ChatGPT

Architecture Plan for a Multiplayer Poker Game (Godot 4.5 + Zig Backend)

Overview

This project will implement a custom multiplayer poker-like card game with a client-server architecture. The **client** is built in **Godot 4.5** (for rich visuals and cross-platform support), and the **server** is implemented in **Zig** for high-performance game logic. The client and server communicate via **WebSockets** using a **JSON-RPC 2.0** protocol (i.e. JSON-formatted remote procedure calls) for structured messaging. Key features include a variable **4-8 player** game mode, a **lobby system** for players to join/create games, an **in-game chat**, and a special **"tool" system** that allows players to buy/use items (e.g. swap positions) during play. An **AI bot** (trained via reinforcement learning) will be integrated as a computer-controlled player, with multiple difficulty levels. The following sections detail the plan for each component, ensuring a cohesive and scalable architecture.

Game Flow and Core Mechanics

The core gameplay is a **custom poker-like card game** where players take turns playing cards one by one until someone runs out of cards. All players are dealt a hand of cards at the start. Play proceeds in rounds with each player playing a valid card on their turn; the round continues **"until there is no card in hand"** – meaning at least one player empties their hand, effectively ending the round with that player as the winner (alternatively, play can continue until all cards are played to rank all players). The design will support **4 to 8 players** per game, which means the deck size might be adjusted as needed (e.g. use multiple standard decks shuffled together for 7-8 players so that card distribution is sufficient). The turn order typically proceeds clockwise (or in a defined sequence of player positions), and the server will track whose turn it is and enforce that only the current player can play a card at any given time. If a player tries an invalid action (e.g. playing out of turn or playing a card not in their hand), the server will reject it and send an error response. A typical game flow will look like:

1. **Game Setup:** One player creates a game (becoming the host) and optionally sets the desired number of players (4–8). The server initializes a new game instance, shuffles the deck(s), and awaits players. Other players join the game via the lobby. Once the minimum number of players (4) have joined (or the host manually starts the game when ready), the server deals cards to each player and notifies clients that the game has started (each client receives their own hand of cards privately).
2. **Turn-by-Turn Play:** The server designates the starting player (e.g. the host or a random player) and sends a message indicating whose turn it is. Players take turns in order. On a player's turn, they can perform an action such as *play a card* (standard move) or *use a tool* (special action, described below). The client sends the chosen action via JSON-RPC to the server (e.g. `playCard` or `useTool` method). The server validates the action against game rules (e.g. the card exists in that player's hand, and it fits any game rule constraints) and then updates the game state. It then broadcasts a notification to all players about the action (e.g. "Player X played [Card]" or "Player X used [Tool]") and updates whose turn is next.
3. **Round End:** When a player empties their hand, the round (or game) ends. The server determines the outcome – e.g. the first player to run out of cards is the winner of the game. (If the game

design calls for playing until all cards are gone to rank players, the server can continue the cycle until only one player is left with cards.) The server sends a game-over notification with the results (winner and any rankings) to all clients. The players return to the lobby or can choose to start a new game.

Throughout this flow, the **server is authoritative** over game state to prevent cheating – it handles card shuffling, deals, turn order, and validates every move. The client is primarily for rendering and user input, and it trusts the server for the game logic. This clear separation ensures consistency and fairness across the browser and mobile clients.

Tool System (In-Game Items)

A unique feature of this game is the **tool system**, which allows players to buy and use special tools during gameplay to gain advantages or alter the game's state. For example, one such tool might allow a player to **switch positions** with another player. The server will incorporate this as part of the game rules:

- **Tool Availability:** At the start of each game (or in the lobby), players can access a list of available tools (e.g. "Swap Position", and potentially others like "Skip your next turn", "Peek at one card", etc., depending on game design). Since the user system is lightweight and without persistent accounts, tool purchases can be handled in a simple way – for instance, each player could be given a certain number of coins or tokens at the start of a game or earned through previous rounds. They can spend these tokens to activate a tool. No complex e-commerce or real currency is needed; it's an in-game mechanic. The **client UI** will show a tool menu/button when it's the player's turn (or at appropriate times) so they can choose to use a tool instead of playing a card.

- **Using a Tool:** When a player decides to use a tool, the client sends a JSON-RPC request like `useTool`, with parameters identifying the tool type and any target (e.g. target player to swap with). The **server validates** whether the player *has the tool and can afford it* (for example, checks the player's token balance or inventory) and whether using it is legal at that moment (some tools might only be usable on one's turn, while others could be allowed off-turn if designed so). If valid, the server applies the tool's effect to the game state. For the "switch positions" tool, the server would swap the positions of the two players in the turn order (and possibly swap their seating order for all clients' UI). This could mean if Player A uses "Swap Position" on Player B, their turn order is exchanged – so Player A takes Player B's spot in the rotation and vice versa. The server then broadcasts a notification so all clients update the display (showing that players have swapped seats) and adjust turn sequencing accordingly. No actual cards are exchanged – only their seating/turn order – which can be an interesting strategic reset of the advantage of turn position.

- **Tool Implementation:** The server's game logic module will include a **Tool Manager** or similar component that defines each tool's effect as a function that modifies the game state. For extensibility, tools can be represented as data objects (with properties like cost, name, and effect function). When `useTool` is called, the server looks up the tool, checks cost and availability, deducts the cost, and invokes the effect function on the current game state. After applying the effect, the server sends appropriate updates to players (e.g. in the swap case, a message like "Player A swapped positions with Player B"). The design will ensure that using a tool consumes the player's turn (unless specified otherwise by game rules), so after using a tool, it likely becomes the next player's turn (unless the tool defines something else).

This system introduces an extra layer of strategy. From an architecture perspective, it means the game state isn't just cards and turns, but also includes each player's **inventory/currency** and game parameters that can change (player order, etc.). All of this will be maintained on the server side to ensure fairness. The client will provide the interface for players to trigger these tools, and the server will enforce and synchronize the effects.

Lobby and Chat Systems

Beyond the core gameplay, the platform will include a **lobby** where players can meet, create games, join games, and chat. The lobby is essentially a waiting area and game browser:

- **Lightweight User System:** Because a **very light user system** is required (no formal authentication or passwords), players will join as guests. When a client opens the game (application or webpage), they can be prompted to enter a **display name** (nickname) or one can be assigned randomly. This name is only used for identification in the lobby and games (no login needed). The server can generate a simple unique ID for each connection (to track the player internally) and associate it with the chosen nickname. Since there's no persistent account, once a player disconnects, their identity isn't saved beyond maybe a temporary session ID. This keeps things simple and frictionless for users.

- **Lobby Interface:** The **Godot client** will have a Lobby screen that lists available game rooms. The server maintains a list of open games (those waiting for players or in progress). For each game room, info like current players count (e.g. "Room 5: 3/6 players") can be displayed. Players have the option to **create a new game** (specifying the max players up to 8, if desired) or **join** an existing one from the list. If no automatic matchmaking is needed, players manually select a room. The server provides this list on request or via update broadcasts when new rooms are created. Since no tournament or ranked matchmaking is required, this is straightforward: essentially a list of active game sessions the user can pick from.

- **Game Room Management:** When a player creates a game, the server generates a new game instance with a unique ID (and possibly allows the host to set parameters like number of players, private/public, etc. – though if not needed, all games can be public by default). Joining a game will involve the client sending a `joinGame` request (with the game ID) to the server. The server adds that player's connection to the game's player list and notifies the other players in that room that a new player joined (so the UI can update the player list in the room). In the lobby UI, once a player joins a game, they transition to the **game scene** (even if the game hasn't started yet, they'll be in a waiting room state). If the game requires a minimum of 4 players, the server will wait until at least 4 have joined before allowing a start. Optionally, if after some time not enough human players join, the host or server could fill remaining slots with AI bots (this ensures games can start even if fewer humans are available – this is especially useful for testing and for players who want to practice).

- **Chat System:** There will be two scopes of chat: **lobby chat** and **in-game chat**. The **lobby chat** allows all connected players (or those in a particular lobby channel) to communicate globally – for example, to socialize or ask for others to join their game. The **in-game chat** is limited to the players within a single game room (once in a game, the chat messages should only go to those participants). This prevents game-specific talk from flooding the global chat. Technically, both can be handled similarly by the server: each chat message is a JSON-RPC notification (or request) sent to the server which then broadcasts it to the appropriate recipients. For instance, a client might call `sendChat` with a message and the server tags it with the sender's name and relays it to either all lobby users or all game-room users. On the Godot client side, there will be a chat UI panel in both the lobby scene and the game scene. Messages will appear with the sender's name. Because the user system has no login, impersonation could be an issue (anyone could choose the same nickname), but for simplicity we can accept this risk in a casual game. If needed, the server could append an internal user ID to messages to reliably distinguish players, but it won't be exposed.

- **Implementation Details:** The chat is implemented over the same WebSocket connection (no separate connection needed). It might use a specific JSON-RPC method or notification (e.g. `chatMessage`) with parameters `{text:"...", from:"PlayerName"}`. The server doesn't store chat history persistently (unless desired); it just relays messages. This keeps things lightweight. The lobby and room system will be managed by a **Lobby Manager** on the server, which keeps track of active games, waiting players, etc. With no tournaments or ranking, the structure remains simple.

In summary, the lobby provides the **social hub** and game management, and the chat system fosters communication both globally and within games. These systems ensure players can organize matches without automated matchmaking, fitting the requirements of no matchmaking needed.

Frontend: Godot 4.5 Client (Browser & Mobile)

The client will be built with Godot 4.5, leveraging its scene system and cross-platform capabilities to target **web browsers** as well as **mobile devices (tablets/phones)**. Godot's strength in 2D/GUI and ability to export to multiple platforms makes it ideal for this project. Key aspects of the frontend design include:

- **User Interface & Scenes:** The UI will be organized into multiple scenes for clarity. We will have a **Lobby Scene** (showing the game list, global chat, and options to create/join games) and a **Game Scene** (showing the actual card table, players, hands, and in-game chat). Godot's UI nodes (Control nodes like Panel, Label, Button, etc.) will be used to create an interface that is intuitive and scales to different screen sizes. For example, in the Game scene, we'll display each player around a virtual table (with their name and number of cards, perhaps an avatar), and the local player's hand of cards at the bottom. The tool system can be shown as small icons or buttons that light up when usable. The chat window can be a scrollable text area on one side with an input field. Care will be taken to keep the UI responsive – **Godot supports anchoring and scaling** for different resolutions, which is important for mobile vs desktop browser. We will design the UI with a flexible layout so it works on widescreen (desktop) and portrait (mobile) by anchoring UI elements and possibly switching to a vertical layout on narrow screens.

- **Input and Controls:** On desktop (browser), players will use mouse clicks to select cards and buttons, whereas on mobile, touch input will be used. Godot abstracts input events so we can handle both. For example, tapping a card in your hand could highlight it, and a "Play" button plays the selected card. Drag-and-drop could be implemented (drag card to table to play), but simple tap + button might be more reliable for touch. We will ensure that the interactive elements (cards, tool buttons, chat input) are sized appropriately for touch on mobile (larger buttons). Testing on various screen sizes will refine this.

- **Networking Integration:** Godot 4.5 provides a WebSocket client API, which we will use to connect to the Zig server. **Godot supports WebSocket in both native and HTML5 exports** [1], meaning the same code can run in a browser (via WebAssembly + WebGL) or in a mobile app. We will create a singleton node (autoload) or a dedicated `NetworkManager` node in Godot that manages the WebSocket connection. For a web build, connecting to a `wss://` (secure WebSocket) endpoint is possible if the server is hosted with SSL (or `ws://` for local testing). Once connected, the `NetworkManager` will listen for incoming messages and dispatch them to the appropriate handlers in the game. Since we're using JSON-RPC, incoming data will be JSON strings. Godot's GDScript (or C# if using it) can easily parse JSON (e.g. using `JSON.parse()` to get a dictionary). For example, when a message `{ "method": "gameStarted", "params": {...} }` arrives, the client code will detect the `method` and call the corresponding function to handle it (e.g. setup the game UI). Outgoing actions (like play card, chat, etc.) will be formatted as JSON and sent through the WebSocket. We won't use Godot's high-level MultiplayerAPI for RPC in this case, because our server is custom; instead, we handle the networking manually with WebSockets and JSON.

- **Rendering & Feedback:** Godot will handle all rendering of cards and animations. This means when the server broadcasts a game state change (e.g. a card played), the client will animate that card moving from the player's hand to the center pile or to a discard area. Similarly, when the "swap positions" tool is used, the client might animate the two players swapping their positions on the screen. Godot's animation system or tweening can be used for these polish elements. Real-time updates via WebSocket ensure that all clients see the state changes synchronously. Latency requirements are moderate – card games are turn-based, so a few hundred milliseconds delay is acceptable; still, we aim for efficient messaging.

- **Cross-Platform Deployment:** Using Godot means we can export **to Web (HTML5)** and to **Android/iOS** with largely the same code. For web, Godot will export the project as a WebAssembly application that runs in-browser. We must ensure to enable the proper export templates and that the **WebSocket** functionality is working in HTML5 (Godot uses WebSocketPeer which is compatible with browsers). According to the Godot documentation, the WebSocket implementation works in HTML5 by using browser APIs under the hood [1]. We will need to host the game files on a web server along with enabling Cross-Origin for the WebSocket server if they are on different domains. For mobile, we can export APK for Android and an Xcode project or IPA for iOS. On mobile, optionally, we could allow the game to connect to the same backend server, enabling cross-play between web and mobile players in the same games. This is feasible since the protocol is the same. We will just need to handle the slight differences (for instance, iOS might require `wss://` secure sockets if not on localhost).

- **Testing and UI polish:** We will test the Godot client on different platforms to ensure consistency. Godot 4.5's rendering (with Vulkan) might need fallback to OpenGL ES for web; the Godot export templates typically handle that. We should also consider using **Godot's debugging tools** to simulate different network conditions and ensure the client doesn't freeze if the connection is lost (e.g. handle a disconnect event by returning to lobby and showing a message).

Overall, the Godot frontend will provide a rich, interactive experience, while delegating game logic to the server. Its cross-platform nature addresses the requirement to target browsers and mobile devices with one codebase.

Backend: Zig Server and Game Logic

The backend will be a dedicated server application written in **Zig**, chosen for its performance, safety, and low-level control which are beneficial for a real-time multiplayer server. The server's responsibilities include maintaining game state, processing client actions, enforcing rules, and sending updates to clients. Here's the plan for the server architecture:

- **Server Structure:** The server will run as a **WebSocket server** listening on a specified port (e.g. 8080). It will handle multiple client connections concurrently. Using Zig, we have a couple of options for concurrency: Zig's standard library offers non-blocking I/O and we can use an event loop, or we can spawn threads to handle different tasks. A likely approach is to use an async event loop with non-blocking sockets to manage many connections in one process, since Zig excels at handling asynchronous workflows without heavy threads. We can utilize an existing Zig WebSocket library to simplify development – for example, *karlseguin's* `websocket.zig` library can handle the WebSocket protocol framing and provide callback hooks for incoming messages. This library also ensures that only one message per connection is processed at a time (preventing race conditions in a single client's handler) [2], which is useful for thread safety. The server will maintain a list of active connections (each representing a player session).

- **Game and Player Management:** We will create data structures in Zig to represent the game state. A **Game** struct will hold all information about a single match: the list of players (each with a hand of cards and related info), the deck(s) of cards (probably represented as an array or stack of card identifiers), the current turn index, and any ongoing play info (like the last card played, if relevant to game rules). A **Player** struct will hold details for each player: their connection (to send messages), their name, their current hand (list of cards), their remaining tool tokens or inventory, and maybe a flag if they are an AI bot. We will also have a **Lobby** or **GameManager** struct globally that holds all active games (e.g. a dictionary mapping game IDs to Game instances) and possibly a list of players who are not in a game (idle in the lobby). When players connect, they initially exist in the lobby context; when they join a game, the server moves them into a game instance.

- **JSON-RPC Handling:** We'll use **JSON-RPC 2.0** as the message protocol on top of WebSockets. This means each incoming message from a client will be a JSON object typically containing keys like `"jsonrpc": "2.0"`, a `"method"` name, a `"params"` object/array, and possibly an `"id"` if it's a request expecting a response. Zig can parse JSON strings – we might use a JSON library or even better, use a dedicated JSON-RPC library. **ZigJR** is one such library that provides a full implementation of JSON-RPC 2.0 for Zig, including parsing, composing messages, and dispatching to handler functions [3]. Adopting ZigJR could accelerate development: we can register handler functions for methods like `joinGame`, `playCard`, etc., and ZigJR will call them with the parsed parameters. It supports notifications (calls without response) and error handling out of the box [4]. If we use ZigJR, our code will focus on implementing the logic in those handler functions. Otherwise, we will manually parse the JSON (Zig's std.json can parse into a DOM or we can use a SAX style parser since our protocol is known) and then route the call. Each method from the client will correspond to a server function.

- **Key JSON-RPC Methods:** We will define a set of RPC methods that clients can call (requests), and some notifications that server will send:

- `setName` **(request):** Called right after connecting, to set the player's display name in the server's records (since no login, this just ties a nickname to the connection). The server stores the name and returns a confirmation.
- `createGame` **(request):** Creates a new game room. Params might include desired max players (default 4 or another number up to 8). The server creates a Game struct, assigns an ID, marks

the requesting player as the host and first member, and returns the game ID. The player is now in that game's lobby (state = waiting for others).

- `joinGame` **(request):** Player requests to join an existing room (by ID). The server checks if the game exists and is not full or already started. If ok, it adds the player to that Game's player list. It then notifies existing players in that game that a new player joined (via a `playerJoined` notification), possibly including the new player's name. It returns success or failure (e.g. if game not found or full).
- `startGame` **(request):** Possibly used by the host to manually start the game (if we allow the host to begin before the room is full, e.g. start with 5 out of 8 if they don't want to wait). The server will only honor this if the current player count >= minimum (4). On start, the server initializes the deck, shuffles, deals cards to each player (distributing card values to each player's data). Then it sends out a `gameStarted` **notification** to all players in the room. This message might contain some public info like the turn order (whose first, maybe a list of player names in playing order) and the initial hand for the **receiving** player (the server can send each player their own cards in a separate message, e.g. `yourCards` notification with an array of card identifiers). Card identifiers can be simple (like "AS" for Ace of Spades, etc.) or numeric codes – the client will have a mapping to actual card images.
- `playCard` **(request):** Sent by a client when playing a card from their hand. Params: card identifier (and possibly additional info if needed by rules). The server checks that it's actually that player's turn and the card is in their hand. It also checks rule validity (for example, if there was a rule like "must play higher card than previous" – but if it's a simple discard game, maybe no such rule). Assuming valid, the server updates the Game state: removes the card from that player's hand, updates any central piles or last played card state, and maybe records that player's last action. Then the server sends a `cardPlayed` **notification** to all players, containing which player played what card. (If some information should be hidden, e.g. in certain games you might not reveal the card to all, but in most shedding games the played card is visible to everyone). The server then checks win condition – if that play made the player's hand empty, broadcast a `playerWon` (or `roundEnded`) event. Otherwise, advance the turn to the next player and send a `turn` notification indicating it's now Player Y's turn.
- `useTool` **(request):** Sent by a client to use a special tool (as described in the tool system). Params might include the tool type (e.g. "swap") and a target (e.g. target player's ID or name to swap with). The server verifies availability and then executes the tool's effect (using the Tool subsystem). After effect, send appropriate notifications: e.g. `toolUsed` to all, describing what happened (could contain e.g. "Player X used Swap with Player Y"). If the tool affects turn order or other state, all clients will receive the updated state (perhaps via the same `toolUsed` payload or separate messages like an updated player order). Then proceed with turn updates (in the swap example, turn might instantly pass to next in the *new* order).
- `sendChat` **(request/notification):** For chat messages. Params: the text content. The server, upon receiving, determines the context (if the player is currently not in a game, it's a lobby chat; if in a game, it's game chat). Then it broadcasts a `chatMessage` **notification** to either all lobby connections or all players in that game. The notification will include the sender's name and the text. (This can be a notification with no response needed).

- **Server Notifications:** In addition to the above, the server will issue notifications for state changes: `playerJoined` (with new player info), `playerLeft` (if someone disconnects mid-game or leaves), `gameStarted`, `cardPlayed`, `toolUsed`, `turn` (whose turn it is now), `gameEnded` (with results). These are unidirectional messages from server to client, so in JSON-RPC they would be sent as notifications (no `id` field) or as requests that the client is expected not to reply to. JSON-RPC 2.0 supports this pattern for push updates [4]. Using a library like ZigJR can help manage sending these from the server to all relevant clients easily.

- **Game Logic Enforcement:** The server will contain the logic for the card game rules. This includes shuffling algorithm, dealing, checking legal moves, and determining round winners. Since this is a custom game, we will implement these rules in plain Zig code. It's advisable to isolate the game logic in its own module (for example, a `GameLogic.zig` that has functions like `start_game(game: *Game)`, `play_card(game: *Game, player: *Player, card: Card) -> Error?`, `use_tool(game: *Game, player: *Player, tool: Tool) -> Error?`). This way the rules are clear and testable. We can even write unit tests for these using Zig's testing framework to ensure correctness. The server will call these functions when corresponding RPC handlers are invoked. By separating the logic, we could also reuse it for AI training (more on that later).

- **Handling Multiple Games:** The server must handle potentially several games concurrently (though each player connection is only in one game at a time). The data for each game is separate. The lobby will route incoming `joinGame`/`createGame` requests to the Lobby Manager which creates or assigns a game. Once a player is in a game, their messages like `playCard` will need to be routed to the correct Game instance. This can be done by storing a reference or ID of the game in the player's session data. For example, when a connection is accepted, we create a `PlayerSession` object with fields `{ id, name, currentGameID?, connection, ... }`. When they join a game, we set `currentGameID`. Then in the message handler, we check `player.currentGameID` to know which game to apply the action to. This way, if two games are happening, actions from players in game A only affect game A. Broadcasting messages will also use the game's player list. We will implement utility functions for broadcasting within a game (to loop through players and send them a JSON notification via their WebSocket). Zig's concurrency will allow simultaneous games; we just have to guard shared data if using threads (e.g. if two threads could access the game list, use a mutex or run game logic on a single thread). Since WebSocket events per connection are single-threaded in the mentioned library, we might handle each game's logic on one thread as well, or funnel all game events through a single thread via message queues. Simpler yet is to use one event loop (single-threaded) that sequentially handles messages – given a card game's pace, this can be sufficient for dozens of games and players. If scaling up, we could allocate different worker threads per game or group of games. The configuration of the WebSocket library allows setting number of worker threads (listener threads) [5], which we can use to distribute load.

- **Error Handling and Validation:** The server will gracefully handle invalid requests. Using JSON-RPC, we can return standardized errors. For example, if a player sends `playCard` when it's not their turn, the server can respond with an error object (`error: { code: ..., message: "Not your turn" }`). The client can display this to the user or handle it silently (e.g. by ignoring any out-of-turn UI action). Likewise, if `joinGame` fails, return an error (game full or not found). ZigJR or our own implementation will allow sending such JSON-RPC error responses easily. By adhering to JSON-RPC 2.0, we ensure the protocol is clear and debuggable (JSON is human-readable, which helps during development).

- **Persistence:** Since the user system is minimal and there's no need for long-term stats or account data, the server can be largely stateless between sessions. We don't need a database for users. We might temporarily store some data in memory, like if we wanted to implement a simple leaderboard of number of games won in the current run, but that resets on server restart. The focus is on real-time state in RAM. This keeps the backend simple.

In essence, the Zig backend will act as a real-time authoritative game server. Its architecture uses **WebSockets for communication**, **JSON-RPC for structured messaging**, and **pure Zig data structures** for game state. We leverage libraries where useful (WebSocket handling, JSON parsing) to avoid reinventing low-level protocols, allowing us to concentrate on game-specific logic. Zig's performance and safety (e.g. no null/dangling pointers by default, etc.) will help in handling many players without the overhead of a garbage-collected runtime, making it efficient for possibly hundreds of concurrent connections.

WebSocket Communication Protocol (JSON-RPC)

Using **WebSocket + JSON-RPC** provides a robust and flexible protocol for client-server communication in this game. Here we outline how the messaging works and ensure it covers all required interactions:

- **WebSocket Layer:** After a client connects to the server's WebSocket endpoint, a persistent bidirectional channel is established. WebSockets are ideal for game communication because they allow the server to push data to clients without each client polling. This low-latency update mechanism is necessary for real-time games. The server will accept the WebSocket upgrade handshake and keep track of the connection object for each player. All game and chat messages will flow through this. We'll use text frames (UTF-8) for JSON messages, as is standard (binary frames aren't needed since JSON is text). Godot's WebSocket client will be used on the front-end, and the Zig server uses a WebSocket server library to manage the connections.

- **JSON-RPC 2.0 Format:** JSON-RPC 2.0 is a lightweight RPC protocol that works nicely over WebSocket. Each message has a predictable JSON structure, making it easy to parse and route. For example, a client request to play a card might look like:

  ```
  {"jsonrpc": "2.0", "method": "playCard", "params": {"card": "AS"},
  "id": 42}
  ```

  The server would respond (if needed) with:

  ```
  {"jsonrpc": "2.0", "result": {"status": "OK"}, "id": 42}
  ```

  Notifications (server -> client or client -> server without expecting reply) have the same format minus the `id`. For instance, a server telling everyone a card was played:

  ```
  {"jsonrpc": "2.0", "method": "cardPlayed", "params": {"player":
  "Alice", "card": "AS"}}
  ```

  There is no `id` because it's not a response to a particular request; it's an unsolicited event. JSON-RPC **supports requests, responses, notifications, and even batch messages** [4], though we may not need batching. This structure will make it easy to add new message types if needed without breaking the protocol.

- **Message Routing and Dispatch:** On the **server side**, if using ZigJR, it will take incoming JSON, parse it, and automatically call the corresponding handler function (registered by method name) [3] [6]. For example, we register that `"playCard"` maps to the function `handlePlayCard` in our code. The library will give us the parameters neatly as Zig types (it can map JSON to Zig types). This saves us from manually decoding JSON keys every time. Similarly, for sending messages, we can use the library to compose JSON strings from Zig types. If we weren't using a library, we'd implement a simple router: parse the JSON (perhaps into a dictionary), do a `switch` or if-else on the `"method"` string, and call the corresponding function. After the function, if it returns a result, we send a response. If it needs to broadcast notifications, those functions will call a broadcast utility. Using a library just streamlines this and ensures compliance with the JSON-RPC spec.

- **Client handling of messages:** On the **client side** (Godot), we will similarly have a dispatcher. We can write a GDScript function that runs every time a message is received on the WebSocket (Godot's WebSocket API likely emits a signal or we can poll using `WebSocketClient.poll()`). When a message arrives as a string, we parse it with `JSON.parse()`. We then inspect if there's an `id` field to tell if it's a response to something the client sent. Typically, the client might not need to send many requests that require responses except maybe joinGame or createGame to know if successful. Those responses can be handled by keeping track of the last request ID. More often, we'll be dealing with server notifications (like game updates). For those, we check the `"method"` and call the relevant handler in the client. We will implement handlers like `onGameStarted(params)`, `onCardPlayed(params)`, `onChatMessage(params)`, etc. For instance, `onGameStarted` will transition the UI to the game scene, display players and their card counts, etc. `onCardPlayed` will update the game UI (remove a card from that player's hand in the UI, show it on table). By structuring the client code to handle these message events, we make the client reactive to server state. The JSON-RPC standard helps because we know exactly how data is formatted (e.g. no custom delimiter issues, since JSON can be framed by WebSocket message boundaries easily – each WebSocket message corresponds to one JSON-RPC message, making things simpler).

- **Reliability and Ordering:** WebSockets guarantee in-order delivery of messages per connection. This is important (we don't want an out-of-order scenario like a turn update arriving before the card play message). So the protocol will naturally preserve sequence from server to each client. We should be careful to ensure our server sends messages in the correct order when broadcasting to multiple clients, but since each broadcast loop will send messages likely back-to-back, and each client processes its queue in order, it should be fine. If using ZigJR's streaming, it can also handle continuous streams of messages properly. In case of any disconnect, the client should attempt to reconnect or at least return the user to the lobby screen with a message. This is more of an edge-case handling – not directly architecture, but something to keep in mind during implementation.

- **Security Considerations:** Although not explicitly asked, one consideration is that without authentication, anyone can join and potentially try to cheat by sending improper JSON commands. We rely on the server logic to validate everything. Also, JSON-RPC doesn't include built-in encryption, but if we run the WebSocket over WSS (TLS), data will be encrypted in transit. We should plan to deploy the game server with SSL (especially if the web client is served over HTTPS, browsers will require WSS for security). We'll also implement basic rate limiting on the server (e.g. don't allow a client to send 1000 chat messages per second and spam others – if such behavior is detected, we can throttle or drop that connection). Zig's efficiency helps here as well.

By using JSON-RPC over WebSocket, we create a clean separation of concerns: **what** actions or events are happening (defined by RPC methods) is separate from **how** they're transmitted (WebSocket). This makes debugging easier and the system more extensible. For example, adding a new tool action is just adding a new RPC method; adding a new feature like a "ping" for latency measurement is simple. The chosen protocol and communication layer are well-suited for the real-time yet structured needs of a multiplayer card game.

AI Opponent and Reinforcement Learning

One major extension of this game is the inclusion of an **AI robot player** that can take part in matches. The plan is to develop this AI using **reinforcement learning (RL)**, training it to play the custom card game effectively. We also want the AI to have multiple difficulty levels to provide appropriate challenges to players. The development of the AI can be seen in two phases: **training** the AI model (offline, potentially using Python or other ML tools) and **integrating** the trained AI into the game server for live play.

- **Defining the Environment:** First, we need to formalize the game as an environment for RL. Card games are typically **imperfect information** environments (players don't see each other's hands), which makes them challenging. We will define the state for the AI as all the information a player has at a given time: the cards in its hand, the number of cards other players have (and possibly what cards have been played or are visible), the turn order, and any other relevant context (like which tools have been used or are available). The actions available to the AI on its turn are: play any valid card from its hand, use any available tool, or if no valid move (unlikely in a shedding game, since one can always play something even if just throw a card), maybe pass (if game rules allow skipping – but likely not needed here). Each game, the final reward could be +1 for winning, 0 for anything else (or perhaps a ranking-based reward for more granularity, but simplest is win/lose). The goal for the RL agent is to maximize its win rate.

- **Training Process:** We will likely use an external framework or library to train the AI due to the complexity of RL. A toolkit like **RLCard** (Reinforcement Learning for card games) could be very useful – *"RLCard is a toolkit for RL in card games, supporting multiple card environments... bridging reinforcement learning and imperfect information games."* [7] . RLCard or similar libraries (OpenAI Gym, PettingZoo for multi-agent, etc.) can help set up the environment and run self-play training. We might implement the game logic in Python (mirroring the Zig logic) to interface with these frameworks easily. For example, create a Python environment where N agents play our card game against each other. We can start with simple agents (random play or some heuristic) to get baseline performance. Then use self-play: initialize one or more neural network based agents and train them via Deep Q-Learning or policy gradient methods by playing many rounds against themselves and learning from outcomes. **Self-play deep reinforcement learning has been very successful** in card games – for instance, the DouZero project mastered DouDizhu (a shedding-type card game where the goal is to empty one's hand) via self-play RL [8] . We can draw inspiration from such research. The state and action space of our game will determine the algorithm; if the game has a large action space, methods like Deep Monte Carlo or policy gradient might be more suitable than vanilla Q-learning. The training would likely involve thousands of simulated games. Since our game is somewhat similar to other shedding games, an RL agent can learn strategies like which card to get rid of first, how to time the use of tools (e.g. maybe saving the swap tool for a crucial moment).

- **AI Difficulty Levels:** We plan to provide multiple difficulty levels for AI opponents (e.g. Easy, Medium, Hard). There are a few ways to achieve this:

- **Different Strategies/Models:** We can train multiple agents or use snapshots from training at different competency. For example, an "Easy" AI could be a simplified heuristic or a model that hasn't fully converged (one that still makes mistakes). A "Hard" AI would be the fully trained model that plays optimally (or close to it). Alternatively, we can use one strong model and just handicap it for lower difficulties. For instance, an easy mode might choose a random action 30% of the time (introducing mistakes), whereas hard mode is purely greedy with the learned policy.

Another method is to have distinct AI algorithms: easy = completely random moves, medium = rule-based (if any simple strategy can be coded), hard = RL-trained model. This ensures a noticeable difference in challenge.

- **Parameter Tuning:** If the RL algorithm is something like epsilon-greedy (common in Q-learning), for easy level we can set a high epsilon (meaning it will explore/randomly play a lot, mimicking a novice), and for hard level epsilon is near 0 (almost always take the best action). Or if using a trained neural network policy, we can sometimes add slight noise to its policy distribution to make it occasionally do suboptimal things for lower difficulty.

- **Multiple Models:** We might even train separate models specialized for certain styles or difficulty. However, maintaining one well-trained model and dialing its performance up or down is often simpler.

- **Integration into the Server:** Once we have a trained AI model (likely developed in Python or using a library), we need to integrate it into our Zig server so that the AI can act as a player. One approach is to **export the trained model** (for example, as an ONNX or TensorFlow Lite model) and then either load it in Zig (if we have a way to do inference in Zig, possibly via calling C libraries or embedding a simple inference engine) or run a separate microservice for AI. Given Zig's focus on performance and that it can interface with C easily, one idea is to use a C/C++ inference library (like ONNX Runtime, or write a simple feed-forward if the model is not huge) that Zig calls into. Another approach is to have the Zig server communicate with a Python script that runs the AI (e.g. via a localhost socket or RPC). However, that adds complexity and latency. If the model is not extremely large (card game models might be a few MB at most), we could aim to do inference in-process. There's also the possibility of implementing a simpler AI that doesn't need a neural network: for instance, a Monte Carlo Tree Search or heuristic-based AI could be coded in Zig directly. But since the brief is RL, we'll assume a neural policy.

In practice, we might prototype the AI in Python, and once satisfied, either translate the policy to Zig (if it's simple enough like a set of learned weights) or use a library in Zig. The AI will function as follows on the server: We designate an AI player by a `Player.ai = true` flag and no associated WebSocket (since it's internal). When it's an AI player's turn, the server will not wait for a network message (human input). Instead, the server invokes the AI logic: gathers the current state (from the AI player's perspective), feeds it into the AI model/policy, obtains an action (e.g. which card to play or whether to use a tool). The server then applies that action as if a client had sent it. It will broadcast the resulting events to all players (so humans see "AI_X played card Y"). Essentially, the AI is just another decision-maker plugged into the game loop. We have to ensure this happens with minimal delay to keep the game flow. A well-optimized model should make decisions quickly (milliseconds). If needed, we could insert a small delay to make the AI's play appear more human (like think for 1 second on Easy, 0.2 seconds on Hard, etc., to mimic reaction times and also to not overwhelm human players by acting instantly).

- **Training Iteration:** We should be prepared to iterate on the AI. After initial training, we'll test the AI in the actual game environment with humans. If it's too weak or too strong or behaves oddly (e.g. uses the swap tool nonsensically), we may need to adjust the reward function or training scenarios. For example, we might shape the reward to encourage the AI to finish as early as possible, and perhaps give a slight negative reward for remaining cards (so it learns to shed high cards first). In reinforcement learning, designing the reward and training against a variety of opponents is key. We might train the AI via self-play (multiple instances of the same network playing each other), which tends to gradually improve all of them. There is research showing self-play can achieve high-level play without supervision in games like poker. In fact, algorithms like Deep CFR or RL with opponent modeling have been used for poker. Our game is simpler (no betting or bluffing element, presumably just playing cards), so it should be tractable.

- **Example Reference:** To validate our approach, consider **DouZero**, which is an RL framework that mastered DouDizhu (a 3-player shedding card game). It used self-play with deep reinforcement learning to achieve superhuman performance, despite the game's large complexity [8]. DouZero's success demonstrates that with enough training, an AI can learn when to play certain cards or when to hold them, even in imperfect information settings. We won't likely need something as sophisticated as DouZero's full algorithm (which uses deep Monte Carlo with big neural networks) unless our game is similarly complex. But it's reassuring that the approach is viable. Another reference is **RLCard**, which provides implementations of DQN, CFR (counterfactual regret minimization), etc., for card games like Texas Hold'em and UNO [9]. We can leverage such tools to jump-start our training environment and focus on tweaking it for our custom rules.

- **Bot Participation:** In the actual game deployment, AI bots can be used in two ways: **filling seats** in games where not enough humans join, and **single-player mode** where a user just wants to play against AIs. The architecture easily supports this: when creating a game, if the player selects a single-player option or if after waiting some time not all seats are filled, the server can instantiate AI players (with difficulty as chosen by the user or default). These AI players will occupy player slots and the humans will see them just like other players (maybe with a label "BOT" on their name). The server will run them as described (automatically make their moves). This adds a lot of value as players can practice or play at off-peak times.

- **Scaling and Performance for AI:** Running AI inference will consume CPU (or GPU if we had one, but on the server likely just CPU). If many games have many AI players, we need to ensure the server can handle it. We can mitigate this by perhaps capping how many AI are allowed per game when server load is high, or by making the AI computation efficient (e.g. using a simpler model or running inference asynchronously so as not to block the main server loop). We might dedicate a thread or task queue for AI computations – e.g. when it's a bot's turn, offload the decision computation to a separate thread pool, then callback with the action. This way the server can continue handling other games in parallel. Given that card game decisions are not extremely intensive (compared to, say, deep learning for image recognition), this should be manageable with modern CPUs as long as models aren't huge.

In conclusion, the AI robot subsystem will significantly enhance the game's appeal. We plan to train it using reinforcement learning, leveraging existing research and toolkits for card games. By integrating the trained AI into the Zig backend, we maintain a unified server architecture where both human and AI players are handled through the same game logic. Different difficulty levels ensure that players of all skill ranges can enjoy the game, from casual fun with a weak AI to a serious challenge against a strong AI. The combination of Godot frontend, Zig server, and RL-trained AI will result in a comprehensive gaming system that is engaging, scalable, and modern.

Conclusion and Further Considerations

We have outlined a comprehensive plan covering all aspects of the multiplayer poker game: from the Godot 4.5 client (for browser and mobile) through the Zig-powered backend with WebSocket JSON-RPC communication, to the integration of a reinforcement learning-based AI opponent. This architecture emphasizes separation of concerns (UI vs logic), real-time communication, and scalability. A few additional points to consider as development moves forward:

- **Testing & Quality:** We will implement unit tests for game logic in the server (using Zig's testing features) to ensure rules and tools work correctly. We'll also test the full stack in stages: first get multiplayer working with humans, then introduce AI bots and verify they don't disrupt the game flow. Load testing can be done by simulating many games to ensure the server can handle it.
- **Deployment:** The server can be a standalone binary that we host on a cloud or a game server. If targeting browsers, the server should handle potentially many concurrent WebSocket connections. Zig's efficiency should allow a single server instance to handle the load of our expected user count (depending on popularity, we might consider a lobby server + multiple game servers architecture, but that seems overkill initially). The Godot HTML5 build will be hosted on a web server (or itch.io, etc.), and mobile builds distributed via app stores if needed.
- **Security & Fairness:** Because it's an online game, we should secure the WebSocket (use WSS) and possibly implement basic cheat prevention. The server being authoritative already prevents clients from giving themselves extra cards or illegal moves. We may also want to prevent players from opening multiple sessions to spy on more than one hand (which is a general issue in online card games without authentication). With no login, one person could open multiple browser tabs and join the same game under different names – we can't robustly prevent that without accounts, but we can mitigate by not showing which cards belong to which connection on the client side (which we already do not). It's mostly an honor system in casual play.
- **Extensibility:** The architecture is designed such that adding features later (matchmaking, different game modes, tournaments, richer user profiles) would be possible. For example, a matchmaking system could be a layer on top of the lobby that automatically groups players into games, but since it's not needed now, we keep manual rooms. Similarly, a ranking or tournament could be built by tracking wins and scheduling games, without changing the core game logic module.
- **References & Inspiration:** We have drawn on known solutions (Godot's cross-platform networking support, Zig's JSON-RPC capabilities [3], RLCard and research like DouZero for AI [8]) to ensure our plan is grounded in proven technology. By using these as guidelines, we minimize risk in implementation.

By following this plan, we will develop a multiplayer card game that is fun and accessible on multiple devices, with an architecture that is robust and maintainable. The combination of a **Godot frontend** and a **Zig backend** connected via **WebSockets/JSON-RPC** will provide a smooth gameplay experience, and the addition of **AI opponents** will add depth and replayability to the game. Each component has a clear role, and together they fulfill all the requirements set out for the project.

---

[1] Networking - WebSocket - 《Godot v4.3 Documentation》 - 书栈网 ...
https://www.bookstack.cn/read/godot-4.3-en/baf02164a5e121ed.md

[2] [5] GitHub - karlseguin/websocket.zig: A websocket implementation for zig
https://github.com/karlseguin/websocket.zig

[3] [4] [6] GitHub - williamw520/zigjr: A Lightweight Zig Library for JSON-RPC 2.0
https://github.com/williamw520/zigjr

[7] [9] RLCard: A Toolkit for Reinforcement Learning in Card Games — RLcard 0.0.1 documentation
https://rlcard.org/

[8] GitHub - kwai/DouZero: [ICML 2021] DouZero: Mastering DouDizhu with Self-Play Deep Reinforcement Learning | 斗地主AI
https://github.com/kwai/DouZero