Terrible People — Technical Specification v1.0

1. Overview

Terrible People is a real-time multiplayer web game inspired by Cards Against Humanity. Up to 4 players join a room, fill-in-the-blank prompts are read aloud, and players compete to submit the funniest (or most terrible) answer. Unfilled seats are played by bots until a human joins.

This spec is written as a Claude CLI implementation guide. Each phase is self-contained with clear acceptance criteria.

---

2. Architecture & Tech Stack

2.1 Runtime & Hosting

Layer	Technology	Notes
Hosting	**Vercel** (free tier)	Serverless functions + static assets
Framework	**Next.js 14+ (App Router)**	Server components, API routes, static pages
Language	**TypeScript**	Strict mode throughout
Styling	**Tailwind CSS**	Utility-first, minimal custom CSS

2.2 Real-Time Communication

Layer	Technology	Notes
WebSocket provider	**Pusher Channels** (free tier)	200k messages/day, 100 concurrent connections
Client library	`pusher-js`	Lightweight browser SDK
Server library	`pusher` (Node)	Trigger events from API routes

2.3 Data & State

Layer	Technology	Notes
Primary store	**Vercel KV** (Redis)	Room state, game state, player sessions

Layer	Technology	Notes
Alternative	**Upstash Redis** (free tier)	Drop-in replacement if Vercel KV is unavailable

> **Why Redis over a database?** Rooms are ephemeral — they live for 30-120 minutes then disappear. Redis TTLs handle automatic cleanup. No migrations, no schema, no persistence needed.

2.4 Key Architecture Decisions

Serverless game loop: All game logic runs in API routes (server-side). The client is a "dumb terminal" — it receives state via Pusher and sends actions via API calls. This prevents cheating and keeps logic centralized.

Single source of truth: The full game state lives in Redis under one key (`room:{roomCode}`). Every mutation MUST be atomic — never naive read → modify → write. All mutations use one of: Redis WATCH/MULTI/EXEC optimistic locking, Lua scripts, or equivalent Upstash/Vercel KV atomic transactions. This is critical in a concurrent serverless environment where multiple Lambda invocations may execute simultaneously.

Serverless-safe timing: No `setTimeout`, `sleep`, or long-running execution inside API routes. All delayed transitions (bot actions, phase pauses, reveal timers) use a **timestamp-driven state machine**. Timestamps like `phaseEndsAt` are stored in Redis. On each relevant request or heartbeat, the server evaluates whether a phase transition is due and advances state deterministically and idempotently. This pattern is reliable on Vercel's serverless runtime where functions may cold-start, duplicate, or timeout.

Idempotent game actions: All game mutation endpoints (`/api/game/start`, `/submit`, `/judge`, `/play-again`) MUST be idempotent. Repeated identical requests must not corrupt state or double-apply effects. Each action should validate current phase and whether the action has already been applied before mutating.

Game-agnostic room system: The room/lobby layer knows nothing about CAH. It manages players, readiness, and lifecycle. The game module is injected when the creator starts the game. This supports future multi-game expansion.

Connection authority: Redis heartbeat tracking is the **single source of truth** for player connection status and all game logic decisions. Pusher presence channels may be used for UI/UX indicators only and must NOT drive authoritative game state.

---

3. Data Models

3.1 Room (Redis key: `room:{roomCode}`)

```typescript
```

```typescript
interface Room {
  // Identity
  roomCode: string;         // 6-char uppercase alphanumeric (e.g., "X7KQ2M")
  createdAt: number;        // Unix timestamp

  // Lifecycle
  status: 'waiting' | 'playing' | 'finished';
  ownerId: string;          // Current room owner (transfers on leave)

  // Players (ordered by join time)
  players: Player[];        // Always length 4 (bots fill empty seats)

  // Game state (null until game starts)
  game: GameState | null;
}

interface Player {
  id: string;               // UUID generated on join
  name: string;             // Display name (entered by user)
  isBot: boolean;
  isConnected: boolean;     // Tracks active connection
  joinedAt: number;
  score: number;
}
```

3.2 Game State (CAH Module)

typescript

```ts
interface GameState {
  // Round tracking
  currentRound: number;
  targetScore: number;       // Default: 7 (first to X wins)

  // Card Czar rotation
  czarIndex: number;         // Index in players array

  // Current round
  phase: 'czar_reveal' | 'submitting' | 'judging' | 'round_result' | 'game_over';
  blackCard: BlackCard;

  // Submissions (keyed by player ID)
  submissions: Record<string, WhiteCard[]>;

  // Reveal order (shuffled player IDs — hides who submitted what during judging)
  revealOrder: string[];

  // Winner of current round
  roundWinnerId: string | null;

  // Hands (keyed by player ID)
  hands: Record<string, WhiteCard[]>;

  // Deck tracking
  blackDeck: BlackCard[];    // Remaining black cards
  whiteDeck: WhiteCard[];    // Remaining white cards
  discardWhite: WhiteCard[]; // Used white cards
}

interface BlackCard {
  id: string;
  text: string;              // Contains underscore(s) "_" as blanks
  pick: number;              // How many white cards to play (1 or 2)
}

interface WhiteCard {
  id: string;
  text: string;
}
```

3.3 Player Session (Redis key: `session:{playerId}`)

```typescript
interface PlayerSession {
  playerId: string;
  playerName: string;
  roomCode: string;
  joinedAt: number;
}
// TTL: 2 hours (auto-cleanup)
```

---

4. Room Lifecycle

4.1 Room Creation Flow

```
[Home Page] → User clicks "Create Room"
  → User enters display name
  → POST /api/rooms/create { name: string }
  → Server generates:
     - roomCode (6-char, collision-checked)
     - playerId (UUID)
     - Room object with creator as players[0], 3 bots filling seats
  → Server stores Room in Redis (TTL: 2 hours)
  → Server stores PlayerSession in Redis
  → Server creates Pusher channel: `room-{roomCode}`
  → Response: { roomCode, playerId, playerName }
  → Client redirects to /room/{roomCode}
  → Client subscribes to Pusher channel
```

4.2 Room Joining Flow

```
[Join URL: /join/{roomCode}] OR [Home Page → "Join Room" → enter code]
  → User enters display name
  → POST /api/rooms/join { roomCode, name }
  → Server validates:
     ✓ Room exists
     ✓ Room status is 'waiting'
     ✓ At least one bot seat available
     ✗ If no seats → 410 Gone ("Room is full")
```

✗ If room doesn't exist → 404 ("Room not found")
→ **ATOMIC SEAT CLAIM**:
Redis transaction: read room → check bot seats → replace first bot → write room
If transaction fails (race condition) → retry once → 410 if still no seat
→ Server stores PlayerSession
→ Server triggers Pusher event: `player-joined` on channel
→ Response: { roomCode, playerId, playerName }
→ Client redirects to /room/{roomCode}

> **Race condition handling**: The atomic Redis transaction ensures that if two players hit "join" simultaneously for the last seat, only one succeeds. The other receives a clear error with a "Return Home" button.

4.3 Room Key / Sharing

The `roomCode` (6-char alphanumeric) serves as the room key. Sharing options:

- **Direct URL**: `https://{domain}/join/{roomCode}`
- **Manual entry**: Enter code on home page

The room code expires when:

- Room is full (4 humans) → joining returns "Room is full"
- Creator leaves and no humans remain → room is destroyed
- Room TTL expires (2 hours of inactivity)

4.4 Owner Transfer & Player Departure

Player disconnects or leaves:
→ POST /api/rooms/leave { roomCode, playerId }
→ Server removes player, replaces with bot
→ If leaving player is owner:
→ Owner transfers to next human player (by join order)
→ If no humans remain → room is destroyed (Redis delete)
→ If game is in progress:
→ Player's hand is discarded back to white deck
→ Bot takes over their seat mid-game
→ If departing player was Czar → advance to next round with new Czar
→ Pusher event: `player-left` { playerId, newOwnerId? }

4.5 Disconnect Detection

- Client sends heartbeat every `HEARTBEAT_INTERVAL_MS` (10s) via `POST /api/rooms/heartbeat`
- Server updates `lastSeen` timestamp on player record
- A background check (triggered on each heartbeat from any player) marks players as disconnected if `lastSeen > DISCONNECT_TIMEOUT_MS` (30s)
- Disconnected players are replaced by bots after `BOT_REPLACEMENT_TIMEOUT_MS` (60s) of no reconnection
- Reconnection: If a player's session cookie matches a disconnected (but not yet replaced) player, they reclaim their seat

4.6 Room TTL Refresh

The room's Redis TTL (`ROOM_TTL_SECONDS`: 2 hours) is refreshed on:

- Successful heartbeat from any player
- Player join or leave
- Any game state mutation (start, submit, judge, play-again)

This prevents active rooms from expiring mid-game.

---

5. Game Loop (CAH Module)

5.1 Game Start

```
Owner clicks "Start Game" (only visible to room owner)
  → POST /api/game/start { roomCode, playerId }
  → Server validates: requester is owner, room status is 'waiting'
  → Server initializes GameState:
     - Shuffle black deck, white deck
     - Deal 10 white cards to each player (including bots)
     - Set czarIndex = 0 (owner starts as first Czar)
     - Draw first black card
     - Set phase = 'czar_reveal'
     - Set room status = 'playing'
  → Store updated Room in Redis
  → Pusher event: `game-started` { initial game state (hands are per-player, sent individually) }
```

→ After 3-second reveal pause → phase transitions to 'submitting'
→ Pusher event: `phase-changed` { phase: 'submitting' }

5.2 Round Flow

```
Phase: submitting
  ├─ Non-Czar humans see their hand → select card(s) → POST /api/game/submit
  ├─ Non-Czar bots auto-submit after random 2-5 second delay (random card selection)
  ├─ Czar sees "Waiting for answers..."
  ├─ As each player submits → Pusher: `player-submitted` { playerId } (no card reveal)
  └─ When all non-Czar players have submitted → transition to 'judging'

Phase: judging
  ├─ All submissions revealed (shuffled order, anonymous)
  ├─ Czar (human) picks the winner → POST /api/game/judge
  ├─ Czar (bot) picks randomly after 3-second delay
  └─ Winner selected → transition to 'round_result'

Phase: round_result
  ├─ Winning submission revealed with player name
  ├─ Winner's score incremented
  ├─ Check win condition:
  │   ├─ If score >= targetScore → transition to 'game_over'
  │   └─ Else → after 5-second display pause:
  │       ├─ Discard all played white cards
  │       ├─ Deal replacement cards (back to 10 per hand)
  │       ├─ Advance czarIndex (wraps around)
  │       ├─ Draw new black card
  │       └─ Transition to 'czar_reveal' → new round
  └─ Pusher event: `round-result` { winnerId, winnerName, score, isGameOver }

Phase: game_over
  ├─ Final scoreboard displayed
  ├─ Owner sees "Play Again" button (reshuffles and restarts)
  └─ All players see "Leave Room" button
```

5.3 Bot Behavior (v1 — Random)

Bots are simple in v1:

- **As player**: Select random card(s) from hand after randomized delay
- **As Czar**: Pick random submission after delay

- **Names**: Pre-defined bot names: "Bot Alice", "Bot Bob", "Bot Charlie"

Serverless-safe bot timing: Bot actions do NOT use `setTimeout` or `sleep`. Instead:

- When a phase begins that requires bot action, a `botActionAt` timestamp is stored in Redis (current time + random delay from `BOT_SUBMIT_DELAY_RANGE_MS`)
- On each heartbeat or API call, the server checks: has `botActionAt` passed? If yes, execute the bot action and advance state
- Bot actions are idempotent — if already executed (checked via submission/judgment records), the check is a no-op

Mid-game bot takeover: When a human player is replaced by a bot mid-game:

- Bot inherits the player's current hand, score, and seat position
- If the player had not yet submitted this round, the bot submits after a randomized delay (using timestamp pattern above)
- If the player was Czar, the bot assumes Czar duties immediately and picks randomly after delay

> **Future (v2)**: Bot personality system — each bot has a humor profile that weights card selection toward certain themes (dark, absurd, wholesome). Czar bots have a preference profile that weights their judging.

---

6. Pusher Event Schema

6.1 Channel Naming

- Room channel: `presence-room-{roomCode}` (presence channel for connection tracking)

6.2 Events

Event	Payload	Triggered When
`player-joined`	`{ player: Player }`	Human joins room
`player-left`	`{ playerId, newOwnerId?, replacementBot: Player }`	Human leaves
`game-started`	`{ gameState (sanitized) }`	Owner starts game
`phase-changed`	`{ phase, blackCard?, czarId? }`	Phase transitions

Event	Payload	Triggered When
`player-submitted`	`{ playerId }`	Player submits cards (no card data)
`submissions-revealed`	`{ submissions: { id, cards }[] }`	All submissions in, anonymous reveal
`round-result`	`{ winnerId, winnerName, submission, scores }`	Czar picks winner
`game-over`	`{ finalScores, winnerId, winnerName }`	Someone hits target score
`hand-updated`	`{ hand: WhiteCard[] }`	Private event (per-player channel)
`room-destroyed`	`{}`	Room is shut down
`player-disconnected`	`{ playerId }`	Heartbeat timeout
`player-reconnected`	`{ playerId }`	Player reclaims seat

6.3 Private Channels

- Player-specific channel: `private-player-{playerId}`
- Used for: `hand-updated` (so players can't see each other's cards)

---

7. API Routes

All routes are Next.js API routes under `/app/api/`.

7.1 Room Management

Method	Route	Body	Returns
POST	`/api/rooms/create`	`{ name }`	`{ roomCode, playerId }`
POST	`/api/rooms/join`	`{ roomCode, name }`	`{ playerId }`
POST	`/api/rooms/leave`	`{ roomCode, playerId }`	`{ success }`
GET	`/api/rooms/{roomCode}`	—	`{ room (sanitized) }`
POST	`/api/rooms/heartbeat`	`{ roomCode, playerId }`	`{ success }`

7.2 Game Actions

Method	Route	Body	Returns
POST	`/api/game/start`	`{ roomCode, playerId }`	`{ success }`
POST	`/api/game/submit`	`{ roomCode, playerId, cardIds[] }`	`{ success }`
POST	`/api/game/judge`	`{ roomCode, playerId, winnerId }`	`{ success }`
POST	`/api/game/play-again`	`{ roomCode, playerId }`	`{ success }`

7.3 Auth Strategy

No accounts. Player identity is maintained via:

1. `playerId` stored in a **cookie** (httpOnly, sameSite: lax)
2. Matched against `session:{playerId}` in Redis
3. Every API call validates: does this playerId belong to this room?

---

8. Page Structure

```
/                   → Home page (Create Room / Join Room)
/room/[roomCode]    → Lobby + Game view (single page, state-driven)
/join/[roomCode]    → Join page (enter name → redirects to /room)
```

8.1 Home Page (`/`)

- App title and tagline
- Two buttons: "Create Room" / "Join Room"
- Join Room: text input for room code
- Both flows → name entry modal/step → redirect

8.2 Lobby View (`/room/[roomCode]` — status: waiting)

- Room code displayed prominently (with copy button)
- Shareable link (with copy button)
- Player list (4 slots, bots shown differently from humans)

- "Start Game" button (owner only)
- "Leave Room" button

8.3 Game View (`/room/[roomCode]` — status: playing)

- **Top bar**: Round number, scores, current Czar indicator
- **Center**: Black card (prominent)
- **Submissions area**:
  - During `submitting`: progress indicators ("3/3 submitted")
  - During `judging`: anonymous cards revealed, Czar selects
  - During `round_result`: winning card highlighted with player name
- **Bottom**: Player's hand (white cards), selectable
- **Player strip**: Avatars/names with scores, Czar crown icon

8.4 Game Over View

- Final scoreboard (ranked)
- "Play Again" button (owner)
- "Leave Room" button

---

9. Card Data (Starter Set)

For the build and testing phase, a minimal card set:

- **40 black cards** (prompts) — mix of pick-1 and pick-2
- **200 white cards** (answers)

Cards are stored as a static JSON file in the repository (`/data/cards.json`). This is loaded server-side when a game starts. No database needed for cards.

```typescript
```

```
// /data/cards.json structure
{
  "black": [
    { "id": "b001", "text": "What's the secret ingredient in grandma's famous soup?", "pick": 1 },
    { "id": "b002", "text": "Step 1: _. Step 2: _. Step 3: Profit.", "pick": 2 }
  ],
  "white": [
    { "id": "w001", "text": "A questionable life choice" },
    { "id": "w002", "text": "Accidentally replying all" }
  ]
}
```

> **Phase 3 (Data Import)**: Expand to full custom deck. Potentially support user-uploaded card packs in future.

---

10. Implementation Phases (Claude CLI)

Phase 1: Infrastructure & Room System

Goal: A working room that players can create, join, and leave with real-time updates.

1. Initialize Next.js project with TypeScript, Tailwind
2. Set up Vercel KV (Redis) connection
3. Set up Pusher (server + client)
4. Implement Room data model and Redis operations (create, read, update, delete with TTL)
5. Build API routes: create, join, leave, heartbeat
6. Build home page (create/join flow with name entry)
7. Build lobby page (player list, room code display, copy link, start button, leave button)
8. Wire up Pusher events for lobby (player-joined, player-left)
9. Implement owner transfer logic
10. Implement race condition handling for concurrent joins

Acceptance Criteria:

- [ ] User can create a room and see a 6-character room code
- [ ] Second user can join via URL or code entry
- [ ] Both users see real-time player list updates
- [ ] Room shows bots in unfilled seats

- [ ] When creator leaves, ownership transfers to next human
- [ ] Two users joining the last seat simultaneously: one succeeds, one gets error
- [ ] Room auto-expires after 2 hours of inactivity

Phase 2: Game Engine (CAH Module)

Goal: A fully playable game of CAH with bots and humans.

1. Create starter card set (JSON file — 40 black, 200 white)
2. Implement GameState initialization (shuffle, deal, set Czar)
3. Build game start API route
4. Implement round phases (czar_reveal → submitting → judging → round_result)
5. Build card submission API route with validation
6. Build judging API route
7. Implement bot behavior (random card selection/judging with delays)
8. Implement hand replenishment after each round
9. Implement Czar rotation
10. Implement win condition check (first to target score)
11. Implement play-again functionality
12. Build game UI (black card display, hand, submissions, judging, scoreboard)
13. Wire up all Pusher events for game state
14. Handle mid-game player departure (bot takeover)

Acceptance Criteria:

- [ ] Owner can start game from lobby
- [ ] Players see their private hand (10 cards)
- [ ] Players can submit cards for pick-1 and pick-2 prompts
- [ ] Submissions are anonymous during judging phase
- [ ] Czar can select a winner
- [ ] Bots auto-play with random selections after realistic delays
- [ ] Scores update correctly after each round
- [ ] Game ends when a player reaches target score
- [ ] Mid-game player departure handled gracefully (bot replacement)
- [ ] "Play Again" reshuffles and restarts

Phase 3: UI/UX Polish & Bot Personality

Goal: Make it feel good. Smooth transitions, clear feedback, bot personality.

1. Phase transition animations (card reveals, submissions)
2. Timer/countdown indicators for bot delays and phase transitions
3. Toast notifications (player joined, player left, round winner)
4. Mobile-responsive layout optimization
5. Connection status indicator
6. Reconnection flow (reclaim seat after disconnect)
7. Bot personality system (humor profiles, weighted selection)
8. Loading states and skeleton screens
9. Error state handling and recovery
10. Sound effects (optional, mutable)

Phase 4: Data Import & Card Expansion

Goal: Full card library with potential for custom decks.

1. Curate full original card set (500+ white, 100+ black — all original content, not CAH IP)
2. Card category/tag system for future filtering
3. Potential: user-created card packs (stored in Redis with room TTL)

---

11. Environment Variables

```
env

# Vercel KV / Upstash Redis
KV_REST_API_URL=
KV_REST_API_TOKEN=

# Pusher
PUSHER_APP_ID=
PUSHER_KEY=
PUSHER_SECRET=
PUSHER_CLUSTER=
NEXT_PUBLIC_PUSHER_KEY=
NEXT_PUBLIC_PUSHER_CLUSTER=

# App
NEXT_PUBLIC_BASE_URL=       # e.g., https://terrible-people.vercel.app
```

12. Project Structure

```
/app
  /page.tsx                → Home page
  /room/[roomCode]/page.tsx  → Lobby + Game (state-driven)
  /join/[roomCode]/page.tsx  → Join flow
  /api
    /rooms
      /create/route.ts
      /join/route.ts
      /leave/route.ts
      /heartbeat/route.ts
      /[roomCode]/route.ts
    /game
      /start/route.ts
      /submit/route.ts
      /judge/route.ts
      /play-again/route.ts
    /pusher
      /auth/route.ts        → Pusher channel auth for presence/private channels

/lib
  /redis.ts              → Redis client + room CRUD operations (all mutations atomic)
  /pusher.ts             → Pusher server + client config
  /game-engine.ts          → CAH game logic (pure functions, no side effects)
  /bots.ts               → Bot behavior logic (timestamp-driven, idempotent)
  /types.ts              → All TypeScript interfaces
  /constants.ts            → All game config constants (see below)
  /errors.ts             → Standardized error response helpers
  /utils.ts              → Room code generation, shuffling, etc.
```

12.1 Core Constants (`/lib/constants.ts`)

```typescript
```

```
export const HAND_SIZE = 10;
export const MAX_PLAYERS = 4;
export const DEFAULT_TARGET_SCORE = 7;
export const BOT_SUBMIT_DELAY_RANGE_MS = [2000, 5000] as const;
export const BOT_JUDGE_DELAY_MS = 3000;
export const CZAR_REVEAL_DURATION_MS = 3000;
export const ROUND_RESULT_DURATION_MS = 5000;
export const HEARTBEAT_INTERVAL_MS = 10000;
export const DISCONNECT_TIMEOUT_MS = 30000;
export const BOT_REPLACEMENT_TIMEOUT_MS = 60000;
export const ROOM_TTL_SECONDS = 7200; // 2 hours
```

12.2 Standardized Error Responses

All API error responses follow a consistent shape:

```
typescript

interface ApiError {
  error: string;    // Human-readable message
  code: string;     // Machine-readable code (e.g., "ROOM_FULL", "GAME_IN_PROGRESS")
}

// Example codes:
// ROOM_NOT_FOUND, ROOM_FULL, GAME_IN_PROGRESS, NOT_OWNER,
// INVALID_PHASE, ALREADY_SUBMITTED, UNAUTHORIZED, RACE_CONDITION
```

```
/data
/cards.json              → Card library

/components
/Home.tsx                → Home page UI
/Lobby.tsx               → Lobby view
/Game.tsx                → Game view (parent)
/BlackCard.tsx           → Black card display
/WhiteCard.tsx           → White card (in hand + in submissions)
/PlayerList.tsx          → Player strip with scores
/Hand.tsx                → Player's hand of white cards
/Submissions.tsx         → Anonymous submission display + judging
/Scoreboard.tsx          → End-game scoreboard
/RoomCode.tsx            → Room code + copy button
/NameEntry.tsx           → Name input modal/step
```

---

13. Edge Cases & Error Handling

Scenario	Handling
Room doesn't exist	404 → "Room not found" with home button
Room is full	410 → "Room is full" with home button
Room is already playing	403 → "Game in progress" with home button
Simultaneous join (last seat)	Atomic Redis transaction, loser gets 410
Player submits after phase ends	409 → Ignore, client state will catch up via Pusher
Czar disconnects during judging	Bot takes over Czar role, picks randomly
All humans leave	Room destroyed
Card deck runs out	Reshuffle discard pile back into deck
Invalid playerId / roomCode	401 → Redirect to home
Pusher connection lost	Client shows "Reconnecting..." banner, auto-retries
Browser tab closed	Heartbeat stops → 30s disconnect detection → 60s bot replacement

---

14. Future Considerations (Out of Scope for v1)

- **Multi-game framework**: Room system is already game-agnostic. Add game selection to room creation.
- **Player customization**: Avatars, colors, emotes
- **Room settings**: Target score, timer per turn, NSFW card toggle
- **Spectator mode**: Watch without playing
- **Chat**: In-game text chat
- **Native apps**: iOS/Android wrappers once web app is proven
- **Custom card packs**: User-created content
- **Persistent stats**: Win rates, favorite cards (requires accounts)
- **Player count flexibility**: 2-8 player support per game module