Assignment Report

SFWRENG 2AA4 (2026W)

Assignment 1

Author(s)

Devan Sandhu	sandhd7@mcmaster.ca	sandhd7
Nadeem Mohamed	mohamn84@mcmaster.ca	mohamn84
Billy Wu	wu897@mcmaster.ca	wu897
Nikhil Ranjith	ranjin1@mcmaster.ca	ranjin1
Vivek Patel	patev124@mcmaster.ca	patev124

GitHub URL

https://github.com/sparksavior/2aa4-team1

February 13, 2026

1 Executive Summary

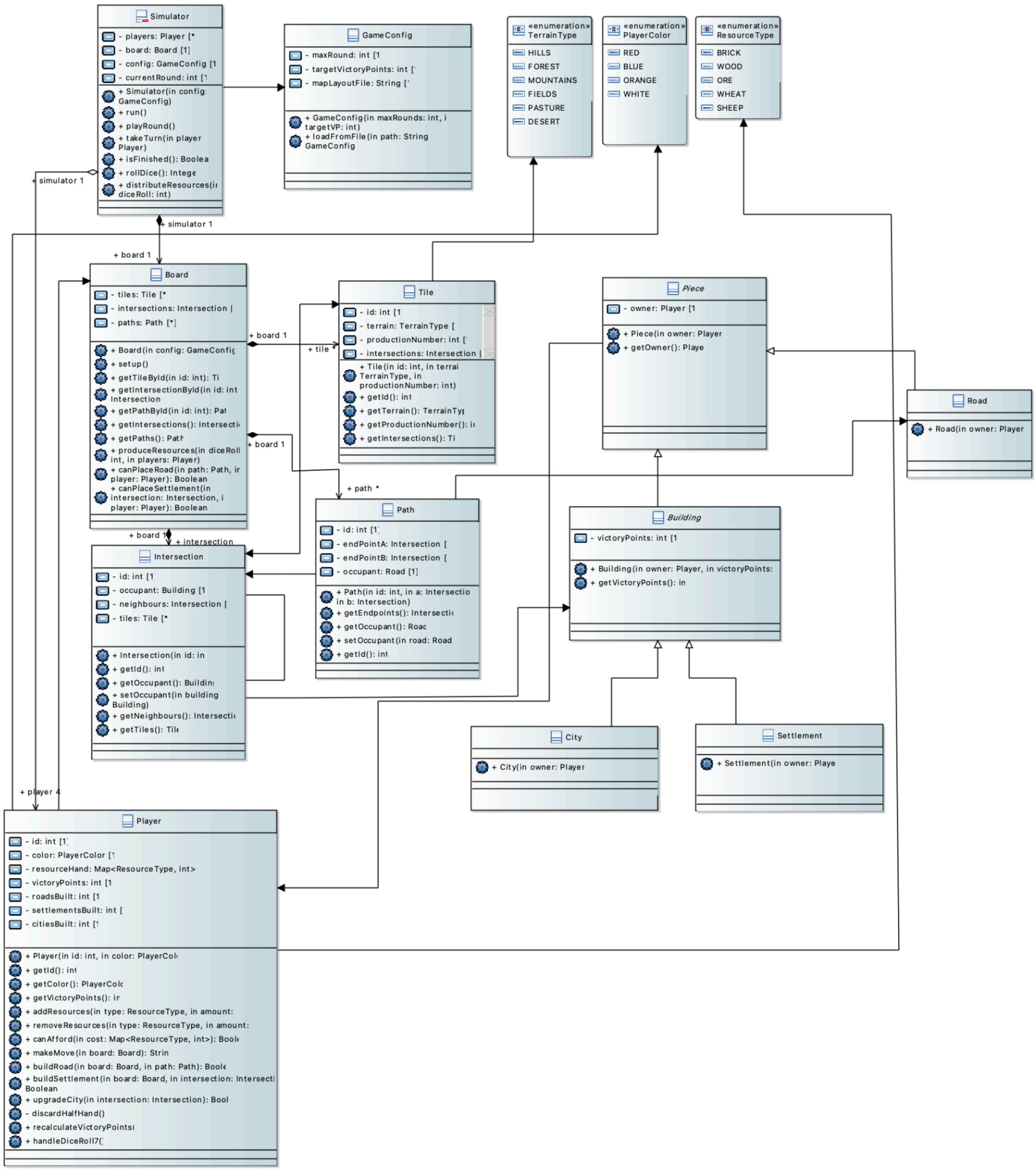
We designed the architecture and implemented the basic structure of the Settlers of Catan game simulator. We started with broad conceptual sketches in draw.io to discuss as a group. Throughout the design process we applied several software design principles from SOLID to GRASP, and STUPID. From those initial diagrams in UML via papyrus, the approach moved stepwise into actual code, turning our ideas into functional parts that meet the simulator requirements. Implementing object oriented concepts helped shape the structure, while SOLID guidelines helped keep components clear and independent. Game rules remained fixed throughout, serving as guardrails during development. Progress wasn't always linear as we were shifting back and forth between models and coding exposed mismatches that needed fixing. Despite these bumps along the way, the outcome matched expectations, delivering a complete, running system aligned with requirements. The UML modelling stage helped sort out roles and relationships among key classes and enums. Generating the model to code via Eclipse revealed weaknesses in automatic code creation along with those in generative AI tools. The end result was a working simulator that fulfilled its intended purpose, showing how structured development in the early stages proves its strength when compared to unplanned programming.

2 Requirements Traceability

Req ID	Status	Implemented in	Design considerations
R1.1 Map setup and identification scheme	Implemented	Board.java Board.setup(), setupTiles(), setupIntersections(), setupPaths()	The board and tile portions of the code are hard coded in the Board class. They are laid out in the proper structure with 0 for center, 1 to 6 for inner ring, 7 to 18 for outer ring. The intersection identifiers follow a sequence to assign them in a different order than the tiles.
R1.2 Four randomly acting agents	Implemented	Simulator.java, Player.java Simulator.createPlayers(), Player.makeMove()	The simulator class should create 4 players. The players actions are picked in a certain priority order from all the available actions at their turn.
R1.3 Simulation follows rulebook	Implemented	Simulator.java, Board.java, Player.java Simulator.playRound(), Board.produceResources( ), Player.handleDiceRoll7()	This includes the resource collection and main building mechanic. When a 7 gets rolled it should discard players cards (because of the robber).
R1.4 Simulation runs for configured rounds or until	Implemented	GameConfig.java, Simulator.java GameConfig.fromFile(),	The configuration file holds the num of rounds as per the rules. The simulator should end the program when it reaches the round limit or when a player gets 10

10 victory points		Simulator.isFinished()	victory points.
R1.5 Simulator halts upon reaching termination conditions	Implemented	Simulator.java Simulator.run()	The main loop in simulation should check if the game has met ending conditions after each turn and stop the game as soon as one condition is met.
R1.6 Key invariants respected	Implemented	Board.java, Player.java Board.canPlaceRoad(), Board.canPlaceSettlement(), Player.upgradeCity()	To place a road you should need to be connected to a structure of that players ownership. You can't have 2 adjacent intersections built on by one player. You can place a settlement without connecting to your roads.
R1.7 Console output format and victory point summary	Implemented	Simulator.java Simulator.takeTurn(), Simulator.playRound()	Each action is printed in the required format and victory points are displayed at the end of rounds.
R1.8 Players with more than seven cards attempt to build	Implemented	Player.java Player.makeMove()	If a player holds more than seven cards, the Player class will attempt to build or upgrade if they are able. However, actions are selected and not random from all possible actions.
R1.9 Demonstrator class with main method	Implemented	Demonstrator.java Demonstrator.main()	The Demonstrator class has a main method that initializes the configuration and executes a simulation run.

3 Design and Domain Modeling


Simulator
- players: Player [*
- board: Board [1]
- config: GameConfig [1
- currentRound: int [1
+ Simulator(in config: GameConfig)
+ run()
+ playRound()
+ takeTurn(in player Player)
+ isFinished(): Boolea
+ rollDice(): Intege
+ distributeResources(in diceRoll: int)
GameConfig
- maxRound: int [1
- targetVictoryPoints: int ['
- mapLayoutFile: String ['
+ GameConfig(in maxRounds: int, i targetVP: int)
+ loadFromFile(in path: String GameConfig
«enumeration» TerrainType
HILLS
FOREST
MOUNTAINS
FIELDS
PASTURE
DESERT
«enumeration» PlayerColor
RED
BLUE
ORANGE
WHITE
«enumeration» ResourceType
BRICK
WOOD
ORE
WHEAT
SHEEP
+ simulator 1
+ simulator 1
+ board 1
Board
- tiles: Tile [*
- intersections: Intersection |
- paths: Path [*]
+ Board(in config: GameConfig
+ setup()
+ getTileById(in id: int): Ti
+ getIntersectionById(in id: int Intersection
+ getPathById(in id: int): Pat
+ getIntersections(): Intersecti
+ getPaths(): Path
+ produceResources(in diceRoll int, in players: Player)
+ canPlaceRoad(in path: Path, ir player: Player): Boolean
+ canPlaceSettlement(in intersection: Intersection, i player: Player): Boolean
+ board 1
+ tile *
+ board 1
Tile
- id: int [1
- terrain: TerrainType [
- productionNumber: int ['
- intersections: Intersection |
+ Tile(in id: int, in terrai TerrainType, in productionNumber: int)
+ getId(): int
+ getTerrain(): TerrainTyp
+ getProductionNumber(): ir
+ getIntersections(): Ti
Piece
- owner: Player [1
+ Piece(in owner: Player
+ getOwner(): Playe
Road
+ Road(in owner: Player
+ path *
Path
- id: int [1
- endPointA: Intersection [
- endPointB: Intersection [
- occupant: Road [1]
+ Path(in id: int, in a: Intersectio in b: Intersection)
+ getEndpoints(): Intersectic
+ getOccupant(): Roac
+ setOccupant(in road: Road
+ getId(): int
Building
- victoryPoints: int [1
+ Building(in owner: Player, in victoryPoints:
+ getVictoryPoints(): in
+ board 1
+ intersection
Intersection
- id: int [1
- occupant: Building [1
- neighbours: Intersection [
- tiles: Tile [*
+ Intersection(in id: in
+ getId(): int
+ getOccupant(): Buildin
+ setOccupant(in building Building)
+ getNeighbours(): Intersectic
+ getTiles(): Tile
City
+ City(in owner: Player
Settlement
+ Settlement(in owner: Playe
+ player 4
Player
- id: int [1]
- color: PlayerColor [1
- resourceHand: Map<ResourceType, int>
- victoryPoints: int [1
- roadsBuilt: int [1
- settlementsBuilt: int [
- citiesBuilt: int [1
+ Player(in id: int, in color: PlayerColo
+ getId(): int
+ getColor(): PlayerColc
+ getVictoryPoints(): ir
+ addResources(in type: ResourceType, in amount:
+ removeResources(in type: ResourceType, in amount:
+ canAfford(in cost: Map<ResourceType, int>): Boole
+ makeMove(in board: Board): Strin
+ buildRoad(in board: Board, in path: Path): Boole
+ buildSettlement(in board: Board, in intersection: Intersecti Boolean
+ upgradeCity(in intersection: Intersection): Bool
- discardHalfHand()
+ recalculateVictoryPoints
+ handleDiceRoll7(
The UML Diagram for our project.

Please note the above UML image was taken directly from the papyrus screenshot tool and some text cutoff has occurred. This was the only best option to ensure a clear image in the report. If you would like a clearer image you can view it in our repository here or view our Papyrus project zip here.

Utility of the Domain Model vs. Specification

The domain model was useful because it made the ambiguous text from the assignment specs into a more structured, visual representation that is quickly comprehensible compared to reading a long page of text. The UML modelling stage of this assignment forced our team to make concrete decisions about cardinality/multiplicity, for example, 1..
* vs 0..
* and data types that the natural language specification left open to interpretation.

Natural Language vs. Conceptual Modelling

The natural language description in rulebook motivates conceptual modelling because human language is imprecise and prone to contradictions. This UML modelling step was needed to bridge the gap between Catan game rules and software logic. Some observations I noted:

Visualization of Coupling: The diagram reveals how tightly coupled the Board is to its components (Tile, Intersection), allowing us to comprehend the complexity of the logic before writing a single line of code.

Validation of Responsibilities: It allows us to quickly look at specific scenarios. For example, asking where the build logic is, reveals whether it belongs in Player or Board before we commit to an architecture/structure.

Early Error Detection: Before implementing the logic, the UML model can help iron out the design flaws and potential bottlenecks early to save time and potentially money in the future.

Unmodelled Aspects

Game State + Turn Flow: Class diagram shows what objects are there, but not when they interact or show the sequences of every turn such as roll dice, trade, building, etc.

- We can model this using a Sequence Diagram for the turn logic.

Complex Validity Rules: Currently, the diagram cannot enforce the rules such as a new road must connect to an existing road of the same color.

- This logic is best modelled using an Object Constraint Language, allowing to write specific mathematical rules like invariants that the code must obey, right on the diagram.

OO Mechanisms in Design

Object Oriented mechanisms were critical in organizing the complexity of Catan:

UML Reflection

We used an inheritance hierarchy like for example, from superclass Piece extending to subclass Building
and extending to subclass Settlement & City to reduce code duplication. We have also shared attributes
like owners are defined once in the parent Piece class, rather than repeated in subclasses.
We have also utilized polymorphism heavily. Allows the board to treat different objects the same. For
example, Intersection hold Building references without needing to know if it's specifically a City or a
Settlement until victory points are calculated. Encapsulation was also utilized in the design. We made all class attributes private and provided public accessor methods instead of exposing the attribute information that can impact the integrity of the game. We protect internal states of the player from being corrupted by other classes like Board.

SOLID Principles

We applied SOLID principles to make our simulator design robust. We changed Player.makeMove() to return a String instead of printing to the console. This ensures the Player class only handles game logic (such as deciding the move), while leaving the responsibility of printing/logging entirely to the Simulator. Have used the open close principle. Piece and Building inheritance hierarchy allows the system to be extended (for example adding a ship or metropolis in an expansion) by creating new subclasses without modifying the existing core logic in Board or Intersection. Also the Liskov Substitution Principle was applied. Our design ensures that a City can be used anywhere a Building is expected (like on an Intersection), ensuring that upgrading a piece does not break the board's storage structure.

4 Translating Engineering Models to Program Code

With the help of papyrus software, classes in UML turn into Java classes quickly, saving us time to create the structure implementation. Method signatures form where operations once stood. Where inheritance appears, extends follows, line by line. Enums move across untouched, fitting neatly into Java's enum framework. The core shape of the model holds, undistorted for the most part.

Partial translation of associations was observed in code generation. Arrays were successfully formed from multiplicities like Tile[] or Intersection[]. Despite the correctness, Java benefits more from flexible collections such as List for example. The preserved shape remains useful, yet needs some adjustments before real world application fits perfectly.

Errors appeared throughout the process of creating output. Instead of forming correct Java constructors, the system produced them as void methods, violating basic rules of the language and needing human fixes afterward. Inside certain classes, there were repeats of internal class definitions, for example more than one Board within a single container, even though these duplicates did not exist in the original design diagram and served no purpose. Primitive types from the modeling layer, including EInt and EString, made their way directly into the resulting source files, forcing developers to substitute standard equivalents like int or String manually typed. All this suggests translation from models to working code does not always preserve accurate programming structure, demanding careful inspection after conversion.

Though method signatures formed properly, their bodies stayed blank. This makes sense because behavior cannot come only from structural UML. Moreover, the expected correct inheritance translation occurred, keeping superclass and subclass links exactly as set in the original model.

The main advantage of code generations are keeping the structure aligned across models and real world build. When design matches code closely, through enforced sync, gaps in architecture become less likely. Still, evidence shows output from generators often needs reworking before deployment. For big systems, or those where failure risks are high, this method makes sense, due to its stability. On smaller efforts, though, skipping the extra step of modeling might save time overall.

5 Using Generative AI

The generative AI (LLM) produced Java classes that were perfect in structure and clearly laid out, matching key basic parts of the UML design that we intended to implement. We observed that the AI generated code retained the main components from Board, Simulator, Player, and Tile classes. It kept the main domains distinct and properly defined. What stood out was how inheritance appeared in logical form, especially for a complex UML model. As an example, the Piece class was shown as an abstract class which was what we intended, and Building and Road branched from it to extend. Moreover, this was observed with the Settlement class, following correct hierarchy rules as expected. Unlike what often happens with Papyrus, which tends to nest classes unnecessarily.

Still, generative AI (LLM's) struggled to maintain structural meaning. Where our papyrus UML showed 2 way links, the outputs often turned them into 1 directional or left them out entirely. Take Tile and Intersection for example, their connection appeared incomplete, weakening the overall framework. This suggests generative AI may copy surface layouts but fails to consistently uphold underlying relationships.

When building big systems, generative AI can help draft initial skeleton code structures (not the relationships). The pure design process is still something only humans can master. Because models make assumptions, every generated class needs checking alongside specifications and intent. This is a common downside of using generative AI to generate code, which is detailed unit testing of generated classes. Without careful review, errors might slip into foundational layers. All in all using design models in software development works best for large systems.

Looking at the running company reflection and the churn rate, we would select Model driver SE as the primary strategy because it requires 30 total days, where 25 for design and 5 for coding. The impact of churn rate heavily outweighs the cost of development time. Losing 10 clients every day is costing about $10000 in daily revenue. Choosing model driven means we need to accept the cost delay to secure a lower 2.5% churn rate but we could preserve the $225 million in net revenue compared to other approaches.

What distinguishes coding from engineering is that coding aims for faster results, and engineering values accuracy, high standards, and lasting performance.

Our Generative AI Prompt:
"Act as a Senior Java Developer. Convert the following UML class diagram into Java code. Ensure all attributes use private visibility and include getter/setter methods. Ensure all association relationships, inheritance are translated into Java and multiplicities are considered."

Hyperparameters: Default (Balanced)

6 Implementation

It took more than one try to get the domain model to the level of our design expectations. During the shift from design to implementation, several rounds of revision happened, which is typical in this kind of development work. As an example, we ended up swapping String IDs in the UML diagrams for int IDs when implementing the model. The idea behind identifiers stayed abstract at the modelling level. However, in practice, working with list indices proved faster than using HashMaps. These observations we noticed only came once the implementation in Java began, highlighting how thinking about structure often differs from handling execution details. In a later version, we tackled 2 way links between elements. Even though the design called for mutual connections between Tiles and Intersections, early code kept data in just one direction. Testing revealed that knowing nearby tiles from an intersection improved how resources were generated, leading us to add backward references while maintaining alignment across both sides.

These situations show how modelling stays removed from concrete operations, whereas building it reveals the nitty gritty aspects of actual software deployment. Because of this gap, going back and forth isn't a flaw in design thinking, it's simply what happens when intricate systems like a game simulator with multiple layers get made.

Through each cycle, a few hurdles showed up. Some were tied to code, others to thinking about the structure. To keep 2 way connections stable, safeguards had to be built right into the logic, stopping repeated or mismatched links before they formed. Speed issues, like how quickly elements could find needed resources, only became clear once testing began. From a design standpoint, walking the line between following SOLID rules strictly and allowing a practical, workable UML layout turned out to weigh heaviest. We faced a tradeoff between practical design and strict software design principles. Applying the SOLID principles correctly would have needed more classes and interfaces (making UML more complex), but we chose to prioritize a clear and simple design that relaxed the SRP and DIP principles.

Future iterations would benefit from round trip engineering tools to automatically sync code changed back to UML models and object constraint language parsers to validate game logic rules before compiling.

Using familiar object oriented concepts from the design phase, inheritance links classes like Settlement and City to their parent Building. Through abstract types, different objects behave uniquely at runtime, this supports flexible interactions across the system, low coupling and maintainability. The data stays protected within each class, exposed only via specific getters and setters methods that manage how changes occur (even though this can cause leaky abstraction). Rather than standalone parts, elements such as tiles, paths, and intersections come together under a single Board structure. Where appropriate, subclasses fit seamlessly into parent type expectations, meeting Liskov requirements set by rule R1.6.

Still, adherence to every SOLID guideline wasn't strict. Some components handle more than one role, which is not ideal in theory, yet helpful here for clarity and fitting project limits. What stands out here stems from intentional choices, not confusion. Looking closer reveals a central idea, which is chasing flawless design structure can miss the point. Real world solutions demand weighing clarity against speed, upkeep, how tasks are divided, and what each part must do.

7 Reflection on the Engineering Process

Our team was a group of 5 and we put in about 10 to 20 hours of time dedicated to design and implement the simulator. More time was spent on the design stage and UML modelling as that was the main task of this assignment. The work was split so all members took one main task and every member shared their reflection points pertaining to their task in the repo/kanban board. Although we all had 1 major task, we all worked together and contributed to each other's parts when the need arose. Billy tackled setting up the repo, managing tasks with Kanban boards, plus handling docs structure. Nikhil worked on UML diagrams, refining them step by step and implemented the model in Papyrus. Nadeem generated code and studied how it translated. Vivek explored generative AI outputs, noting patterns found along the way. Billy assisted with the implementation and fine details such as maps, config reading, rules that stay fixed, agent behaviors, logs, and added notes to the Demonstrator class. These were just the general tasks, we all jumped in to assist each other out where needed. Sections came from a common outline that everyone owned their part with proof tracked via Kanban project as we went.

Most of our communications happened in person, though we also used Microsoft Teams when needed outside lecture/tutorial time. Our work flowed through GitHub, where we tracked changes, managed tasks, and followed versions closely. Progress took shape on a Kanban board, making it easier to see who did what and how far things had moved. What slowed things down most was underestimating implementation complexity. Though the first steps seemed clear and under control, halfway through, new layers showed up, connections and checks that no one expected when drawing UML diagrams. Starting off with a solid plan helped us stay ahead of deadlines. From the time we started this assignment, having predefined dates kept things moving smoothly instead of piling up later. The tasks were spread across the team fairly, meaning no one got overloaded along the way. Tracking each step carefully via Kanban thru GitHub made it easier to adjust tasks and next steps when needed. What stood out most was how small updates throughout improved results and brought value. Clear roles gave everyone direction without confusion. Planning like this turned complex work into manageable pieces. The whole experience showed that steady organization beats last minute rushes every time.

8 Roles and Responsibilities

The team members contributed equally to the deliverable.