

2.3.4 Key usage scenarios 12

2.4 Layered Architecture 14

2.4.1 Description 14

2.4.2 Benefits 14

2.4.3 Trade-offs 15

2.4.4 Key usage scenarios 15

3. Selected Architecture **17**

3.1 Comparison of the 4 architecture models 17

3.2 Rationale Behind Choosing Client-Server Architecture 17

3.2.1 Advantages 17

3.2.1.1 Scalability 17

3.2.1.2 Parallel Execution 17

3.2.1.3 Decoupling 17

3.2.1.4 Flexibility 18
3.2.2 Trade-offs 18
3.2.2.1 Congestion 18
3.2.2.2 Lack of Robustness 18
3.2.2.3 High Cost 18
4. Subsystem Interface Design 19
5. Updated Class Diagram 27

1. Introduction

Architectural design for a system is a process to identify subsystems and the framework for subsystem control and communication. It describes the major components in the system, their relationships and how they interact with each other. To decide on a suitable architecture design, we will adhere to the 4 software design principles, namely: Modularity, Single Responsibility Principle, Principle of Least Knowledge and Open Close Principle.

For our Food Wars Application, the key requirements that we would need to satisfy are as follows:

1. **Scalability**
   The game system has to cater to a huge number of students which will require a lot of resources. The system's performance should not be affected by the increased workload.

2. **Parallel execution**
   The game system will require multiple components to run simultaneously and thus will need them to be executed in parallel.

3. **Testability**
   The game system should allow for ease of testing for each of the components or subsystems to ensure that bugs are not present in the system

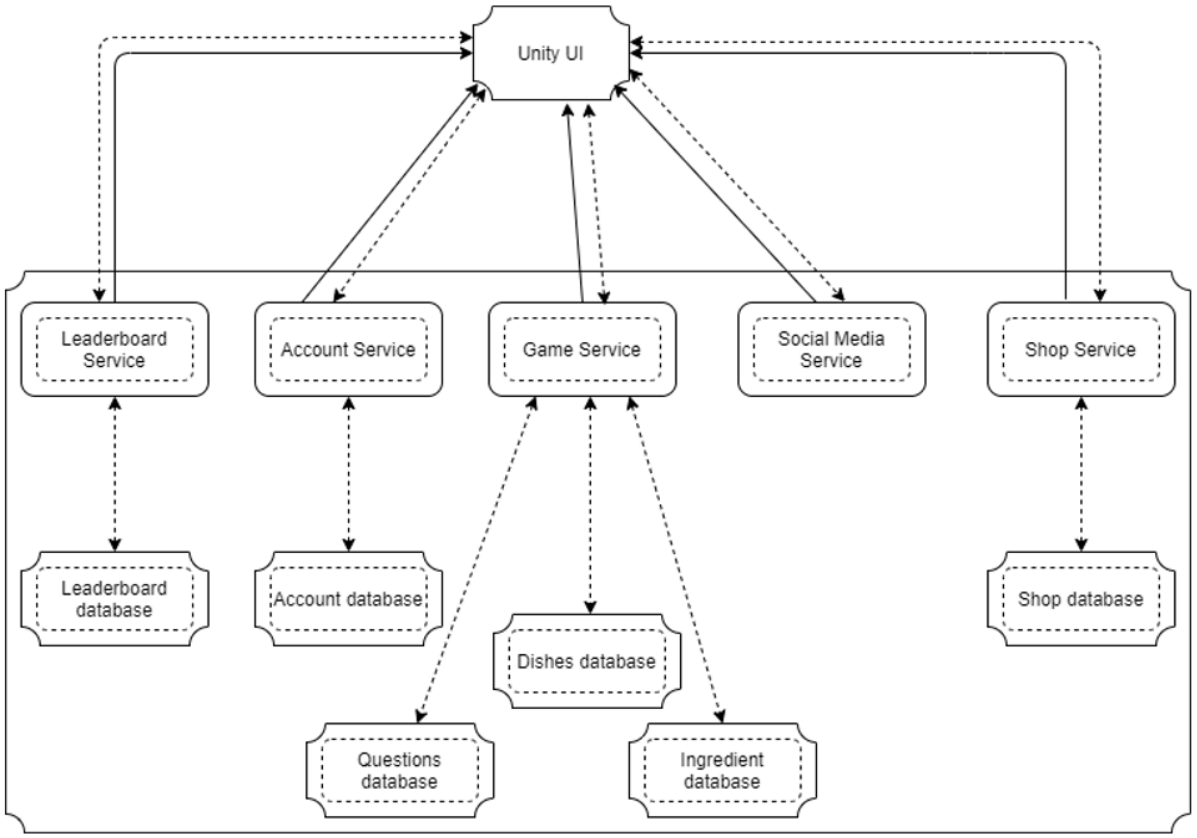
4. **Flexibility**
   The game system should be flexible for adding on new features, functions or capabilities in the future as the game becomes more complex.

5. **Decoupling**
   The game system should have low coupling and high cohesion so that the individual components can be easily modified when required.

In the next section, the document discusses the candidate architectures, followed by their benefits and trade-offs.

2. Potential Architectures

2.1 Microservices Architecture



2.1.1 Description

Microservices Architecture is a service-oriented architecture, a subtype of distributed systems architecture. It is focused on splitting a large, complex system vertically (either by functional or business requirements) into smaller sub-systems of independent processes. These processes communicate with each other via lightweight, language-agnostic network calls that can be either synchronous or asynchronous.

The most important aspect within this architecture is the independent processes, called *service components*, which contain a number of modules to serve a single purpose. Each service is autonomous, self-contained and has varying granularity. Hence, it is crucial to design the right level of service component granularity for the microservices architecture.

This form of architecture is often used when there are multiple teams working on various functions of an application, and when the application is relatively large scale.

2.1.2 Benefits

The benefits of the Microservices Architecture are as follows:

1. Each microservice is **small** and **loosely coupled**. This ensures that each service is a separate codebase where a small team of developers can work on.
2. Each microservice can be **deployed independently**. This ensures that a team can update an existing service without redeploying or rebuilding the entire application. An added benefit is that it is easy to identify faults within the entire application as the bug can be easily isolated from the rest of the application.
3. Since each microservice uses its own database, there is **data isolation** between the different services which simplifies schema updates, where changes to a database will not impact any of the other services. An additional benefit is that each microservice is able to use a database that is best suited to its needs.

2.1.3 Trade-offs

The trade-offs of the Microservices Architecture are as follows:

1. **Communication between services** is complex. As the services are made to be independent, it may be difficult to handle requests that move from one service to another.
2. It is challenging to maintain and **manage multiple databases** across all the different microservices.
3. **Testing** is more difficult as compared to monolithic architectures as it is challenging to test service dependencies and refactor across service boundaries, especially when the application is evolving quickly.
4. **Data integrity** has to be maintained. Data consistency may end up being inconsistent across the various databases.

2.1.4 Key usage scenarios

Scenario 1: User visits the shop and buys something

The user will first access the shop via Unity UI to shop service. If the user purchases something from the shop (e.g A power up), the shop database would then have to be updated followed by communication to the account database through the account service through Unity UI.

Scenario 2: User wins a multiplayer game and is newly added to the leaderboard, and the user wants to share this news with his friends on social media.

The user will exit the game service back to Unity UI and the system would have to update the leaderboard database via the leaderboard service from Unity UI. After which, the user assesses the social media service which connects to the user's social media platform to share the news.

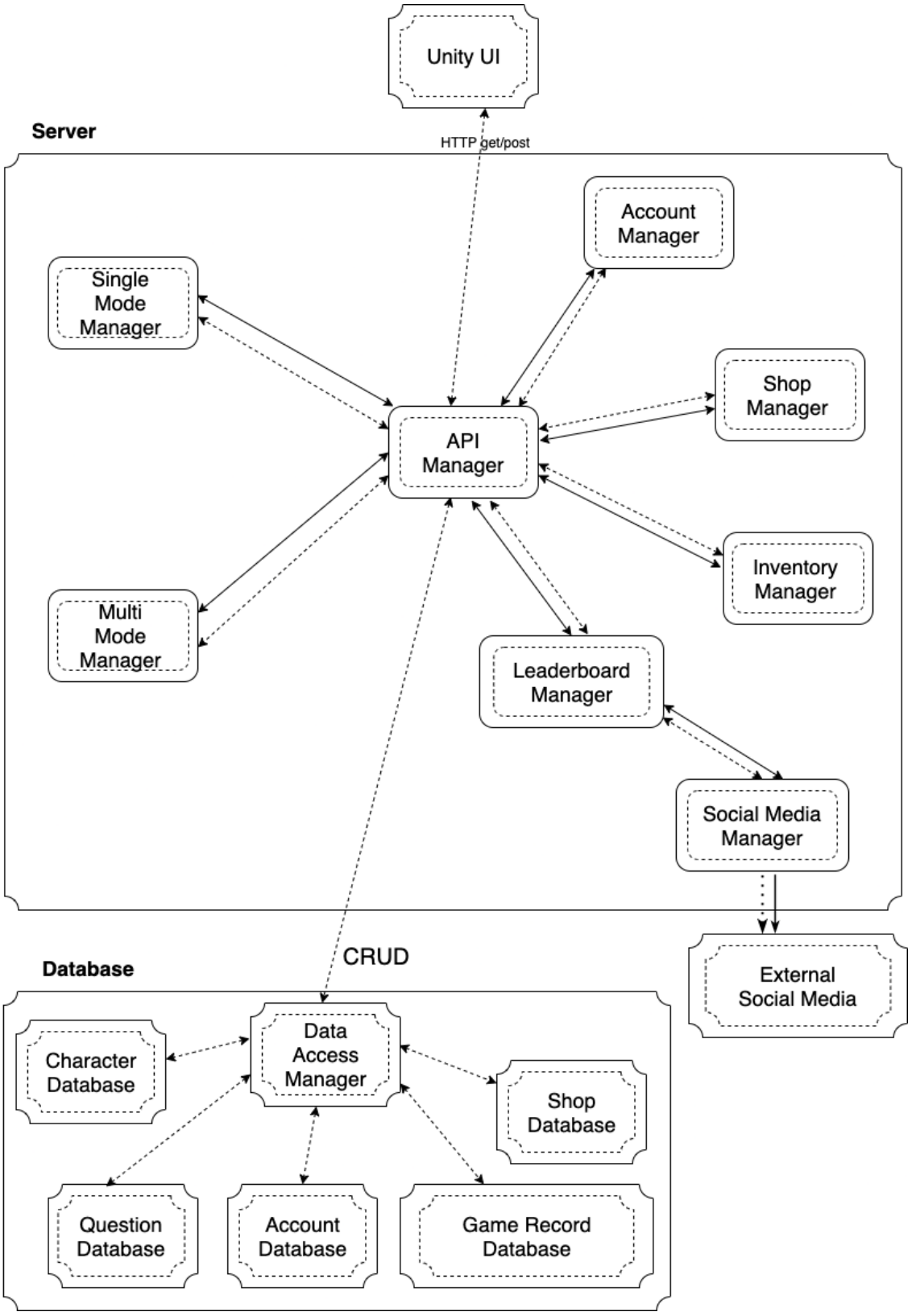
Scenario 3: Users complete a single player level and move on to the next level.

The system would need to save the progress in the account database from account service via Unity UI.

Evaluation:

As we can see from the above multiple scenarios, all of the key usage scenarios would have to go through Unity UI, as each microservice is independent. Thus, this creates a vulnerability if Unity UI is down, the whole system architecture would be jeopardized. It is however noteworthy that if one of the other services other than Unity UI fails, the system can still function. But if that happens to be the game service, then the user can't play the game, which is the main service that we are trying to provide, and this beats the point of our software, thus this isn't really a huge benefit. What is really important is the syncing of databases between all of the services as the databases should all be telling the same thing. Having multiple databases that need to be updated would mean that it allows for the possibility of data integrity being breached, and hence this wouldn't be as good as an architecture with a single source of truth.

2.2 Client Server Architecture



2.2.1 Description

Client-server architecture, architecture of a computer network in which many clients request and receive service from a centralized server

Client computers provide an interface to allow a computer user to request services of the server and to display the results the server returns. Clients are often situated at workstations or on personal computers.

Servers wait for requests to arrive from clients and then respond to them. Ideally, a server provides a standardized transparent interface to clients so that clients need not be aware of the specifics of the system (i.e., the hardware and software) that is providing the service. Servers are located elsewhere on the network, usually on more powerful machines.

2.2.2 Benefits

The benefits of the Client Server Architecture (CSA) are as follows:

1. **Centralised Control** that CSA is integrated with. All necessary information is placed in a **single location**. This is particularly advantageous for the network administrator, who has complete control over management and administration. Any problem that arises throughout the entire network can be **resolved in one location**. Also as a result of this, **upgrading resources and data has gotten a lot easier**.
2. Because of its **centralized architecture**, the data in a client-server network is **effectively protected**. It can be enforced with **access controls**, allowing only authorized people access. Imposing credentials such as login and password is one such approach. Furthermore, if the data is lost, the files can be **retrieved quickly** from a **single backup**.
3. **Scalability** is a key feature of CSA. The quantity of resources available to the user, such as clients and servers, can be increased at any time. As a result, the server's capacity can be increased **without much interruptions**. Because the server is **centralized**, permission to network resources is not a concern even if the size grows. As a result, very few staff are required for the configurations.
4. It is rather **easy to manage files** because they are all kept on a **centralized server**. The **optimal management** for tracking and finding records of essential files is found in CSA.
5. Every client is given the option to log into the system, **regardless** of their **location or platform**. Employees will be able to access business information **without** having to use a **terminal mode or a processor** in this manner.

2.2.3 Trade-offs

The trade-offs of the Client Server Architecture are as follows:

1. The most significant disadvantage of Client Server Architecture is **traffic congestion**. Too many clients making requests from the same server will cause the **connection to fail or slow down**. An **overburdened server** causes numerous issues with information access.
2. Since Client Server Architecture is **centralized**, in the event that the primary server fails or is interfered with, the **entire network** will be **interrupted**. As a result, this architecture is lacking in terms of **robustness**.
3. The **cost** of setting up and maintaining a server in Client Server Architecture is **typically higher**. Because the networks are so powerful, they can be **costly to acquire**. As a result, not every user will be able to afford them.

4. Client Server Architecture will **work nonstop** once the servers have been installed. That is to say, it **requires proper attention**. If any issues arise, they must be **resolved immediately**. As a result, the server should be maintained by a specialized network manager.

2.2.4 Key usage scenarios

Scenario 1: User visits the shop and buys something

For now, the shop use case is not particularly complex, involving just the purchase of power ups, in transactions that happen only between the game and the player. Therefore, CSA is rather fitting for the use-case as there is direct linkage between a client machine and the server. Given that transactional integrity is crucial in a financial transaction, having the data stored and manipulated at a single point - the server - is ideal.

However, if there were future development plans for player-to-player transactions, depending on the nature of these transactions, an architecture that supports client-to-client connections might be more desirable. Still, this would still be a very niche situation, as most player-to-player transactions are not time-sensitive enough to warrant a client-to-client connection.

Scenario 2: User wins a multiplayer game and is newly added to the leaderboard, and the user wants to share this news with his friends on social media.

Similar to the previous use-case, the integrity of the leaderboard is crucial. Therefore it is important that any reference to a leaderboard should come from a single source of truth maintained by a central server, rather than each player having a local copy of the leaderboard that is periodically synchronized with a master leaderboard.

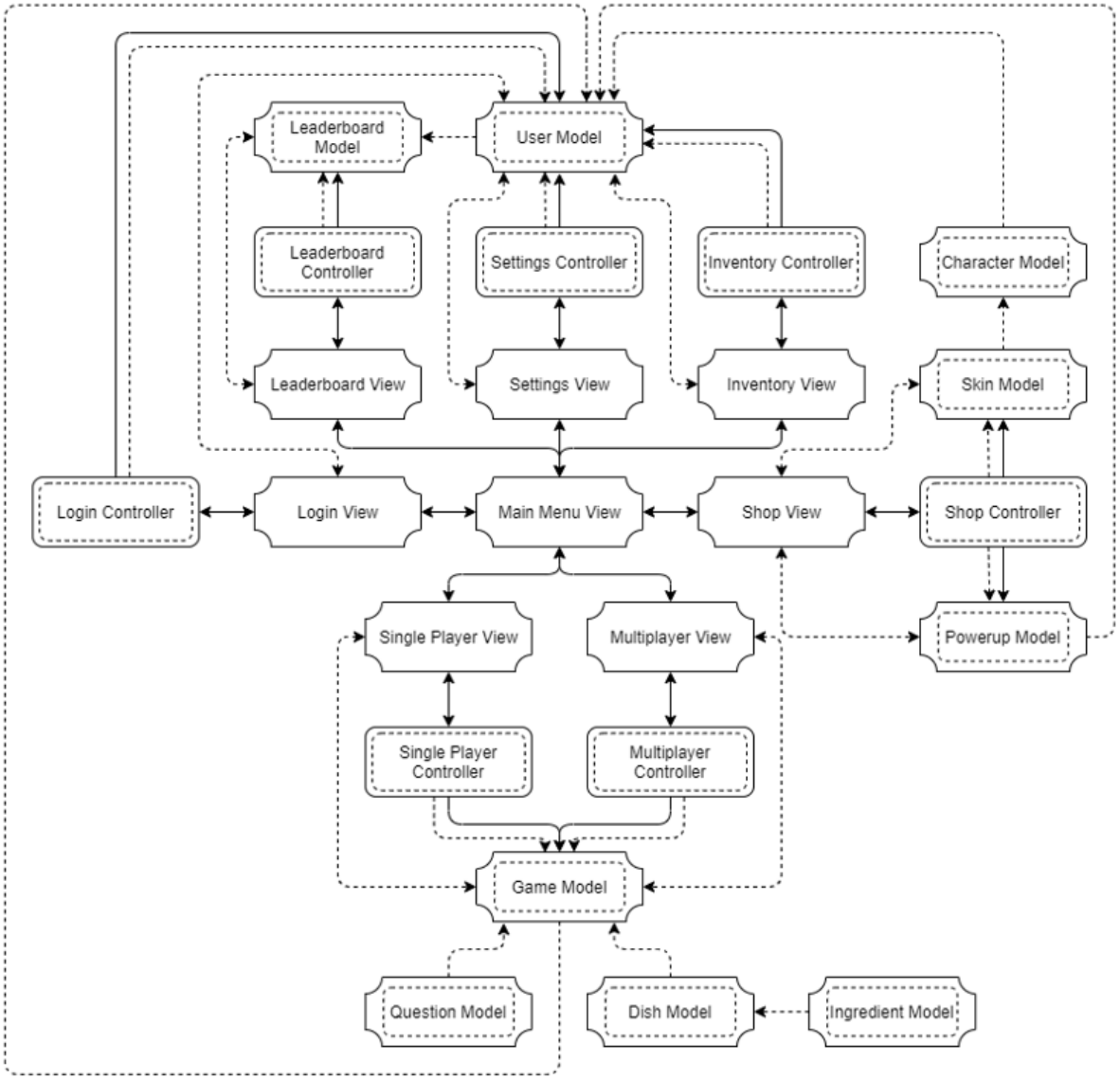
Scenario 3: User completes a single player level and moves on to the next level.

Given this is a single-player scenario, again this boils down to a communication channel involving only a single client and the server, where the direct connection between client and server offered by CSA comes in handy.

Evaluation:

As most of Food Wars is a single-player experience, the direct connection between a client and a single server serves this system well. In this case, CSA provides a solution architecture that achieves the desired outcomes with minimal complexity. In scenarios involving multiple players (multiplayer game mode, leaderboard, etc.), while these are situations played out in real-time, the speed of data transfer in maintaining the real-time nature of the system is not the priority. Instead, system integrity is decidedly more important, hence CSA still stands as a very viable system having a single central server to act as a single source of truth.

2.3 Model-View-Controller Architecture



2.3.1 Description

Model-View-Controller (MVC) architecture is designed to facilitate interactive intensive systems where multiple presentations are used to display the same shared information. The user interface is susceptible to change and is loosely coupled with its functionality and data.

MVC consists of 3 components - model, view, and controller. Models contain control and entity classes and are in charge of the app logic and data. Views define how data is presented and each view provides an alternative of the same model. Controllers capture user input and pass them to the view and model. The separation of models from views and controllers allow for multiple views of the same model. When the user changes the model using the controller of one view, all other views using the same model will be updated to reflect the changes.

MVC incorporates 2 design patterns - observer and strategy. The observer pattern describes the view-model relationship. Views behave like observers and subscribe to a model. When there are

changes in the model made by the user, it notifies all views and their displays are updated. The strategy pattern describes the view-controller relationship. Views act as contexts that employ several strategy interfaces for users to modify data. Each controller serves as a different strategy interface, and is implemented depending on the user's interface behaviour.

2.3.2 Benefits

The benefits of the Model-View-Controller (MVC) Architecture are as follows:

1. MVC has **high cohesion** as its components are logically grouped together. Related actions are grouped together on a controller, and views subscribing to the same model are grouped together.
2. There is **low coupling** between models, views or controllers.
3. MVC supports **simultaneous development**. Different components can be developed in parallel.
4. MVC provides **clean separation of concerns (SoC)**. Models handle the app logic and data, while controllers and views handle the interface. Changing the interface should not affect the app logic, and vice versa.

2.3.3 Trade-offs

The trade-offs of the Model-View-Controller (MVC) Architecture are as follows:

1. MVC has **poor code navigability** and a **pronounced learning curve**. The framework navigation can be complex as it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
2. MVC requires **multi-artifact consistency**. As a single feature is decomposed into three artifacts (model, view and controller), developers are required to maintain the consistency between these artifacts.
3. To support simultaneous development, **multiple developers** are required for parallel programming of different components.
4. It is **difficult to reuse** components and **perform unit testing** on the MVC.

2.3.4 Key usage scenarios

Scenario 1: User visits the shop and buys something

This scenario involves the shop view, shop controller and other models depending on the item bought, such as power-up or skin models. As MVC is highly modularised, only the shop controller will be activated to handle user interactions. The respective models will be updated to keep track of the user’s inventory and currency, and the shop view will be updated to reflect this change. Due to high cohesion and low coupling, only relevant components are used and updated.

Scenario 2: User wins a multiplayer game and is newly added to the leaderboard, and the user wants to share this news with his friends on social media.

Once the multiplayer game ends, the leaderboard model is updated using the game results. When the user goes to the leaderboard view, it will display the data from the updated leaderboard model. When the user clicks on the share button, the leaderboard controller activates the sharing functionality and alerts an external social media API to perform the sharing.

Scenario 3: User completes a single player level and moves on to the next level.

After the user completes a single player level, the user model is updated to store the user's progress. The single player view reflects this change by displaying the option to continue playing the next level. When the single player controller is used to start the next level, the game controller & game model & user model is invoked to display a new set of questions with different dishes and ingredients based on the user level.

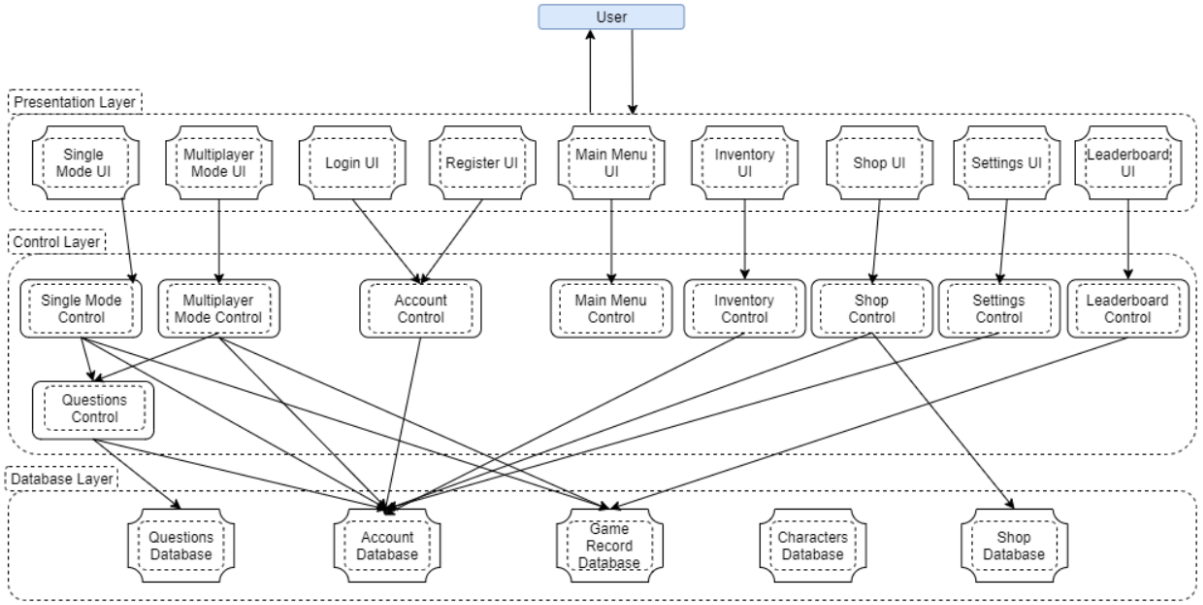
Evaluation:

There is a clean separation of concern. For instance in scenario 1, the shop view is only incharge of showing data (items sold in the shop) & responding to events. For example, when the user clicks on the buy button, users would be redirected to the confirmation pop-up. The shop controller would be in charge of processing the data after getting a request from the shop view. Lastly, the shop model would work directly with the database. It does not have to deal with user interface or data processing as it would have been handled by the view & controller respectively.

In the MVC Architecture, **views could be overburdened with update requests.** For instance in scenario 2, when a user accesses the leaderboard, if the leaderboard model keeps on going with frequent changes (leaderboard ranking keeps changing), the view component could fall behind on updates.

Multiple Controllers & Models are invoked for a simple action. For instance in scenario 3, when a user completes a single player game, the user model needs to be updated & store the user's progress & new level. The single player controller, single player view will also be invoked to display this information. The game model would also need information from the user model to understand which data to pass to the game controller before it is displayed on the screen by the single player view for the next level. This reflects the complexity of using a MVC Architecture, which is better for simple softwares.

2.4 Layered Architecture



2.4.1 Description

The Layered Architecture, also known as the n-tier architecture pattern, is one of the most commonly used architecture systems in software engineering. The pattern comprises horizontal layers that each perform a specific role and responsibility. Components are grouped based on their functionalities and placed into the horizontal layer that describes its function. The main idea behind the Layered Architecture would be separation of concerns via source code organization as different functionalities are performed by different layers and components and there is no component that performs multiple functionalities spanning across various layers.

In the Layered Architecture, the layers are closed which means that a request would have to move from one layer to the layer below it and it cannot skip layers to reach non-adjacent levels. This is a key feature of the Layered Architecture called the layers of isolation concept which ensures that changes made in one layer generally does not affect other layers except for possibly itself and associated adjacent layers.

With the above features, Layer Architecture encourages low coupling as components are only linked to components from adjacent layers as well as high cohesion as each component and layer has a specific functionality that it performs and not a variety of functionalities.

2.4.2 Benefits

The benefits for Layered Architecture is as follows:

1. High ease of development.
    a. By separating the application's layers by skill sets, this pattern becomes a natural choice for many application's development. Allowing for easy allocation of tasks, with clear objectives for all team members.
2. High testability
    a. Due to the modularity of having specific layers, differing layers can be ignored during the testing process, making testing of components easy.

2.4.3 Trade-offs

1. Low overall agility.
    a. The application is not able to respond quickly to change. Any changes in the code is made difficult due to the monolithic nature of most components as well as the tight coupling of their components.
2. Low performance
    a. While layered architectures can perform well, high performance applications are not possible due to the inefficiencies of having to go through multiple layers for each request.
3. Low scalability
    a. Due to tightly coupled and monolithic implementations of this pattern, the overall granularity is too broad, making it expensive to scale.

2.4.4 Key usage scenarios

Scenario 1: User visits the shop and buys something

The user will access the shop via the *Shop UI* first (Presentation layer). The subsequent layer's *Shop Control* (Control layer) will then determine what to display in the shop interface based on the data from the last layer's (Database Layer) *Account Database* to check what the user already owns and shop database to see what items are available to display the items that the user can buy from the shop. Finally, if the user buys an item, the *Shop Control* will then update the *Account Database* so that the account owns the item bought.

Scenario 2: User wins a multiplayer game, and the user wants to share this news with his friends on social media.

The system would display the results and the share results button on the *Multiplayer Mode UI* (Presentation layer). Should the user choose to share the results, they can click on the button which will call the processing logic in *Multiplayer Mode Control* to share the results via their desired messaging application. (Control layer)

Scenario 3: User completes a single player level and moves on to the next level.

The system will show the results of the level on the *Single Mode UI* (Presentation Layer), after which it will call the processing logic in *Single Mode Control* (Control Layer) to save the user's progress in the *Account Database* (Database Layer) so that the system can track the user's progress.

Evaluation:
There is a clean separation of concern, for example if the user wants to purchase something from the Shop:

He will interact with the *Shop UI* (Presentation layer) to select the item he wants to purchase. Upon clicking on the purchase button, the processing logic in *Shop Control* (Control Layer) will be called and finally the *Account Database* (Database layer) will be updated.

For every major step there is a clean separation of concern which leads to easy development and testability due to the architecture's modularity.

However, due to the low agility and scalability of this architecture we have opted not to choose this architecture.

3. Selected Architecture

3.1 Comparison of the 4 architecture models

The table below outlines whether each of the candidate architectures are able to satisfy the requirements set in section 1. A tick (✓) denotes “Yes”, a cross (x) denotes “No” while a dash (-) denotes “Neutral”.

	Microservices	Client Server	Model-View-Controller	Layered
Scalability	x	✓	x	x
Parallel Execution	✓	✓	-	-
Testability	x	-	x	✓
Flexibility	✓	✓	✓	✓
Decoupling	✓	✓	✓	x

3.2 Rationale Behind Choosing Client-Server Architecture

3.2.1 Advantages

3.2.1.1 Scalability

The quantity of resources available to the clients and servers can be increased easily without any interruptions. Horizontal scaling and vertical scaling are both supported by this architecture. Horizontal scaling involves adding more client workstations without having an observable impact on the performance. Vertical scaling involves migrating the entire server to a faster one or by adding more server machines.

3.2.1.2 Parallel Execution

Since all the components in this architecture are independent and they communicate via messages, different processes are able to execute simultaneously. Each process is not affected by the progress of other processes. For example, when the user uses a power-up during a single player mode, the inventory manager can update the player’s inventory while the single player manager continues the game for the player uninterrupted.

3.2.1.3 Decoupling

The autonomous layers in the client server architecture allow for separation of concerns by loosely coupling components. By organizing the subsystems in this manner, the components within the specific subsystem would also adhere to the Single Responsibility Principle, dealing with only one facet of a feature. For example, the account manager in the server layer is only used for creating and

authenticating users while the question manager is only in charge of selecting the questions to be used for each level.

3.2.1.4 Flexibility

This architecture allows new components to be easily added as all other components are following the Single Responsibility Principle, hence the new feature can be inserted into its respective subsystem and function with the rest of the system. For instance, if a new feature (creating a forum) is added into the game, we will just need to add an additional manager into the server (to control the forum), as well as a forum database (to store forum posts) and connect it to the factory database for this new feature to work with the other features.

3.2.2 Trade-offs

3.2.2.1 Congestion

When multiple clients make requests from the primary server, the network may be overloaded which can result in slow or failed connections. Large amounts of requests may also result in an overburdened server which can lead to issues in accessing information. Since the project is still in an early development phase, high traffic volume is a very unlikely scenario at this point this time. Hence, its effects on the project are minimal.

3.2.2.2 Lack of Robustness

The nature of this architecture results in a highly centralised framework. In the event where the primary server fails or experiences interference, the network will be interrupted and all clients will be affected.

3.2.2.3 High Cost

It is costly to set up and maintain the primary server in this architecture. The network operations are very powerful, so they can be very costly as well. Future scaling and upgrading of this architecture will incur more costs in the long run too.

4. Subsystem Interface Design

Game Subsystems:

Subsystem Interface Name	Single Mode Manager
Public Method 1	chooseRestaurant(selection, primaryLevel)
Rationale	The difficulty of questions differs depending on the restaurant that the user has selected
Parameters Required	● **selection** which indicates the user's selection ● **primaryLevel** which indicates the user's current primary level. Users cannot select restaurants that are beyond his current level of studies
Usage Scenarios	1. User selects on the single player game mode 2. System directs users to the available restaurant page 3. User selects a restaurant 4. System directs user into the selected restaurant
Public Method 2	startSingleGame()
Rationale	To start the single player game.
Parameters Required	-
Usage Scenarios	1. User clicks on start game 2. System directs user to game page
Public Method 3	calcPoints(user, newPoints)
Rationale	To keep track of user score in the single player game. It would also act as an indicator if the user has passed the stage at the end of the game.
Parameters Required	● **user** which indicates the user to calculate points for ● **newPoints** which indicates points to be updated to the total score.
Usage Scenarios	1. User answers a question correctly 2. System will calculate & update the points accordingly

Subsystem Interface Name	Multi Mode Manager
Public Method 1	createRoom(hostId, roomName)
Rationale	Players need to be able to create rooms before inviting other players to

	play with them. Returns room ID.
Parameters Required	• **hostId** which tells the subsystem the id of the player to assign room ownership to • **roomName** for players to identify and differentiate between rooms
Usage Scenarios	1. User selects the multi player game mode 2. User selects the create room option 3. User enters the room name
Public Method 2	invitePlayer(friendId, roomId)
Rationale	Players need to be able to invite other players to rooms they have created.
Parameters Required	• **friendId** which tells the subsystem the id of the player to invite • **roomId** to identify which room the invitation is for
Usage Scenarios	1. User clicks on the invite player button 2. User enters the player ID of the player to be invited
Public Method 3	acceptInvite(accepterId, roomId)
Rationale	Players need to be able to accept invitations from other players who are hosting rooms.
Parameters Required	• **accepterId** which identifies the new player to be added into a room • **roomId** which identifies which room the new player should be added into
Usage Scenarios	1. An invitation request will pop up while the user is on the Main Menu Page 2. User selects the accept invite option
Public Method 4	startMultiGame(roomId)
Rationale	Room hosts need to be able to start the game once players have been added to the room.
Parameters Required	• **roomId** which tells the subsystem the id of the room to start the game in
Usage Scenarios	1. As host, user is done inviting friends into the room 2. User selects the start game option
Public Method 5	chooseRestaurant(selection, primaryLevel)
Rationale	The difficulty of questions differs depending on the restaurant that the user has selected
Parameters Required	• **selection** which indicates the user's selection • **primaryLevel** which indicates the user's current primary level. Users cannot select restaurants that are beyond his current level of

	studies
Usage Scenarios	1. User selects on the multi player game mode 2. System redirects users to the available restaurant page 3. User selects a restaurant 4. System redirects user to selected restaurant
Public Method 6	calcPoints(user, newPoints)
Rationale	To keep track of user score in the multi player game. It would also act as an indicator if the user has won the game first.
Parameters Required	● **user** which indicates the user to calculate points for ● **newPoints** which indicates points to be updated to the total score.
Usage Scenarios	1. User answers a question correctly 2. System will calculate & update the points accordingly

Subsystem Interface Name	Account Manager
Public Method 1	createAccount(emailAddress, password, primaryLevel)
Rationale	For security reasons. Only authorized users can play the game and only the user can access his or her account.
Parameters Required	● **emailAddress** an identification used by the user to access the game ● **password** A Base64 UTF-8 encoded password for the user ● **primaryLevel** which indicates the user's current primary level
Usage Scenarios	1. User clicks on sign up button 2. System directs user to sign up page 3. User enter his or her email address, password and primary school level 4. User clicks on sign up button 5. System will save user’s email and password into database 6. System will redirect user to login page
Public Method 2	loginAccount(emailAddress, password)
Rationale	For security reasons. Only authorized users can play the game and only the user can access his or her account.
Parameters Required	● **emailAddress** an identification used by the user to access the game ● **password** is a Base64 UTF-8 encoded password for the user
Usage Scenarios	1. User enters his or her email address and password

	2. User clicks on login button 3. System authenticate users' email address and password a. Successful: System will redirect user to home page b. Unsuccessful: System will display error message and prompt user to enter his or her email address and password again
Public Method 3	logoutAccount()
Rationale	To terminate current login session
Parameters Required	-
Usage Scenarios	1. User clicks logout button 2. System terminate user login session, clear cookies and cache 3. System redirect user to login page

Subsystem Interface Name	Shop Manager
Public Method 1	displayShop()
Rationale	Allows users to access the shop and view the available power ups and items.
Parameters Required	-
Usage Scenarios	1. From the main Menu Page, user clicks on "shop" button and will be directed to the Shop Page 2. The displayShop method will access the shop and display all the available items to the user on the Shop page.
Public Method 2	purchaseItem(user, item)
Rationale	Allows user to purchase items and updates shop and user's inventory
Parameters required	● **user**: An object containing all information about the user ● **item**: An object containing all information about the item
Usage Scenarios	1. From the Shop Page, the user selects the item he wants to purchase and clicks on the purchase button 2. The purchaseItem method will add the item to his inventory and will also update the shop to indicate the item has been sold.

Subsystem Interface Name	Inventory Manager
Public Method 1	displayInventory(user)
Rationale	Access the account database and return all the power-ups and skins associated with a particular account
Parameters Required	• **user**: An object containing all information about the user
Usage Scenarios	1. From the Main Menu Page, the user clicks on the "view inventory" button and will be directed to the Inventory Page, which triggers the displayInventory method to be called. 2. The displayInventory method will return all the power-ups and skins owned by the user and they will be displayed on the Inventory Page.
Public Method 2	changeCharacter(user, newCharacterId)
Rationale	This method allows the user to change to a new character.
Parameters Required	• **user**: An object containing all information about the user • **newCharacterId**: which identifies the character in the game, must be unique for each character
Usage Scenarios	1. From the Inventory Page, the user clicks on the "customize character" button and will be directed to the Customize Character Page. 2. On the Customize Character Page, the user can click on the new character, which will be set to the currentCharacter by the changeCharacter method.

Subsystem Interface Name	Leaderboard Manager
Public Method 1	displayLeaderboard(leaderboard)
Rationale	Assess the leaderboard database and displays the current leaderboard rankings
Parameters Required	• **leaderboard** which is a list of users who are currently in the season's leaderboard
Usage Scenarios	1. From the Main Menu page, the user clicks on the "Leaderboard" button and will be directed to the Leaderboard Page.
Public Method 2	displayRewards(rewards)

Rationale	This allows the users to view the rewards available for the current season
Parameters Required	• **rewards** which is a list of items to be given out as rewards for the current season
Usage Scenarios	1. The user accesses the leaderboard from the main menu 2. User clicks on 'Rewards' button 3. The leaderboard manager will call the displayRewards methods and display the items available as rewards for the current season.

Subsystem Interface Name	Messaging App Manager
Public Method 1	shareGameResult(currentGameResult)
Rationale	This allows the users to share the post-game report for their single-player or multiplayer games via supported messaging applications
Parameters Required	• **currentGameResult** which is the image of the results screen of the current game
Usage Scenarios	1. The user has completed a single-player or multiplayer game 2. User clicks on 'Share game results' 3. The Messaging App Manager requests the user to choose a messaging platform and channel to share the game report with 4. The user clicks on their desired platform and channel to share with 5. The game report will be shared on the desired platform and channel
Public Method 2	shareLeaderboard(currentLeaderboard)
Rationale	This allows the users to share the current leaderboard via support messaging applications
Parameters Required	• **currentLeaderboard** which is the image of the current Leaderboard
Usage Scenarios	1. The user accesses the leaderboard from the main menu 2. User clicks on 'Share leaderboard' button 3. The Messaging App Manager requests the user to choose a messaging platform and channel to share the leaderboard with 4. The user clicks on their desired platform and channel to share with 5. The leaderboard will be shared on the desired platform and channel

Subsystem Interface Name	Data Access Manager
Public Method 1	getUser(emailAddress, password)
Rationale	This method returns a User object once the account details has been verified
Parameters Required	• **emailAddress** an identification used by the user to access the game • **password** A Base64 UTF-8 encoded password for the user
Usage Scenarios	1. The user logs into the game. 2. The user login credentials are verified and this method will be invoked to retrieve the user details.
Public Method 2	getRestaurant(primaryLevel)
Rationale	This method returns a Restaurant object
Parameters Required	• **primaryLevel** indicates the user's current primary level
Usage Scenarios	1. From the Main Menu Page, the user clicks on the “Single Player Mode”. 2. This method will be called to display the correct picture and name of the Resturant object.
Public Method 3	getLeaderboard(seasonId)
Rationale	This method returns a list of User sorted according to their elo-rating.
Parameters Required	• **seasonId**: Unique identifier for each season of the game
Usage Scenarios	1. From the Main Menu Page, the user clicks on the “Leaderboard” button and will be directed to the Leaderboard page. 2. This method will be called to display all the usernames and their corresponding elo-ratings.
Public Method 4	getRewards(seasonId)
Rationale	This method returns a list of items that will be rewarded to players on achieving a certain ranking on the leaderboard.
Parameters Required	• **seasonId**: Unique identifier for each season of the game
Usage Scenarios	1. From the Leaderboard Page, the user clicks on the “View Rewards” button and a pop-up page will appear. 2. This method will be called to display all the rewards available.
Public Method 5	getShopItems()
Rationale	This method returns a list of items that can be purchased from the shop.

Parameters Required	-
Usage Scenarios	1. The user clicks on the “Shop” button from the “Main Menu Page” 2. This method is called to retrieve all the items available for purchase in the shop.
Public Method 6	getQuestions(primaryLevel)
Rationale	This method returns a list of questions (based on the level of the user) that will be used in the game selected by the user.
Parameters Required	● **primaryLevel** indicates the user's current primary level
Usage Scenarios	1. The user starts either a single player game or multiplayer game. 2. This method will be called to retrieve the questions required for the game.

5. Updated Class Diagram