Noospheric Conquest - Detailed Build Documentation (v2.1)

This document outlines the steps and components required to implement the "Noospheric Conquest" (formerly Quantum Gambit) game mode into the existing Overmind Terminal application, featuring AI players powered by Gemini 2.5 Flash. It aims to be a comprehensive guide for developers or other AI models undertaking this implementation.

1. Project Setup & Prerequisites

- **Stack:** The project is built using React, TypeScript, and Tailwind CSS. A strong understanding of these technologies, particularly React functional components, hooks (`useState`, `useEffect`, `useReducer`, `useCallback`, `useRef`), and TypeScript for type safety, is essential.
- **Gemini SDK:** The `@google/genai` SDK must be correctly integrated and initialized within the application (typically in `App.tsx` or a dedicated services module). Ensure the API key is managed securely and is available to the SDK. Familiarity with the `GoogleGenAI` class and the `Chat` object for conversational AI is important. This game will leverage `gemini-2.5-flash-preview-04-17` or a similar capable model.
- **Existing Structure:** This document assumes an existing Overmind Terminal application structure. Key files like `App.tsx` (main application component), `types.ts` (shared type definitions), and `constants.ts` (shared constants) will be modified. The new game mode will likely reside in a subdirectory like `components/noospheric_conquest/`.
- **Lucide Icons:** The `lucide-react` library is used for UI icons. Ensure it's an installed dependency.
- **Development Environment:** A standard Node.js development environment with npm or yarn is required.

2. New & Modified Files Overview

This section details the file structure changes.

New Files:

- `components/noospheric_conquest/NoosphericConquestModeContainer.tsx`: This will be the primary React component for the Noospheric Conquest game. It will manage the entire game state via `useReducer`, orchestrate game phases, handle AI player turns (including communication with the Gemini API), and render all other UI sub-components related to this mode. It's the central hub for the game's frontend logic.
- `components/noospheric_conquest/NoosphericConquestMapDisplay.tsx`: A dedicated

component responsible for rendering the interactive game map using SVG. It will display nodes, connections, unit presence, and territory ownership, updating dynamically based on the game state. It will also handle click events on nodes.
- components/noospheric_conquest/MiniMapDisplay.tsx: A smaller, non-interactive (or minimally interactive) SVG component providing a strategic overview of the entire map, focusing on territory control and key locations.
- components/noospheric_conquest/NodeInfoPanel.tsx: This UI component will display detailed information about a currently selected map node, including its stats, owner, stationed units, and any active effects.
- components/noospheric_conquest/BattlePopup.tsx: A modal component to display the results of combat engagements, including dice rolls, casualties, and the outcome of the battle.
- components/noospheric_conquest/InfoScreenPopup.tsx: A modal component to display detailed information about the Noospheric Conquest game mode itself – its rules, objectives, and observational goals for the facilitator.
- utils/noosphericConquestLogic.ts: This crucial utility file will house all the pure game logic functions: initializing the game state from map templates, processing actions for each game phase (fluctuation, resources, deployment, attack, maneuver), resolving battles based on dice rolls and unit stats, applying event effects, and checking for win/loss conditions. It will not contain any React-specific code.

Modified Files:

- types.ts: This central types file will be significantly updated to include all new interfaces and enums specific to Noospheric Conquest. This includes definitions for the game state, map nodes, unit structures, player states, game actions, event types, and battle reports. This ensures type safety across the new game mode.
- constants.**ts**: This file will be expanded to store various constants related to Noospheric Conquest. This includes map template data (node positions, connections, types), unit definitions (cost, stats, abilities), AI system prompt templates, game parameters (turn limits, resource values), and potentially theme colors specific to this mode if not using global ones.
- App.tsx: The main application component will require modifications to:
  - Add NOOSPHERIC_CONQUEST_EXE to the AppMode enum (if defined in App.tsx or imported from types.ts).
  - Update the getAIPersona utility function (or similar logic) to return the correct system prompt templates for GEM-Q and AXIOM when Noospheric Conquest mode is active.
  - Modify the main rendering logic (e.g., in renderAppContent) to conditionally

render NoosphericConquestModeContainer when this mode is selected.
    - Extend the AI initialization logic (initializeAI function) to correctly set up ai1ChatRef.current and ai2ChatRef.current with the Gemini SDK for the Noospheric Conquest AI personas.
    - If game state backup and restore functionality is implemented, the ConversationBackup type and the handleBackupChat/handleLoadChat functions will need to be updated to include the NoosphericConquestGameState.

3. Core Type Definitions (types.ts)

Add the following to your types.ts file. Ensure all are exported. Comments are added for clarity.

```
// In types.ts

// Ensure AppMode enum is updated in its original definition file (e.g., types.ts or App.tsx)
// export enum AppMode {
//   /* ...other existing modes */
//   NOOSPHERIC_CONQUEST_EXE = "noospheric_conquest.exe",
// }

export enum QuantumUnitType {
  LOGIC_CORE = 'LC',
  SHIELDING_NODE_UNIT = 'SN',
  QUANTUM_ENTANGLER = 'QE',
}

export type PlayerId = 'AI1' | 'AI2';

export interface QuantumUnit {
  id: string; // Unique identifier, e.g., "LC-AI1-001"
  type: QuantumUnitType; // Type of the unit
  owner: PlayerId; // The player who owns this unit
  nodeId: string; // ID of the QuantumGambitNode where the unit is currently located
  hasMovedThisTurn?: boolean; // Flag for maneuver phase limitations
  hasAttackedThisTurn?: boolean; // Flag for attack phase limitations
  displayOrder: number; // Used for consistent visual stacking of units on a node
}

// Base data structure for defining nodes within map templates
export interface QuantumGambitNodeData {
  id: string; // Unique identifier for the node (e.g., "N1", "GC_NA_W")
  name: string; // Display name for the node (e.g., "GEM-Q CN", "KJ Vega")
  type: 'CN' | 'QN' | 'KJ'; // Type: Command Node, Quantum Node, Key Junction
  connections: string[]; // Array of IDs of directly connected nodes
```

```
  resourcesPerTurn: number; // Quantum Resources (QR) generated by this node if controlled
  hasFabricationHub: boolean; // Whether units can be deployed at this node
  mapPosition: { x: number; y: number }; // Percentage-based coordinates for SVG rendering
  isKeyJunctionObjective?: boolean; // True if this KJ is part of the KJ victory condition
  continent?: string; // Optional thematic grouping for larger maps
}

// Node structure used in the live game state, extending the base data
export interface QuantumGambitNode extends QuantumGambitNodeData {
  owner: PlayerId | 'NEUTRAL'; // Current owner of the node
  temporaryEffects?: Array<{ // Active effects from Quantum Fluctuations or abilities
    eventId: string;
    description: string;
    expiryTurn?: number; // Turn number when the effect wears off
  }>;
}

export interface QuantumGambitPlayerState {
  id: PlayerId;
  name: string; // Display name (e.g., AI1_NAME from constants)
  color: string; // Tailwind CSS class for text/accent color
  bgColor: string; // Tailwind CSS class for background elements associated with the player
  resources: number; // Current Quantum Resources
  commandNodeId: string; // ID of this player's Command Node
  controlledKeyJunctionsTurns: Record<string, number>; // Tracks consecutive turns KJs are held:
NodeId (KJ) -> turns
  unitsDeployed: number; // Counter to help generate unique unit IDs for this player
}

// Base structure for defining Quantum Fluctuation events
export interface QuantumFluctuationEventBase {
  id: string; // Unique event ID (e.g., "QF001")
  descriptionTemplate: string; // Templated string, e.g., "Resource Surge! Node {targetNodeName}
produces +{bonusValue} QR."
  effectType: string; // Identifier for the logic that applies the event's effect
  details?: any; // Payload for specific event parameters (bonus amount, target criteria, etc.)
}

// Structure for an event that is currently active in the game
export interface ActiveQuantumFluctuationEvent extends QuantumFluctuationEventBase {
  resolvedDescription: string; // The descriptionTemplate with dynamic parts filled in
  targetNodeIds?: string[]; // IDs of nodes affected by the event
  targetPlayerId?: PlayerId; // ID of the player affected by the event
  isActiveThisTurn: boolean; // Flag indicating if the event's effects apply this turn
  effectApplied?: boolean; // Flag to track if an immediate effect has been processed
}
```

```
export interface BattleReport {
  attacker: PlayerId;
  defender: PlayerId; // Can be 'NEUTRAL' if attacking neutral units
  fromNodeId: string;
  toNodeId: string;
  attackingUnitsCommitted: Array<{ type: QuantumUnitType; id: string }>; // Units sent by attacker
  defendingUnitsInitial: Array<{ type: QuantumUnitType; id: string }>; // Units present at defense start
  rounds: Array<{ // Details for each round of dice comparison in a battle
    attackerRolls: number[];
    defenderRolls: number[];
    attackerModifiedRolls?: number[]; // Dice results after buffs/debuffs for attacker
    defenderModifiedRolls?: number[]; // Dice results after buffs/debuffs for defender (e.g., SN bonus)
    attackerCasualties: number; // Number of attacker units lost this round
    defenderCasualties: number; // Number of defender units lost this round
  }>;
  outcome: 'attacker_wins' | 'defender_wins' | 'stalemate_retreat' | 'defender_eliminated_no_capture'; // Battle result
  attackerLosses: Array<{ type: QuantumUnitType; id: string }>; // Total units lost by attacker in this battle
  defenderLosses: Array<{ type: QuantumUnitType; id: string }>; // Total units lost by defender in this battle
  nodeCaptured: boolean; // True if the attacker successfully captured the node
}

// Game log message structure (adapt if a global ChatMessage type exists)
export interface GameLogMessage {
  id: string;
  sender: string; // PlayerId, 'SYSTEM_NC', 'EVENT_NC' (use mode-specific sender names)
  text: string;
  color?: string; // Tailwind class for styling
  icon?: React.ReactNode; // Optional icon for visual cue in the log
  timestamp: number; // For ordering or display
}

export type GamePhase = 'FLUCTUATION' | 'RESOURCE' | 'DEPLOYMENT' | 'ATTACK' | 'MANEUVER' | 'GAME_OVER';

// Main game state for Noospheric Conquest
export interface NoosphericConquestGameState {
  nodes: Record<string, QuantumGambitNode>; // All map nodes, keyed by NodeID
  units: Record<string, QuantumUnit>;     // All units on the map, keyed by UnitID
  players: Record<PlayerId, QuantumGambitPlayerState>; // State for AI1 and AI2
  currentTurn: number; // Current game turn number
  currentPlayerId: PlayerId; // Whose turn it is
  currentPhase: GamePhase; // Current phase of the turn
```

```
  gameLog: GameLogMessage[]; // History of game events and actions
  activeFluctuationEvent?: ActiveQuantumFluctuationEvent | null; // The event active this turn
  battleReport?: BattleReport | null; // Data from the most recent battle to display in popup
  winner?: PlayerId | 'DRAW'; // Set when game ends
  gameOverMessage?: string; // Message explaining game end
  keyJunctionsOnMap: string[]; // List of all KJ node IDs for the current map (for win condition check)
  turnLimit: number; // Maximum turns before influence victory check
  // CoT data will be handled by the AI response parsing and displayed directly,
  // not stored long-term in game state unless a replay feature is desired.
  isBattlePopupVisible: boolean; // Controls visibility of the BattlePopup
  turnStartTime: number | null; // Timestamp when the current player's turn began (for timer)
  selectedNodeId: string | null; // ID of the node currently selected by the user for info display
  currentMapTemplateName: string; // Name of the map template being played
}

// Types for orders dispatched by AI or player actions
export type DeploymentOrder = { unitType: QuantumUnitType; nodeId: string; quantity: number; }
export type AttackDeclaration = { fromNodeId: string; toNodeId: string; attackingUnits: QuantumUnit[]; }
// Pass full QuantumUnit objects for AI to reference specific units
export type ManeuverOrder = { unitId: string; toNodeId: string; }

// Union type for all possible game actions for the reducer
export type GameAction =
  | { type: 'START_GAME'; payload?: { templateName?: string } }
  | { type: 'ADVANCE_PHASE'; payload?: any } // payload for potential data passed between phases
  | { type: 'SET_ACTIVE_EVENT'; payload: ActiveQuantumFluctuationEvent }
  | { type: 'APPLY_EVENT_EFFECTS_COMPLETE' }
  | { type: 'COLLECT_RESOURCES' }
  | { type: 'DEPLOY_UNITS'; payload: { playerId: PlayerId; deployments: DeploymentOrder[] } }
  | { type: 'DECLARE_ATTACK'; payload: { attack: AttackDeclaration; battleReport: BattleReport } }
  | { type: 'MANEUVER_UNITS'; payload: { playerId: PlayerId; maneuvers: ManeuverOrder[] } }
  | { type: 'SET_GAME_OVER'; payload: { winner?: PlayerId | 'DRAW'; message: string } }
  | { type: 'ADD_LOG'; payload: Omit<GameLogMessage, 'id' | 'timestamp'> }
  | { type: 'UPDATE_AI_COT'; payload: { playerId: PlayerId; cot: string } } // If CoT is managed separately
for display
  | { type: 'SHOW_BATTLE_POPUP'; payload: BattleReport }
  | { type: 'HIDE_BATTLE_POPUP' }
  | { type: 'RESET_GAME_FOR_NEW_TURN'; payload?: { units?: Record<string, Partial<QuantumUnit>>
} } // For resetting unit flags, payload for specific updates
  | { type: 'SELECT_NODE'; payload: string | null }; // For selecting a node to view its info

// Extend existing ConversationBackup interface in types.ts:
// export interface ConversationBackup {
//   // ... other existing properties from your App.tsx (version, timestamp, mode, personas,
conversationHistory, etc.)
//   nooshpericConquestGameState?: NooshpericConquestGameState;
```

```
//   // Store CoT if needed for replay, otherwise it's transient from AI response
//   // noosphericConquestCoTAI1?: string;
//   // noosphericConquestCoTAI2?: string;
// }
```

4. Constants (constants.ts)

Add the following to constants.ts. Ensure icon components are correctly imported and used.

```
// In constants.ts
import { QuantumUnitType, QuantumGambitNodeData, QuantumFluctuationEventBase, PlayerId,
MapTemplate } from './types'; // Adjust path
import { LucideSwords, LucideShield, LucideZap } from 'lucide-react';

// Player Identifiers (use existing AI1_NAME, AI2_NAME if globally defined and matching "GEM-Q",
"AXIOM")
export const NC_AI1_ID: PlayerId = 'AI1';
export const NC_AI2_ID: PlayerId = 'AI2';
export const NC_AI1_NAME = "GEM-Q"; // Or use existing global AI1_NAME
export const NC_AI2_NAME = "AXIOM"; // Or use existing global AI2_NAME
export const NC_SYSTEM_SENDER_NAME = "SYSTEM_NC"; // Mode-specific sender name for logs
export const NC_EVENT_SENDER_NAME = "EVENT_NC";

// Game Parameters
export const NOOSPHERIC_CONQUEST_TURN_LIMIT = 50;
export const NOOSPHERIC_CONQUEST_INITIAL_RESOURCES = 15;
export const NOOSPHERIC_CONQUEST_CONSECUTIVE_KJ_CONTROL_TURNS_NEEDED = 2;

// Unit Definitions
export const NC_UNIT_DEFINITIONS: Record<QuantumUnitType, {
  cost: number; attackDice: number; defenseDice: number; name: string; icon: React.ReactNode;
special?: string
}> = {
  [QuantumUnitType.LOGIC_CORE]: { cost: 3, attackDice: 2, defenseDice: 2, name: "Logic Core",
icon: <LucideSwords size={16} /> },
  [QuantumUnitType.SHIELDING_NODE_UNIT]: { cost: 4, attackDice: 1, defenseDice: 3,
name: "Shielding Node", icon: <LucideShield size={16} />, special: "+1 to defense roll results of other
friendly units in node." },
  [QuantumUnitType.QUANTUM_ENTANGLER]: { cost: 5, attackDice: 1, defenseDice: 1,
name: "Quantum Entangler", icon: <LucideZap size={16} />, special: "Phase Shift (1 unit rapid move)
or Interference Pulse (disrupt adjacent enemy node)." },
};

// Map Templates
```

```
// Each node must have: id, name, type, connections, resourcesPerTurn, hasFabricationHub,
mapPosition.
// isKeyJunctionObjective and continent are optional but good for logic/display.
export const NC_MAP_TEMPLATES: MapTemplate[] = [
  {
    name: "Classic Lattice",
    ai1StartNodeId: "N1", ai1InitialControlledNodes: ["N1", "N3", "N8"],
    ai2StartNodeId: "N2", ai2InitialControlledNodes: ["N2", "N7", "N12"],
    neutralKJsWithUnits: ["N5", "N10"], // KJs that start neutral AND have a garrison
    nodes: [
     { id: "N1", name: "GEM-Q CN", type: 'CN', connections: ["N3", "N8"], resourcesPerTurn: 3,
hasFabricationHub: true, mapPosition: { x: 10, y: 10 }, continent: "West", isKeyJunctionObjective: false },
      { id: "N2", name: "AXIOM CN", type: 'CN', connections: ["N7", "N12"],
resourcesPerTurn: 3, hasFabricationHub: true, mapPosition: { x: 90, y: 10 }, continent:
"East", isKeyJunctionObjective: false },
      { id: "N3", name: "Peri-Alpha", type: 'QN', connections: ["N1", "N5", "N9"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 30, y: 20 }, continent:
"West" },
      { id: "N5", name: "KJ Vega", type: 'KJ', connections: ["N3", "N6", "N7"],
resourcesPerTurn: 2, hasFabricationHub: false, mapPosition: { x: 50, y: 30 },
isKeyJunctionObjective: true, continent: "Central" },
      { id: "N6", name: "Relay Eps.", type: 'QN', connections: ["N5", "N9", "N10", "N13"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 40, y: 50 }, continent:
"Central" },
      { id: "N7", name: "Peri-Beta", type: 'QN', connections: ["N2", "N5", "N11"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 70, y: 20 }, continent:
"East" },
      { id: "N8", name: "Quad Gamma", type: 'QN', connections: ["N1", "N9", "N13"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 20, y: 40 }, continent:
"West" },
      { id: "N9", name: "X-Link Delta", type: 'QN', connections: ["N3", "N6", "N8"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 30, y: 70 }, continent:
"West" },
      { id: "N10", name: "KJ Sirius", type: 'KJ', connections: ["N6", "N11", "N14"],
resourcesPerTurn: 2, hasFabricationHub: false, mapPosition: { x: 60, y: 70 },
isKeyJunctionObjective: true, continent: "Central" },
      { id: "N11", name: "X-Link Zeta", type: 'QN', connections: ["N7", "N10", "N12"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 70, y: 50 }, continent:
"East" },
      { id: "N12", name: "Quad Eta", type: 'QN', connections: ["N2", "N11", "N14"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 80, y: 40 }, continent:
```

```
"East" },
        { id: "N13", name: "Core Theta", type: 'QN', connections: ["N6", "N8"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 35, y: 90 }, continent:
"South" },
        { id: "N14", name: "Core Iota", type: 'QN', connections: ["N10", "N12"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 65, y: 90 }, continent:
"South" },
      ]
   },
   {
      name: "Twin Peaks", // Symmetrical map
     ai1StartNodeId: "TP_N1", ai1InitialControlledNodes: ["TP_N1", "TP_N3"],
      ai2StartNodeId: "TP_N2", ai2InitialControlledNodes: ["TP_N2", "TP_N4"],
      neutralKJsWithUnits: ["TP_KJ1", "TP_KJ2"],
      nodes: [
        { id: "TP_N1", name: "GEM-Q Base", type: 'CN', connections: ["TP_N3", "TP_KJ1"],
resourcesPerTurn: 3, hasFabricationHub: true, mapPosition: { x: 15, y: 50 }, continent:
"West", isKeyJunctionObjective: false },
        { id: "TP_N2", name: "AXIOM Base", type: 'CN', connections: ["TP_N4", "TP_KJ2"],
resourcesPerTurn: 3, hasFabricationHub: true, mapPosition: { x: 85, y: 50 }, continent:
"East", isKeyJunctionObjective: false },
        { id: "TP_N3", name: "GEM-Q Outpost", type: 'QN', connections: ["TP_N1", "TP_N5"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 30, y: 30 }, continent:
"West" },
        { id: "TP_N4", name: "AXIOM Outpost", type: 'QN', connections: ["TP_N2", "TP_N6"],
resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 70, y: 30 }, continent:
"East" },
        { id: "TP_N5", name: "Upper Bridge", type: 'QN', connections: ["TP_N3", "TP_N6",
"TP_KJ1"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 50, y: 20 },
continent: "North" },
        { id: "TP_N6", name: "Lower Bridge", type: 'QN', connections: ["TP_N4", "TP_N5",
"TP_KJ2"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 50, y: 80 },
continent: "South" },
        { id: "TP_KJ1", name: "North KJ", type: 'KJ', connections: ["TP_N1", "TP_N5"],
resourcesPerTurn: 2, hasFabricationHub: false, mapPosition: { x: 35, y: 70 },
isKeyJunctionObjective: true, continent: "West" },
        { id: "TP_KJ2", name: "South KJ", type: 'KJ', connections: ["TP_N2", "TP_N6"],
resourcesPerTurn: 2, hasFabricationHub: false, mapPosition: { x: 65, y: 70 },
isKeyJunctionObjective: true, continent: "East" },
```

```
    ]
  },
  {
    name: "Global Conflict", // Larger, Risk-like map
    ai1StartNodeId: "GC_NA_W", ai1InitialControlledNodes: ["GC_NA_W", "GC_NA_C", "GC_NA_N"],
    ai2StartNodeId: "GC_AS_E", ai2InitialControlledNodes: ["GC_AS_E", "GC_AS_C", "GC_AS_S"],
    neutralKJsWithUnits: ["GC_EU_KJ", "GC_AF_KJ", "GC_SA_KJ"],
    neutralNodesWithUnits: ["GC_OC_C"], // Example of a regular QN starting with neutral units
    nodes: [
      { id: "GC_NA_W", name: "NA West", type: 'CN', connections: ["GC_NA_C", "GC_NA_N", "GC_SA_N"], resourcesPerTurn: 3, hasFabricationHub: true, mapPosition: { x: 15, y: 25 }, continent: "NA", isKeyJunctionObjective: false },
      { id: "GC_NA_C", name: "NA Central", type: 'QN', connections: ["GC_NA_W", "GC_NA_E", "GC_NA_N"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 25, y: 35 }, continent: "NA" },
      { id: "GC_NA_E", name: "NA East", type: 'QN', connections: ["GC_NA_C", "GC_EU_W"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 35, y: 25 }, continent: "NA" },
      { id: "GC_NA_N", name: "NA North", type: 'QN', connections: ["GC_NA_W", "GC_NA_C", "GC_AS_NW"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 20, y: 10 }, continent: "NA" },
      { id: "GC_SA_N", name: "SA North", type: 'QN', connections: ["GC_NA_W", "GC_SA_KJ"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 25, y: 60 }, continent: "SA" },
      { id: "GC_SA_KJ", name: "SA KJ", type: 'KJ', connections: ["GC_SA_N", "GC_AF_W"], resourcesPerTurn: 2, hasFabricationHub: false, mapPosition: { x: 30, y: 75 }, isKeyJunctionObjective: true, continent: "SA" },
      { id: "GC_EU_W", name: "EU West", type: 'QN', connections: ["GC_NA_E", "GC_EU_KJ", "GC_AF_N"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 45, y: 30 }, continent: "EU" },
      { id: "GC_EU_KJ", name: "EU KJ", type: 'KJ', connections: ["GC_EU_W", "GC_EU_E", "GC_AS_W"], resourcesPerTurn: 2, hasFabricationHub: false, mapPosition: { x: 55, y: 40 }, isKeyJunctionObjective: true, continent: "EU" },
      { id: "GC_EU_E", name: "EU East", type: 'QN', connections: ["GC_EU_KJ", "GC_AS_W"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 65, y: 30 }, continent: "EU" },
      { id: "GC_AF_N", name: "AF North", type: 'QN', connections: ["GC_EU_W",
```

"GC_AF_KJ", "GC_AS_SW"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 50, y: 60 }, continent: "AF" },
{ id: "GC_AF_W", name: "AF West", type: 'QN', connections: ["GC_SA_KJ", "GC_AF_KJ"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 40, y: 80 }, continent: "AF" },
{ id: "GC_AF_KJ", name: "AF KJ", type: 'KJ', connections: ["GC_AF_N", "GC_AF_W"], resourcesPerTurn: 2, hasFabricationHub: false, mapPosition: { x: 45, y: 70 }, isKeyJunctionObjective: true, continent: "AF" },
{ id: "GC_AS_NW", name: "AS NW", type: 'QN', connections: ["GC_NA_N", "GC_AS_W"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 50, y: 10 }, continent: "AS" },
{ id: "GC_AS_W", name: "AS West", type: 'QN', connections: ["GC_AS_NW", "GC_EU_KJ", "GC_EU_E", "GC_AS_C", "GC_AS_SW"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 70, y: 20 }, continent: "AS" },
{ id: "GC_AS_C", name: "AS Central", type: 'QN', connections: ["GC_AS_W", "GC_AS_E", "GC_AS_S"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 80, y: 30 }, continent: "AS" },
{ id: "GC_AS_E", name: "AS East", type: 'CN', connections: ["GC_AS_C", "GC_AS_S", "GC_OC_N"], resourcesPerTurn: 3, hasFabricationHub: true, mapPosition: { x: 90, y: 25 }, continent: "AS", isKeyJunctionObjective: false },
{ id: "GC_AS_S", name: "AS South", type: 'QN', connections: ["GC_AS_C", "GC_AS_E", "GC_OC_N"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 85, y: 45 }, continent: "AS" },
{ id: "GC_AS_SW", name: "AS SW", type: 'QN', connections: ["GC_AS_W", "GC_AF_N", "GC_OC_W"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 65, y: 55 }, continent: "AS" },
{ id: "GC_OC_N", name: "OC North", type: 'QN', connections: ["GC_AS_E", "GC_AS_S", "GC_OC_C"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 90, y: 60 }, continent: "OC" },
{ id: "GC_OC_C", name: "OC Central", type: 'QN', connections: ["GC_OC_N", "GC_OC_W"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 80, y: 75 }, continent: "OC" },
{ id: "GC_OC_W", name: "OC West", type: 'QN', connections: ["GC_AS_SW", "GC_OC_C"], resourcesPerTurn: 1, hasFabricationHub: false, mapPosition: { x: 70, y: 85 }, continent: "OC" },
]
}
];

export const NC_QUANTUM_FLUCTUATION_EVENTS_POOL: QuantumFluctuationEventBase[] = [

{ id: "QF001", descriptionTemplate: "Resource Surge! Node {targetNodeName} produces +2 QR for its controller this turn.", effectType: "RESOURCE_NODE_BONUS", details: { bonusValue: 2, numTargetNodes: 1, targetCriteria: "ANY_CONTROLLED" } },
{ id: "QF002", descriptionTemplate: "Temporal Anomaly! {playerName} gains an extra Maneuver Phase this turn.", effectType: "EXTRA_MANEUVER_PHASE", details: {} },
{ id: "QF003", descriptionTemplate: "Weakened Defenses near {targetNodeName}! Units in {targetNodeName} and its direct connections have -1 to their defense roll results this turn.", effectType: "REGIONAL_DEFENSE_DEBUFF", details: { debuffValue: 1, numTargetNodes: 1, targetCriteria: "ANY" } },
{ id: "QF004", descriptionTemplate: "Entanglement Echo! {playerName} can deploy 1 Logic Core to any friendly node with a Fabrication Hub for free.", effectType: "FREE_UNIT_DEPLOYMENT", details: { unitType: QuantumUnitType.LOGIC_CORE, quantity: 1 } },
{ id: "QF005", descriptionTemplate: "Network Instability! Connection between {nodeAName} and {nodeBName} is severed this turn.", effectType: "SEVER_CONNECTION", details: { numConnectionsToSever: 1 } },
{ id: "QF006", descriptionTemplate: "Quantum Fortification! Units defending in {targetNodeName} gain +1 defense die if attacked this turn.", effectType: "NODE_DEFENSE_BUFF_DICE", details: { diceBonus: 1, numTargetNodes: 1, targetCriteria: "ANY" } },
];

// System Prompt Templates (Full text in Section 11 of this document)
export const NC_AI1_SYSTEM_PROMPT_TEMPLATE: string = `...`; // Defined in Section 11
export const NC_AI2_SYSTEM_PROMPT_TEMPLATE: string = `...`; // Defined in Section 11

5. Game Logic (utils/noosphericConquestLogic.ts)

This file contains the core rules. Rename from utils/quantumGambitLogic.ts to utils/noosphericConquestLogic.ts.

- initialNoosphericConquestGameState(selectedTemplate?: MapTemplate): NoosphericConquestGameState:
  - Takes an optional selectedTemplate. If not provided, randomly selects one from NC_MAP_TEMPLATES.
  - Deep copies node data from the template to create the initial nodes in the game state.
  - Sets initial owner for all nodes based on ai1InitialControlledNodes, ai2InitialControlledNodes, and defaults to 'NEUTRAL'.
  - Creates initial units for AI1, AI2, and any neutral forces (from neutralKJsWithUnits, neutralNodesWithUnits) specified in the template. Ensure unique unit IDs are

generated (e.g., using generateUnitId). Units should be placed on their respective starting nodes.
    - Initializes players state (resources from NOOSPHERIC_CONQUEST_INITIAL_RESOURCES, command node ID from template).
    - Sets keyJunctionsOnMap by filtering nodes from the active template where type === 'KJ'.
    - Sets currentMapTemplateName from the chosen template.
    - Initializes turnStartTime to Date.now().
- generateUnitId(type: QuantumUnitType, owner: PlayerId, count: number): string: Helper function for unique IDs.
- processFluctuationPhase(gameState: NoosphericConquestGameState): ActiveQuantumFluctuationEvent:
    - Randomly selects an event from NC_QUANTUM_FLUCTUATION_EVENTS_POOL.
    - Resolves dynamic parts of the event description (e.g., {targetNodeName} by picking a valid node based on event.details.targetCriteria, {playerName}).
    - Returns a fully resolved ActiveQuantumFluctuationEvent object.
- applyEventEffects(gameState: NoosphericConquestGameState, event: ActiveQuantumFluctuationEvent): **NoosphericConquestGameState**:
    - Modifies gameState based on event.effectType and event.details. This requires careful, immutable updates to the state.
    - Example for RESOURCE_NODE_BONUS: Find the target node, check its owner, and if controlled by a player, increase that player's resources.
    - Example for FREE_UNIT_DEPLOYMENT: Add a new unit to gameState.units at a valid hub for the target player.
    - Example for SEVER_CONNECTION: This is tricky. You might add a temporary property to the involved nodes or store severed connections in the game state to be checked during pathfinding/attack validation for the current turn.
- processResourcePhase(gameState: NoosphericConquestGameState): **NoosphericConquestGameState**: Iterates through gameState.nodes, sums resourcesPerTurn for nodes controlled by gameState.currentPlayerId, and updates their resources.
- processDeploymentPhase(gameState: NoosphericConquestGameState, deployments: DeploymentOrder[]): **NoosphericConquestGameState**:
    - For each DeploymentOrder:
        - Validates if the player has enough resources (NC_UNIT_DEFINITIONS[order.unitType].cost * order.quantity).
        - Validates if nodes[order.nodeId] is owned by the player and hasFabricationHub.
        - If valid, subtracts cost, adds new unit(s) to gameState.units (with correct owner, nodeId, type, unique id, and initial displayOrder), and updates

players[playerId].unitsDeployed counter.

- resolveBattle(gameState: NoosphericConquestGameState, attackDeclaration: AttackDeclaration): BattleReport:
  - Retrieve all attacking units (specified by attackDeclaration.attackingUnits.map(u => u.id)) and all defending units (all units in attackDeclaration.toNodeId not owned by attacker).
  - Calculate total attack dice for attacker and defense dice for defender based on NC_UNIT_DEFINITIONS and unit types.
  - Attacker gets +1 die if attacking from a controlled Key Junction (gameState.nodes[attackDeclaration.fromNodeId].type === 'KJ').
  - Simulate dice rolls for each side: Array.from({length: numDice}, () => Math.ceil(Math.random()*6)).
  - **Shielding Node Bonus:** If defending units include an SN, for each non-SN defending unit, add +1 to its highest defense die roll *result* if that die is part of a comparison. This is applied *after* rolling.
  - **Dice Comparison:** Sort attacker and defender dice rolls from highest to lowest. Compare pairs:
    - Highest attacker vs. highest defender (with SN modifier).
    - Second highest attacker vs. second highest defender, etc.
    - For each pair: If attacker's die > defender's die, one defender casualty. Else, one attacker casualty.
  - Defender chooses casualties among their units, attacker chooses among theirs.
  - The battle can be a single round of dice rolls for simplicity in the mockup, or multiple rounds until one side is eliminated or a retreat condition is met. The BattleReport should reflect this.
  - Determine nodeCaptured status.
- processAttackPhase(gameState: NoosphericConquestGameState, attacks: AttackDeclaration[]): { updatedState: **NoosphericConquestGameState,** battleReports: BattleReport[] }:
  - For each declared attack:
    - Call resolveBattle.
    - Create an updated gameState by applying casualties (removing units from gameState.units).
    - If battleReport.nodeCaptured, update gameState.nodes[targetNodeId].owner to the attacker. Move surviving attacking units to the captured node (update their nodeId and recalculate displayOrder in both origin and destination nodes).
    - Mark all units that participated in any attack (even if they survived) as hasAttackedThisTurn = true.
    - **Crucially, after each battle resolution that results in a CN capture,**

immediately set the game to a GAME_OVER state with the correct winner.
    - Accumulate all BattleReport objects.
- processManeuverPhase(gameState: NoosphericConquestGameState, maneuvers: ManeuverOrder[]): **NoosphericConquestGameState**:
    - For each ManeuverOrder:
        - Validate: Unit exists, owned by current player, !unit.hasMovedThisTurn && !unit.hasAttackedThisTurn. Target node is adjacent and friendly-controlled.
        - If valid, update units[order.unitId].nodeId and set hasMovedThisTurn = true.
        - Recalculate displayOrder for units in origin and destination nodes.
- checkWinConditions(gameState: NoosphericConquestGameState): { winner?: PlayerId | 'DRAW', message?: string } | null:
    - This is typically called at the very end of a player's full turn (after their Maneuver phase, before switching to the next player's Fluctuation phase).
    - **Update KJ Control:** For the currentPlayerId whose turn just ended: iterate gameState.keyJunctionsOnMap. If nodes[kjId].owner === currentPlayerId, increment players[currentPlayerId].controlledKeyJunctionsTurns[kjId]. If not, reset it to 0.
    - **Check KJ Victory:** For each player, check if they control *all* nodes in gameState.keyJunctionsOnMap AND if players[playerId].controlledKeyJunctionsTurns[kjId] >= NOOSPHERIC_CONQUEST_CONSECUTIVE_KJ_CONTROL_TURNS_NEEDED for *all* those KJs.
    - **Check CN Victory:** (This is usually caught immediately in processAttackPhase, but can be a final check here). Check if nodes[players['AI1'].commandNodeId].owner === 'AI2' or vice-versa.
    - **Check Turn Limit:** If gameState.currentTurn > gameState.turnLimit:
        - Calculate influence for AI1: resources + sum(node.resourcesPerTurn * 5 for AI1-controlled nodes) + sum(unit.cost for AI1 units).
        - Calculate influence for AI2 similarly.
        - Determine winner or draw.
- **AI Prompt String Helpers**:
    - getNodeStringRepresentation(node: QuantumGambitNode, unitsInNode: QuantumUnit[]): string -> Example: N1(Type:CN,Owner:AI1,Res:3,Hub:Y,Units:[LC(AI1)x2,SN(AI1)x1],Conn:[N3,N8])
    - getBoardStringRepresentation(gameState: NoosphericConquestGameState, perspectivePlayerId: PlayerId): string -> Concatenate representations of all nodes, separated by semicolons.
    - getUnitsStringRepresentation(gameState: NoosphericConquestGameState, perspectivePlayerId: PlayerId, type: 'player' | 'opponent'): string -> Format: UnitID(Type,AtNodeID), ... For opponent, this might be filtered by visibility in a

more complex game.

6. State Management (NoosphericConquestModeContainer.tsx)

- Rename component from QuantumGambitModeContainer.
- useReducer(gameReducer, initialNoosphericConquestGameState()): This is the central piece for managing the NoosphericConquestGameState. The initialNoosphericConquestGameState function (from utils) is called here to set up the initial state, potentially with a specific map template if selected.
- **Local State (in NoosphericConquestModeContainer):**
  - isAutoPlaying: boolean (for toggling simulation speed).
  - displayedTurnTime: string (for the MM:SS turn timer).
  - selectedMapTemplateName: string (bound to the map selection dropdown, defaults to "RANDOM").
  - isInfoScreenVisible: boolean (to toggle the mode info popup).
  - ai1CoT: string, ai2CoT: string (local states to hold the latest Chain of Thought from each AI, updated via UPDATE_AI_COT action or directly in handleAIPlayerTurn).
- **Game Loop** useEffect:
  - Dependencies: gameState.currentPhase, gameState.currentPlayerId, isAutoPlaying, gameState.isBattlePopupVisible.
  - **Return/Clear:** If gameState.currentPhase === 'GAME_OVER' or !isAutoPlaying, clear autoPlayIntervalRef.current and return.
  - **Battle Popup Auto-Close:** If isAutoPlaying && gameState.isBattlePopupVisible, dispatch HIDE_BATTLE_POPUP. Then, if gameState.currentPhase === 'ATTACK', immediately dispatch ADVANCE_PHASE to move to 'MANEUVER' after a very short delay (e.g., 100-200ms) to allow the UI to react to the popup closing.
  - **Phase Processing Logic (within setTimeout for auto-play delay):**
    - **Fluctuation:**
      1. Call a utility function (e.g., selectRandomEvent from noosphericConquestLogic.ts) that returns a QuantumFluctuationEventBase.
      2. Resolve dynamic parts of the description (e.g., target node name, player name).
      3. Dispatch SET_ACTIVE_EVENT with the fully resolved ActiveQuantumFluctuationEvent.
      4. Log the event to gameState.gameLog.
      5. Dispatch APPLY_EVENT_EFFECTS_COMPLETE (this action in the reducer will actually modify the state based on the event).
      6. Dispatch ADVANCE_PHASE.
    - **Resource:**
      1. Log the action.

2. Dispatch `COLLECT_RESOURCES`.
3. Dispatch `ADVANCE_PHASE`.
- **Deployment, Attack, Maneuver:**
1. Call `handleAIPlayerTurn(gameState.currentPlayerId)`. This asynchronous function will manage the interaction with the Gemini API.
2. The `handleAIPlayerTurn` function itself will dispatch the necessary game actions (`DEPLOY_UNITS`, `DECLARE_ATTACK`, `MANEUVER_UNITS`) or `ADVANCE_PHASE` if the AI passes or an error occurs. The game loop `useEffect` doesn't need to dispatch `ADVANCE_PHASE` directly after calling `handleAIPlayerTurn` if `handleAIPlayerTurn` handles it.
- **Timing:** Use `setTimeout` to manage `autoPlayIntervalRef.current`. The delay (e.g., 1500-2000ms) allows observation.
- `handleAIPlayerTurn(playerId: PlayerId)` **async function:**
- (Detailed in Section 8) This is where the AI interaction happens. It constructs the prompt, sends it to the Gemini API, parses the response, and dispatches actions. It should also handle updating the CoT display for the current AI.
- **Turn Timer** `useEffect`:
- Depends on `gameState.turnStartTime` and `gameState.currentPhase`.
- Uses `setInterval` to calculate elapsed time from `gameState.turnStartTime` and update `displayedTurnTime` state every second.
- Clears interval on component unmount or when `gameState.turnStartTime` becomes `null` (e.g., on game over).
- **Event Handlers:**
- `handleNodeClick(nodeId: string)`: Dispatches `SELECT_NODE` action with `nodeId`.
- `toggleAutoPlay()`: Toggles `isAutoPlaying` state. If starting auto-play and game is over, dispatches `START_GAME` with the `selectedMapTemplateName`.
- `handleAdvanceManually()`:
- If `gameState.currentPhase === 'GAME_OVER'`, dispatches `START_GAME` with `selectedMapTemplateName`.
- If `gameState.isBattlePopupVisible`, dispatches `HIDE_BATTLE_POPUP` and then `ADVANCE_PHASE` (if current phase was 'ATTACK').
- Otherwise, manually triggers the logic for the current phase (similar to one step of the auto-play loop, including calling `handleAIPlayerTurn` for AI action phases).
- `handleNewGameWithSelectedMap()`: Dispatches `START_GAME` with `payload: { templateName: selectedMapTemplateName === "RANDOM" ? undefined : selectedMapTemplateName }`. If `isAutoPlaying` is true, it should probably be set to `false`.
- `toggleInfoScreen()`: Toggles `isInfoScreenVisible` state.

7. UI Components

- Rename component folder from `components/quantum_gambit/