These instructions are optimized for an AI platform like Jules, breaking down the requirements for a **scalable backend server** (using a common web framework) and a **single, responsive client** built with a cross-platform game engine.

Phase 3: Backend Server and Cross-Platform Client

GOAL: Establish the persistent game state (backend) and a single, responsive client interface accessible on mobile and web.

Module 7: Backend Server (Node.js/Express or Python/Django)

GOAL: Create a robust API to manage player data, game state, and real-time PvP sessions.

Component	Instruction	Output Requirement
Server Framework	Initialize a backend server using **Node.js with Express** or **Python with Django/Flask**. This choice must support WebSocket for real-time communication.	Basic API structure with routing and environment configuration.
Database Schema	Design a schema for persistent player data. Key collections/tables must include: Players (ID, \mathbb{C}, C-XP, ram_max, is_vip, **K-Map JSON**), Missions (ID, type, difficulty), and Modules (Player ID, script content).	Schema definition file (e.g., ORM models or Mongoose schema) to store all critical game state data.
REST API (State Management)	Implement core **REST endpoints** for persistent data: POST /api/login, GET /api/player/state, PUT /api/player/kmap (for saving K-Map updates), and POST /api/mission/complete (to award \mathbb{C} and C-XP).	Functional CRUD endpoints to manage player and mission data.
Real-Time PvP Handler (WebSocket)	Implement a **WebSocket server** (e.g., Socket.io or Django Channels) to handle dynamic PvP sessions. This channel will manage live **NexusScript** execution status and the shared **LAN State**.	A persistent, authenticated WebSocket connection endpoint.

Module 8: Cross-Platform Client (Phaser or Similar)

GOAL: Create a single, responsive client using a mobile/web-friendly engine (like **Phaser 3**) that communicates solely with the backend API.

Component	Instruction	Output Requirement
Game Engine Choice	Utilize **Phaser 3** (or a similar engine like PixiJS or Godot/JavaScript export) for the client. The output must be easily packaged for **Android/iOS** (via Cordova/Capacitor) and run directly in a web browser.	Basic HTML/JS structure with the chosen engine initialized and running.
Interface Design (Simple UI)	Design a single, fixed-size interface with three main responsive areas: 1) **NexusShell Input/Output** (the primary terminal), 2) **VC Status Sidebar** (displaying RAM, CPU, \mathbb{C}, Level), and 3) **Module Editor Panel** (for edit command use).	A responsive layout that adapts correctly to both portrait mobile and landscape web views.
Terminal Input Component	Implement the primary input field where players type **NexusScript** commands. This component must have a simple, tactile input area suitable for mobile keyboards.	A JavaScript component that captures input and sends it as a string to the backend's REST API or WebSocket.
Server Communication Layer	Integrate both **REST** and **WebSocket** communication within the client. Use REST for initial state loading and saving, and WebSocket for real-time PvP interaction, command responses, and log streams.	A dedicated client-side service to handle API calls and manage WebSocket connections.

Module 9: Integration and Command Execution Flow

GOAL: Define the final flow for executing player commands, ensuring the server handles the game logic, not the client.

Component	Instruction	Output Requirement
Command Execution Flow	The client **MUST NOT** execute **NexusScript** logic. The flow is: **Client Input \rightarrow Server API Endpoint \rightarrow Server VC Logic \rightarrow Server Response \rightarrow Client Output Update.**	A POST /api/cmd/execute endpoint that receives the command string, runs the NexusShell Parser on the server, and returns the resulting text output and any state changes (e.g., \mathbb{C} update).
Real-Time PvP Flow	Use the WebSocket for time-critical commands. When a player executes scan or	WebSocket handler functions to manage and broadcast updates to the PvP LAN state.

Component	Instruction	Output Requirement
	exploit, the command is sent via WebSocket. The server calculates the result, updates the shared **LAN State** (database), and sends the result back to *all* players in the session in real-time.	
Client-Side Visual Feedback	Implement visual feedback for slow operations. The client should display a non-blocking **"Executing..."** or **"Processing (VIP Speed)"** animation while waiting for the server's response to an operation like hashcrack.	UI elements that display status messages based on server response latency.