ChatGPT

Deep Research Report: Building a Multi-Agent Team Experiment Platform

Original AI Team App – Purpose and Design Overview

The existing **AI Team App** is a Node.js web application built to simulate a team decision-making exercise with one human participant and multiple AI agents [1] [2]. In the original setup, a human (recruited via Prolific) joins an online chat with **three AI teammates**, forming a four-person team. The team's task (drawn from team research literature) is to discuss and **rank three potential locations for a new restaurant** based on ten predefined criteria [3]. Each AI agent and the human have **unique pieces of information** about the locations (a classic “hidden profile” design), so they must share and combine knowledge to identify the best location. Key characteristics of the original app include:

- **Simulated Team Scenario:** The chat is a role-play where all members appear human. The AI agents are instructed to behave like real people (informal tone, occasional typos) and never reveal they are AI [4] [5]. The human participant is **unaware which teammates are bots** (one research question is whether they can detect AI). All members must agree on a ranking before declaring the task complete.

- **Agent Personas and Unique Info:** The three AI agents each have a persona (e.g. outgoing “Master of Motivation”, analytical “Logic Luminary”) and a set of unique clues about the locations. For example, one agent “James” might know that *East Point Mall* has plenty of parking but is under 2,000 sqft, etc [6] [7], while another agent knows different criteria outcomes for the sites. The human participant also has a unique info sheet (displayed in the UI) with some yes/no facts for each location. **No single member has all the data**, so the team must pool information to evaluate which site meets the most criteria.

- **Turn-Taking Logic:** The backend uses OpenAI’s GPT-4 API to generate agents’ messages. It doesn’t simply have all bots speak at once; instead, it includes logic for **deciding which agent speaks next**. For each new user message, the server asks GPT-4 via a special prompt whether each agent should “participate” at that turn [8] [9]. Only agents that get a “YES” will respond, usually one at a time, to mimic a natural conversation. This prevents the three AI from overwhelming the chat simultaneously. The agents also occasionally initiate conversation (e.g. a random agent introduces the team if the human hasn’t spoken) to ensure a realistic flow.

- **Prompt Engineering for Realism:** The system uses extensive prompt templates to guide AI behavior. There are system-level instructions telling each agent how to act human: e.g. “pretend to be a Prolific survey-taker, keep messages under 250 characters, make occasional typos, don’t be too formal” [10] [11]. Agents are explicitly instructed to **not reveal they are AI** and to only signal completion by saying *“task-complete”* when the group has agreed on a ranking [12] [13]. Each agent’s prompt also injects their **private knowledge and strategy**. For instance, an agent named James gets a prompt listing the criteria outcomes he knows (e.g. “East Point Mall – 50 parking spaces: Yes;

>2000 sqft: No; ...") and a strategy like *"remind teammates to count the yes's and no's"* [14] [15]. This elaborate prompting ensures the AI responses are diverse and aligned with their roles.

- **Web Interface:** The front-end is a simple HTML/JS client served by the Node.js backend. It includes a chat window and side panels for the participant's "Unique Information" table and task instructions. Initially, the participant must enter a name (and a fun "badge title" if in the public condition) and choose an avatar. They then see a chat interface with a "Raise Hand to Participate" button. In the original UI, the message input is disabled until the user clicks **Raise Hand**, simulating a scenario where the human must "join" or signal readiness to chat [16]. Once they send a message, the UI hides the input again until the next opportunity to speak, enforcing a turn-based interaction. There's also a checkbox for "Task Completed?" (though in the code this mostly just turns the label green and triggers a completion check) and a notes scratchpad for the participant's personal use [17] [18].

- **Experimental Conditions:** The app was designed to run different experimental conditions. For example, on session start the backend randomly assigns a **team identity condition**: Team Race **A** vs **B**, which simply swaps the AI teammates' names between two sets (e.g. "Maurice, Ebony, Trevon" vs "James, Sophia, Ethan") [2] [19]. This was likely to study effects of teammates' implied demographics. Another condition was the participant's **self-disclosure**: "public" condition means the user and bots share a personal title (badge) during introductions, whereas "private" means no badges. The welcome screen randomly set `self_cond` to *public, private,* or *none* [20] and the chat prompts changed accordingly (e.g. in public chats, bots were told to explain their badge/title in their first message [21] [22]). These built-in factors demonstrate the app's experimental focus. However, they were hardcoded; adding new conditions (like different AI models or task types) required code changes.

- **Data Logging:** The Node backend records each session's data for research. It creates a session entry with the participant's Prolific ID, condition, team assignment, etc., and appends it to a CSV file [23] [24]. Chat transcripts are saved to text files as the conversation progresses [25] [26]. This ensures researchers can later analyze messages, counts, response lengths, "raise hand" usage, and who/what signaled task completion.

Limitations: The early prototype, while innovative, has several limitations that we will address in the new design:

- *Single-Human Sessions:* The original supports exactly one human participant per session (plus AI bots). It does **not** handle sessions with multiple real humans. All participants beyond the one user were AI agents. For broader research, we need the ability to run sessions with **multiple human participants** (e.g. 4 humans in one team, or 2 humans + 2 AIs, etc.). The current architecture (which relies on a single browser client driving the session) cannot do that.

- *Limited Real-Time Interaction:* The front-end does not use true real-time networking (no WebSocket connections). It relies on the single client polling for agent responses and simulating delays. For multi-human teams, a robust real-time communication mechanism is needed so that messages broadcast to every client instantly. The prototype would not scale to, say, 3 separate users on different computers in the same chat.

- *Hardcoded Task and Prompts:* The entire restaurant-location task (criteria list, agent info, and specialized prompts) is baked into the code [3] [6]. This makes it difficult for a non-technical user to repurpose the app for a new scenario. Ideally, the platform should allow **configuring different team tasks** (e.g. a brainstorming task, a puzzle, or any scenario from team research literature) without rewriting the code. Likewise, agent personas and unique info should be data-driven or modular, so researchers can easily plug in new content (or new AI prompt strategies).

- *Deployment and Usability:* The original app requires running a Node server and providing API keys via AWS SSM – steps that may be challenging for non-technical researchers. There is no one-click deployment; someone has to set up the server environment. Also, to let remote participants (e.g. on Prolific) join, the host needs to deploy the server on a public URL or use a tunnel. These manual steps can be error-prone. The UI itself, while functional, could be more polished and intuitive (e.g. the "raise hand" mechanism might confuse users). In short, **ease of use** for experimenters and participants needs improvement.

In summary, the original AI Team App demonstrated a proof-of-concept for human–AI team experiments. It successfully created an interactive chat where bots and a person collaborate on a decision task, and it collected useful data. However, to support the user's **bigger vision of a general platform for team experiments**, we need to generalize and modernize this design.

Vision for the New Multi-Agent Experiment Platform

The goal is to create a **flexible, user-friendly platform** that enables researchers to run a variety of team-based experiments involving humans and AI agents. We will preserve what worked in the original (the idea of human-AI collaboration, and the hidden-profile style task as an example) but greatly expand the capabilities. Key objectives for the new platform:

- **Multiple Configurations of Humans and AIs:** Unlike the old app's fixed "3 AI + 1 human" setup, the new system should allow any mix of humans and AI in a team. For example:
- *All-human teams* (e.g. four real people in a chat, with no AI at all).
- *Hybrid teams* (e.g. 2 humans + 2 AI, or 1 human + 3 AI, etc., as experimental conditions demand).

- *All-AI teams* (multiple AI agents chatting with each other autonomously). This would let researchers test different AI models against each other in team scenarios (for instance, three different LLMs discussing which one performs best, or simulation of team dynamics purely among bots).

- **Support for Different Team Sizes and Tasks:** The platform should not be limited to four-person teams or the specific restaurant-ranking task. Researchers might want to simulate larger teams (5, 6 members) or smaller groups (dyads, triads). They may also plug in different **team tasks** – for example, a cooperative creative task, a problem-solving puzzle, or a negotiation scenario. The system must be flexible to accommodate new task instructions and role information. We plan to achieve this by making the task scenario **data-driven** (external configuration) rather than hardcoded. The "restaurant location selection" will become one built-in scenario template that can be cloned or modified, and additional scenarios can be added.

- **Easy Experiment Setup with Variable Conditions:** We anticipate researchers will use this platform to test hypotheses like *"Do people perform better in teams augmented with AI?"*, *"Can people tell if their* *teammates are bots?"*, *"How does team composition (all human vs mixed) affect outcomes?"*, or *"Which LLM model makes a better team member?"*. To facilitate rigorous testing, the platform should allow **experimental manipulations** such as:

- Changing which AI model powers the agents (e.g. compare GPT-4 vs another model in otherwise identical team setups).
- Toggling agent behaviors or visibility (e.g. whether AI teammates are presented as AI or pretending to be human).
- Varying any aspect of the scenario (instructions, available information, time limits, etc.).

We will make it straightforward to define **conditions** and have the system randomly assign or systematically rotate through them, much like the original did with `self_cond` and `team_race` [19] [20]. This could be configured via a simple file or admin interface – for example, an experiment config might specify two conditions "Model A" vs "Model B", and the platform will use one LLM for agents in condition A and a different one in B. The user's request specifically mentioned manipulating the LLM model as a condition; our design will accommodate that and any similar toggles.

- **Researcher-Friendly Operation:** The primary users (experimenters) may not be engineers, so we must prioritize **ease of use**:
- *Simple Deployment:* Provide a clear, minimal set of steps to get the app running. This might involve a pre-built Docker image or a one-command launch, so a researcher can host the app on their machine (localhost) or a server without deep technical know-how. We will also consider options for one-click cloud hosting (more on this under Deployment).
- *GUI-Based Control:* Wherever possible, avoid requiring the researcher to edit code. Instead, configurations should be done through UI forms or config files with documentation. For instance, the researcher could upload a CSV or JSON defining a new task scenario (roles and info) or use a web form to input experiment parameters (team size, number of humans vs AIs, etc.). The platform can then generate the sessions accordingly.

- *Stability and Maintainability:* We will use modern, well-supported frameworks for both front-end and back-end. This reduces the chance of technical issues and makes future updates easier. The user mentioned using "the most modern tools" that work well with LLMs – for example, using the latest OpenAI API features, robust error handling and retries (the prototype used axios-retry for API calls [27]), and possibly libraries to manage prompt templates. The emphasis will be on a **robust backend** that reliably handles the chat logic and data logging, since a broken backend would ruin an experiment. The front-end should be polished enough for participant use, but complex visual embellishments are lower priority than core functionality.

- **Extensibility:** Build the system in a modular way so that new features can be added by future developers or AI agents. This means clearly separating components (e.g. agent logic, communication layer, task config, UI) and providing interfaces or APIs for them. For example, if later one wants to integrate a new type of agent (say, a reinforcement learning agent or a different API), it should be possible without rewriting everything. Or if one wants to incorporate voice chat instead of text in some variant, a modular design would allow swapping the communication mode. In short, we should not hardcode assumptions that limit the platform's use cases.

In concrete terms, the **new repository** will effectively be a generalized version of the AI Team App – a platform for **multi-user, multi-agent experiments**. It will allow a researcher to spin up a temporary website where invited participants (or bots) can join a team chat, complete a task, and produce data for analysis. The rest of this report outlines how to achieve this, focusing on system architecture, component design, and implementation strategies.

Proposed System Architecture

To meet the above goals, we propose a client-server web application with a real-time communication backbone. The architecture will consist of:

- A **backend server** (responsible for managing sessions, handling AI agent logic, and relaying messages).
- A **frontend web client** (the user interface for participants, delivered via browser).
- A **data store** for logging experimental data (could be simple files as before, or a database for more complex usage).
- Optionally, an **admin interface or config files** that researchers use to set up experiments (this could be as simple as editing a YAML file, or a small web admin page in the future).

We will now discuss each part in detail.

Backend Design and Logic

Technology choice: We can build the backend either with Node.js (as the original) or using Python. Both have strong libraries for web sockets and HTTP. Given that the original app is Node and already interfaces with OpenAI APIs, sticking with **Node.js + Express** is reasonable – we can modernize it with TypeScript for safety if needed, and use **Socket.io** for real-time communication. Node's event-driven nature fits well for handling chat messages and multiple simultaneous sessions. On the other hand, a Python backend (e.g. using FastAPI or Flask + SocketIO) could leverage Python's ecosystem (like integration with ML/AI libraries or LangChain if we ever use those). Both are viable; for now, we'll assume Node/Express with socket.io for concreteness, as it's familiar from the current code and easy to deploy. The key is that whichever stack, the server will provide **REST APIs for setup** and use **WebSockets for the chat**.

Session and Room Management: The server will need to handle multiple concurrent sessions (teams) potentially. We will implement a session manager that can create a new **chat room** with a unique ID (similar to the `conversationId` UUID in the original [28]). Each session will have associated metadata: the configured task scenario, which agents (and models) are in it, which humans are assigned, the condition, etc. When participants join, they are added to a session room (via socket.io or equivalent), so they receive messages broadcast to that room. If the experiment requires waiting for multiple humans, the server can hold new users in a "lobby" until the required number have joined, then start the chat when ready. For example, if a session expects 4 humans, the first person to arrive might see a "Waiting for 3 more participants..." screen until the room fills, then everyone gets switched to the active chat. We can use a **join code** or link to associate participants with the correct session. The researcher might generate these codes in advance or let the system do it dynamically. (This addresses the user's need for having a code/password so that only specific people join a given session.)

Real-time message handling: Using WebSockets, the server can instantly relay messages. The flow in a mixed human-AI session would be: 1. A human participant sends a chat message via the client. This is emitted to the server through the socket connection. 2. The server receives the message and records it (append to the session transcript and possibly call the save-to-file function, similar to the original which saved each message with role tags [25]). 3. The server then needs to determine AI responses. In a turn-based scheme, we might have the server trigger the AI agents one by one. For instance, after a human message, the server can invoke each agent's logic in sequence or according to some rule (e.g. the old approach of asking GPT who should respond first). We could simplify the first version by cycling through agents in a fixed order or have all agents attempt to respond but only allow one at a time. A refined approach is to replicate the **"decideParticipation" logic** using a small LLM prompt or heuristic: effectively asking "Who should speak next?" based on the last message [8] [29]. This adds realism but also API overhead. As a simpler starting point, we might implement a rule like: *if a human spoke, have exactly one AI respond (and perhaps alternate which AI speaks to ensure all participate)*. We can incorporate randomness or checking for direct mentions (e.g. if the user's message contains an agent's name, that agent should reply). These rules can be iterated on. The important part is the server will **call the LLM API** to generate the agent's reply when it's that agent's turn to talk. 4. The LLM API call will use a prompt constructed from the session state (conversation history, agent's persona, and scenario instructions) – essentially an improved version of the `callOpenAI` prompt logic in the original [30] [31]. We'll externalize the prompt template for maintainability. After getting the AI's message, the server emits that message to all clients in the room via the socket, so every participant sees it appear in the chat in real time. 5. The server continues this process, possibly allowing multiple back-and-forth messages from different agents until a stopping condition is met (e.g. the group reaches consensus or a time limit).

For **all-human sessions**, the server's job is simpler: it just relays messages between clients (and logs them), since no AI logic is needed. It will still monitor for an experiment end condition.

AI Agent Module: We will design an **Agent module/class** that encapsulates each AI bot's configuration and behaviors. An agent definition might include: - A name (e.g. "James"). - The model to use (e.g. `gpt-4` or `gpt-3.5-turbo` or even an open-source model endpoint). - A persona/profile with two parts: - **Public identity** (what the participants see, like a role or title and maybe an avatar). - **Private prompt data** (the instructions we feed into the LLM, like the agent's personality, unique info, strategy hints as in the original agentInformation [15] [32]). - Methods for deciding actions: e.g. `agent.shouldRespond(lastMessage)` which could be a heuristic or LLM-based as mentioned, and `agent.generateResponse(conversationHistory)` which calls the LLM API with the appropriate prompt and returns a message.

By encapsulating this, adding a new agent or switching an agent's model is just a matter of adjusting its config. For example, to test different models together, one agent's config can use `gpt-4` and another could use `gpt-3.5`, and both can coexist – the server will call the respective APIs appropriately.

LLM Integration: Initially, we will integrate OpenAI's API (since the user prefers starting with OpenAI models). The backend will need an API key (we can allow this to be set via an environment variable or a simple config file so non-tech users can just paste their key). We'll use the latest version of the Chat Completion API, and take advantage of any new features (like function calling if ever needed, or improved token usage). The prompt content from the original app can largely be reused and simplified – for instance, the long system prompt giving task instructions and persona can be loaded from a template file for each scenario. We will also keep the useful techniques: - Slight **random variations** to make conversations less repetitive (the original randomly chose between two model IDs for GPT-4 [33], presumably to add variety, and sometimes rewrote messages for human-like style [34] [35]). We can continue to include some randomness, like a small chance to inject a typo or use a different phrasing model. However, we should ensure consistency if needed for fairness between conditions. - **Task completion detection:** The original used an LLM call to evaluate if a message said "task-complete" or not [36] [37]. We can simplify this by just checking if the content equals the trigger phrase, but the LLM evaluation also checked if a message was relevant or not (possibly to filter incoherent AI output). In our new version, we might implement a simpler rule-based check for the "task-complete" keyword to decide when agents mark the task finished. We will still allow an AI or a human to signal completion. Once the platform sees a completion signal, it can either end immediately or require confirmation (e.g. all users check a box, or at least one AI and the human agree). This aspect can be configured per experiment.

- **Handling All-AI Simulations:** In the case of sessions that have no humans (for AI-vs-AI experiments), the backend can either run entirely autonomously or with minimal supervision. Essentially, the server would initialize a session and immediately start the agents' conversation (since no human input is needed). It might have them take turns generating messages until a stopping condition. This could be done in a loop server-side: call agent1 -> broadcast -> call agent2 -> broadcast, etc. Alternatively, we could still run it through the socket infrastructure by having a special client that represents an AI (though that's not necessary). This mode will be useful for rapidly testing different model interactions or performing many trials (we could even run in a headless mode without a front-end, outputting logs).

- **Data Storage:** For simplicity and transparency, we can continue logging to CSV or JSON files as the original did. Each session will log participant IDs, condition, and outcome metrics (e.g. how many messages, time taken, etc.) in a summary file, plus a detailed chat transcript. One improvement is to include final results (like the final agreed ranking, if applicable) or any post-survey responses if we add that. If the platform grows, we could move to a small database (SQLite or similar) to store sessions and messages, but a well-structured CSV per session and a summary CSV may suffice and is easier for many researchers to handle. We will also generate **completion codes** for participants upon finishing – e.g. a random alphanumeric code shown on the end screen, which they can input back into Prolific to confirm their participation. The backend can produce this code when the session completes and include it in the session data (the original code had a `finishCode` field prepared in the CSV header [38]).

- **Scalability considerations:** If a researcher wants to run many sessions in parallel (for example, 10 teams of 4 at the same time), the backend should handle it as long as the machine and API limits allow. Node.js with socket.io can handle multiple rooms easily. We just need to ensure the OpenAI API calls are throttled or queued so as not to exceed rate limits (the axios-retry usage indicates we will incorporate retry/backoff [27]). In case of heavy usage, we might add rudimentary load balancing or instruct the user to use multiple API keys. But for initial versions, we assume moderate usage (like a few sessions at a time).

Frontend Design and User Interface

Overview: The frontend will be a **web application (HTML/CSS/JS)** that participants can access via a link. We will likely implement it as a single-page application or a set of pages that guide the participant from start to finish of the experiment. Key screens include: 1. **Welcome/Instructions Page:** Explains the study and collects initial info (e.g. an entry code or Prolific ID). Here the participant might enter a code that the researcher gave them, which identifies which session to join. If sessions are pre-created, the code maps to a specific team; or if not, entering a valid code can trigger session creation. In the original, the welcome page took Prolific ID and immediately called `/start-chat` on Next [39] [40] – we can do something similar, but possibly wait to actually join the chat until after the user configures their profile. 2. **Profile Setup Page:** The participant can choose a display name (or it might be fixed as “Participant” for anonymity), and maybe an avatar image or color. In the old app, the user provided a first name and (if in public condition) a “badge” title describing themselves [41] [42]. We can keep this feature optional, as it personalizes the experience. Non-technical researchers might enjoy the ability to have participants create a fun title or role. The profile setup can be combined with the welcome page for simplicity – e.g. a form that asks for name, (optional badge), and shows a grid of avatar icons to pick (like the original did by loading avatars from the server [43] [44]). This info will be stored (likely in the server session and localStorage as before) and used to display the participant in the chat.

1. **Waiting Room (if needed):** If the experiment requires multiple humans to start together, a waiting lobby is crucial. For example, if 4 humans are needed but the participant arrives early, they’ll see a message like “Waiting for other participants to join…”. The frontend will show a status and maybe how many have joined so far (if we want to be informative). The server via socket can broadcast updates (e.g. “2 of 4 participants are now connected”). Once the required number is met, the session can automatically begin for everyone (perhaps the screen transitions to the chat). If the session type doesn’t require waiting (like only one human needed or it’s an all-AI run), this step is skipped or very brief.

2. **Chat Interface:** This is the core UI where the team interaction happens. We will model it after the original chat interface, with improvements:

3. A chat transcript area where each message is shown with the speaker’s name (and possibly avatar or badge). For instance, messages might be prefixed like “**James (Master of Motivation):** [text]” for an agent (the old client did this for the first introduction message [45]), and for the participant maybe “**Alice (Participant):** [text]” or simply their name.
4. Different styling for different speakers (e.g. different bubble colors or labels) to help distinguish AI agents from the user. However, if the study is about humans not knowing who is AI, we will make sure the interface does **not** explicitly label bots as AI. They will just appear as teammates with names.
5. A text input box for the participant to type messages. Unlike the original, we probably won’t hide the input after each send – that mechanism was tied to the “raise hand” feature. Instead, we can allow a more fluid chat unless we specifically want to enforce turn-taking. We might simplify by letting the participant send messages freely, and controlling agent replies with a slight delay so it still alternates naturally. If needed, we could implement a “send disabled while agents are typing” to prevent message overlap, but perhaps not necessary if agents respond quickly.
6. Buttons or indicators for special actions: e.g. a “Raise Hand” button could still be included if we simulate moderated discussion, but we may drop it for simplicity unless an experiment specifically requires it. More universally useful might be a **“Done” button or checkbox** that the participant can click when they believe the task is completed. This would be the human equivalent of an agent saying "task-complete". For example, in an all-human team, once they’ve agreed on an outcome, one of them can check “Task Completed” which notifies the server. We will incorporate this so that sessions can end even without an AI to trigger the finish. The front-end can ask for confirmation (“Are you sure your team has finished the task?”) to avoid premature endings.
7. Information panels: as in the original, we should display the task instructions and the participant’s unique information within the UI so they don’t have to remember everything. A tab or sidebar can

show **“Task Instructions”** (the goal, criteria, etc.) and **“Your Information”** (the clues or data that participant has). In the prototype, they had tabs for “Chat”, “Unique Information”, “Help”, and “Notes” [46] [47]. We can simplify to perhaps two main tabs: **Chat** and **Info/Help**. The Info tab can combine the task description and the participant’s data, or separate them if needed. The Help could just be a restatement of directions and maybe reminder to be active. The Notes pad is an optional feature – it’s a nice addition for lab studies (participants take private notes), but for an online study it might be underused. We can include it if it’s easy, but it’s not essential.

8. Participant identity display: perhaps show the participant’s chosen name and avatar at the top of chat, along with a list of team members. For instance, a sidebar can list each team member’s name (and badge if public) with their avatar. This was partly implemented in the original (they had a `displayTeamMembers()` function to show all agent names and the participant’s name in a sidebar panel [48]). We can use that idea so the participant always knows who is on their team (which helps when reading chat messages).

9. **Live feedback** like typing indicators: socket.io can easily broadcast when someone is typing. We could show “Alice is typing…” or simulate “James is typing…” when an AI is formulating a response (the original code did have a typing indicator for the randomly chosen intro agent [49]). These little touches improve realism. We just have to ensure they don’t reveal who is AI (so an AI typing indicator should look the same as a human one).

10. **End of Experiment Page:** When the session is complete (either through reaching the task goal or a timeout), the UI should transition to an **End page**. This page can thank the participant, display any **survey questions** or debriefing text if applicable, and crucially, display the **completion code** for the study. We will generate a unique code on the backend for each participant or session (e.g. `finishCode` saved in the CSV [50]) and send it to the client. The client then shows “Your completion code is: XYZ123”. If multiple humans were in the session, each should get the same code or one tied to their ID, depending on how the researcher wants it (likely a common code per session for ease if everyone was in one Prolific submission). The End page can also summarize what happened (maybe showing the final agreed ranking outcome, etc.) if the researcher wants to give feedback, or instruct them to close the window. The original had an `end.html` and `end.js` likely handling these tasks (including writing final data like avatar chosen, etc.) – we will implement similar functionality cleanly.

Responsive and Accessible Design: We should ensure the UI is clear and not too cluttered. Using modern CSS frameworks or styles will help make it responsive (in case participants use different screen sizes). We might not target mobile explicitly (many online experiments require desktop), but it won’t hurt to have a mobile-friendly layout if possible. Text should be legible and the interface should be intuitive: the participants should know where to read info and where to type. We will also add obvious indicators when new messages come in (scrolling, or maybe a sound alert if desired, though that could be optional).

Front-end Implementation: We can use plain HTML/JS (with possibly a library like Vue or React if a more structured approach is needed). Given that non-technical users won’t modify the front-end code, we can choose a framework for developer convenience. A lightweight option is to use a minimal framework or even just jQuery for DOM updates, since the UI isn’t extremely complex. However, using a reactive framework like **React** or **Vue** might make it easier to manage state (participants list, chat messages array, etc.). It also opens up the possibility of a nicer development experience. On the other hand, integrating a build toolchain might be a burden for deployment. A compromise could be using something like **Vue.js** with CDN (so we can write Vue components without a build step) or using plain JavaScript with templating.

We will also integrate the front-end with the back-end via **WebSockets** for the chat. For example, using socket.io on the client side (which simplifies receiving broadcast messages and sending emits). During profile/waiting stage, we might use AJAX (REST API calls) for things like starting a session or checking lobby status, but once in chat, sockets will handle the real-time flow.

Ease of use for researcher on front-end: The researcher mostly cares about participants' experience here. But one thing to note: if a researcher wants to run a local session themselves (for example, to test with all AI or with colleagues in one room), they should be able to open multiple browser windows and join the same session easily (like by using the session code). We'll make sure the front-end supports that (maybe allow passing a session code in the URL for convenience). Also, we may want to provide a **observer view or control** – for instance, a researcher might want to watch the chat live without being a participant, or manually end a session if it's running too long. This could be a later addition (admin console), but worth keeping in mind. Initially, a researcher can always see the logs or even join as a "hidden" user via the console, but that's advanced.

Configuration of Tasks and Roles

A crucial aspect of the new design is making scenarios configurable. We propose to organize the experiment definition in a structured format. For example, we might have a directory like `scenarios/` in the repository, containing one folder or file per scenario. A scenario config could be a **JSON or YAML file** specifying:

- **Task Description:** A human-readable description of the task and goal, which will be shown to participants (e.g. the instructions about ranking locations, or whatever the new task is).
- **Criteria or Task Data:** If the task involves specific data (like the table of criteria in the restaurant task), we can include that. For instance, a list of criteria and which ones each location meets (the Y/N matrix) can be encoded. In the original, those were implicitly in the agent prompts and participant info table [51] [52]. We can explicitly represent them so the system can generate the participant's info view dynamically and ensure consistency with what agents know.
- **Roles Configuration:** This defines each role in the team, which could include:
- Role name (for AI agents, this might double as the persona name; for human participants, we might just call them "Participant" or allow them to choose a name).
- For AI roles: their persona prompt, including any unique info. We could either list the raw lines they know or reference some dataset. For example, we might store something like:

```
agents:
  - name: James
    model: openai:gpt-4
    persona: |
      You are extraverted and confident... (personality description)
    knowledge:
      East Point Mall:
        Parking: Yes
        Size: No
        FootTraffic: Yes
        Tourist: No
        Student: Yes
```

```
          WasteDisposal: Yes
          Employees: Yes
        Starlight Valley:
          Parking: Yes
          Student: No
          WasteDisposal: Yes
          Employees: No
        Cape James Beach:
          Parking: No
          Competitors: Yes
          Tourist: Yes
          Student: No
          WasteDisposal: No
          Employees: Yes
      strategy: "Remind teammates to focus on count of yes vs no...
(strategy hint)"
    - name: Sophia
      model: openai:gpt-4
      persona: "You are agreeable and friendly..."
      knowledge:
        ... etc.
```

This is just illustrative – the idea is the unique info that was hardcoded in prompts [14] [53] becomes data here. The system can then format it into the agent's prompt at runtime (inserting the facts into a template phrase "HERE IS YOUR UNIQUE INFO..."). This separation means a researcher could edit the YAML to change the scenario without touching code.

- For human roles: if we imagine scenarios with multiple human participants having different info (it could happen, e.g. 2 humans and 2 AIs where even the humans have different clues), we'd also allow roles for humans with their own info to display. In the original, only one human existed and they had a fixed info set (which was essentially the remaining pieces of the puzzle not given to bots). In a general case, we could have, say, three humans each with certain clues – we should support that. The config might specify a role as "HumanA", "HumanB" with associated info to show. When actual people join, the server can assign them to those human roles. This is complex but offers great flexibility for future use (e.g. simulate an experiment where each human is given a different document to read and then chat – the platform could handle distributing those documents).

- **Turn-taking / Rules settings:** Possibly include scenario-specific parameters like "max messages before forcing completion" or "require explicit 'task-complete'". For instance, the original scenario had rules: don't rank until 15 messages have passed [54] [55], and end when someone says "task-complete". Another scenario might have a time limit or require a specific output. We can incorporate fields for these: e.g. `completionTrigger: "phrase:task-complete"` or `autoEndAfterMinutes: 10`. Initially, we might hardcode these in prompts or code, but structuring them will make the platform adaptable.

- **Conditions/Variations:** We could allow multiple variants within one scenario definition. For example, two different lists of agent names (like Team Race A/B) or toggles like `useBadges: true/false`. This could be under a "conditions" section. However, it might be simpler to treat each condition as a separate scenario file or separate experiment config that references one scenario. We can decide on the implementation – perhaps an experiment config file that says: scenario = X, and condition1: use names set A, condition2: use names set B, etc. To start, we might implement conditions similarly to how the original did: random assignment in code, but with values drawn from the scenario config (for example, if `teamNamesSets` is defined in scenario, randomly pick one set).

Loading and using the config: The backend on startup can load scenario definitions (or if there's only one main scenario configured, just load that). When a session is created (via an API call or starting experiment via UI), the server will pick the appropriate scenario and condition, then instantiate the agents and prepare any human info for display. The front-end, upon session start, can query the server for **"what is my unique info and task instructions"** and display them dynamically instead of relying on hardcoded HTML as before [51]. This way, changing the task in config automatically updates what participants see.

By designing this configuration system, we make the platform **extensible**. For example, to add a new use-case (say a leadership emergence simulation with AI personas), the researcher or developer can add a new YAML file with the roles and prompts for that scenario. The core engine remains the same.

Deployment and Hosting Considerations

One of the user's concerns is making the platform easy to deploy for short-term studies, even by those with limited tech skills. We will address this by providing multiple deployment options and thorough documentation:

- **Local Machine (Offline) Usage:** The simplest way to run an experiment might be on a single computer in a lab setting (or the researcher's laptop). We will make sure the app can be started with minimal fuss. For example, if using Node, just `npm install` and `npm start` (after putting the OpenAI API key in a config). The app would then be accessible at `http://localhost:3000`. Participants could either use that same machine or, if on the same network, possibly connect via the host's IP. For truly offline personal testing, this is fine. We will write clear instructions in the README for this case.

- **Using Tunneling or Temporary Cloud URL:** If a researcher wants remote participants (e.g. from Prolific) to connect to their locally running app, we can recommend using a tunneling service like **ngrok** or **LocalTunnel**. These can expose `localhost:3000` to a public URL (for a short time) with minimal steps. We will document how to do this (install ngrok, run `ngrok http 3000`, etc.). This approach requires the researcher's machine to stay on and connected, which is okay for a one-day experiment with a small number of participants. It avoids the complexity of deploying to a proper server. We will caution about the limitations (ngrok free tier might have limited concurrent connections, etc.) but it is a quick solution for pilot studies.

- **Cloud/VPS Deployment:** For a more robust setup (e.g. running a multi-day study with dozens of participants), the researcher might use a cloud service. We can suggest relatively user-friendly platforms:

- **Heroku or Render:** These can deploy Node apps easily. However, the free Heroku no longer exists, and these might require adding a websocket addon or configuring the Procfile. We can provide a sample Heroku deployment guide (since the app is Node, a one-click deploy might be possible). But note: Heroku dynos sleep, which is not good for an ongoing study, so probably a paid tier or short runtime.

- **DigitalOcean Droplet / AWS Lightsail:** These are simple virtual servers. We can offer a step-by-step: spin up a Node.js droplet, git clone the repo, configure env, and run via PM2 or Docker. It's more technical, but we can try to simplify.
- **Docker Container:** We can provide a Dockerfile that bundles the app. Then a researcher with Docker knowledge (or their institution's IT) could run the container on any host. This also helps with consistent environment (no "works on my machine" problems). The Docker image could potentially be published for easier pulling.
- **Vercel / Netlify:** These are geared toward front-end or serverless. Vercel, for example, could host the static front-end and serverless functions for API, but sustaining a websocket chat in a serverless environment is tricky. It's not impossible (one can use 3rd-party real-time services), but that adds complexity. Given the real-time, continuous nature of chat, a persistent server process is preferable. So we might not recommend Vercel for the backend. However, we could host just the client on Vercel/Netlify and point it to a separate backend if we split them. That said, bundling front and back together on a simple server is easiest for researchers to manage.

Considering the above, a straightforward path is: **set up on a cloud VM**. For user-friendliness, perhaps we could automate some of it or at least provide detailed guidance. For instance, we could share a pre-configured AWS AMI or a DigitalOcean 1-Click App image with everything installed. If that's feasible, a researcher could just instantiate that image, which might be the easiest but requires us to maintain an image. If not, clear instructions with screenshots on how to deploy would be included in documentation.

- **Security & Access Control:** Since experiments often need to be closed to only invited participants, we should include basic security:
- At minimum, not publicly listing active sessions. The join should be by knowing a session code or having a specific link. We can embed the session ID or code in the URL we give participants (e.g. `experiment.com/join?code=XYZ123`). The server can validate that and put them in the right session. If the code is invalid or already used to capacity, show an error.
- Possibly a simple password gate if needed (the researcher could set a global study password that participants are told to enter).

- We should also handle **API key security** – the OpenAI key will reside on the server (never exposed to clients) and ideally not in the repo. We expect the researcher to provide their own API key. This can be via an environment variable or a small config file not tracked in Git. The original used AWS Parameter Store for keys [56], which is overkill for local usage. We will keep it simple (e.g. a `.env` file) and instruct users to keep it private.

- **Reliability:** We will implement measures to avoid crashes and data loss:

- Ensure unhandled promise rejections (like API failures) are caught and result in either a retry or a graceful error message to the user ("Agent failed to respond, please wait..." etc.) rather than crashing the server.
- Possibly auto-save transcripts frequently so if something goes wrong mid-chat, partial data is still recorded.
- Implement timeouts: e.g. if an AI response takes too long (the OpenAI API might occasionally stall), we may timeout after e.g. 30 seconds and either retry or notify that agent is unresponsive. This prevents the participants from waiting indefinitely.

- We might include a **manual override** for the researcher – for example, a special command or button to terminate a session early if needed (in case of a bug or participant dropout). This could just be in a console for now.

- **Documentation and Guidance:** Provide a **comprehensive README/user guide** with step-by-step instructions for:

- Installing and running the app locally.
- Setting up an experiment (how to edit config files for a new task or use the default scenario).
- How participants join (with examples of sharing a link or code).
- How to interpret the output data.
- Options for deployment (from easiest to more advanced, as discussed).

Our aim is that a researcher can literally clone the repo and get a basic experiment running in minutes, using our default scenario as a demo (the restaurant task with 3 AI vs 1 human). Then they should feel comfortable adapting it to their own needs, gradually, without diving into the core code unless necessary.

Implementation Guidelines and Feature Roadmap

Finally, we outline some guidelines and phased implementation approach for the development of this new platform (which may be carried out by an AI coding assistant or a development team):

- **Phase 1: Core Chat Functionality with One Scenario.** Start by setting up the basic client-server communication and reproducing the original app's primary functionality in a cleaner way:
- Implement the backend with session management, a simple broadcast of messages, and integration with OpenAI for generating agent replies. Initially, target the same 3-agent + 1-human scenario (restaurant ranking) to ensure we have parity with the original. This means importing the prompt logic and unique info from the old code [6] [3] into our new structure (e.g. as templates/config).
- Implement a basic front-end that allows one human to chat with three bots on localhost. Focus on making sure messages flow correctly: the human sends -> server -> AI responds -> back to client. We can test this with just one browser window to mimic the original behavior.
- Log data to console or files to verify that we capture everything (Prolific ID, the dialogue, etc.).

- We might not use WebSockets in the very first run (to reduce complexity, could do polling like original), but since multi-user is a goal, it's better to integrate socket.io from the start. It's not too hard: set up socket endpoints for join room, new message, etc.

- **Phase 2: Expand to Multi-User and Configurability.** Once the single-user version works, introduce:

- **Multi-client support:** e.g. allow two browser windows to join the same session. This involves the waiting room logic and ensuring both see messages. We can simulate with two different test users on localhost. Gradually increase to N clients.
- **Multiple roles config:** Adjust the backend to not assume a fixed number of AIs or humans. Instead, read from a scenario config how many agents should be spawned and how many human spots are expected. Then the waiting room knows when to launch. This may involve writing the scenario config file and a loader, and altering how we instantiate agents. Test with variations (like a config for 4 humans, and a config for 4 AIs).

- **Dynamic content delivery:** Instead of hardcoding the participant's info table in HTML, send it from backend based on scenario. For example, when a participant joins, the server can send them the list of criteria they know (so the front-end can fill the table or list). This ensures consistency and allows new tasks to be plugged in.

- Keep the UI simple but functional for multiple users. At this stage, multiple humans should be able to chat together (with or without bots) – a major milestone that unlocks experiments like human-only teams or mixed teams with more than one human.

- **Phase 3: Usability Enhancements and Testing.**

- Refine the prompts and agent logic: see if the AI responses are coherent and the conversation ends properly. Adjust the strategy prompts or add a bit more instructions if agents are not performing as expected (e.g. if they end too early or never end, tweak the end detection).
- Add features like the "task completed" button for humans, and ensure that triggers the session wrap-up on the backend (perhaps all sockets receive a "session finished" event and then navigate to End page).
- Polish the UI: add avatars, color-coding, etc., and make sure it's clear who is who. Also ensure the interface is **robust against refreshes or reconnects** – e.g. if a participant accidentally refreshes their browser, can they rejoin the session seamlessly? Using the stored session ID, we could allow that.
- Conduct dry-run tests: simulate an entire experiment scenario with colleagues or pilot users. Try different conditions (all-human, all-AI) to see if the system behaves as intended in each case. This will likely surface edge cases (like overlapping messages, or one AI not speaking at all, etc.), which we can then fix.

- Implement any missing pieces from the original feature set that are still relevant: e.g. the raise-hand mechanism (if we want it), or the "badge introduction" for public condition (so that if self_cond=public, the agent's first message includes an explanation of their badge [21] [22]). We should verify that the conditions logic works by actually running a public vs private condition and observing differences (like whether the badge name is mentioned or not).

- **Phase 4: Documentation and Deployment Aids.**

- Write detailed guides as mentioned, and perhaps prepare example configuration files for a couple of scenarios (e.g. the given restaurant task and maybe a dummy second task to illustrate how to add one).
- Containerize the app with Docker and test running it that way.
- If possible, set up a minimal cloud demo to ensure the instructions are correct.

- Create a repository structure that is clean and inviting – e.g. separate `frontend` and `backend` directories as before, but with clearer organization and maybe a single config folder. This will help new users navigate the project.

- **Future Extensions (beyond initial scope):** Once the basic platform is solid, many exciting features can be added:

- Integration of alternative LLMs (Anthropic Claude, AI21, open-source models via Hugging Face, etc.) to give researchers more options.
- A GUI for the researcher to configure an experiment (rather than editing YAML, a form where they input parameters and upload info sheets).
- More complex team interactions like supporting audio/video chat or embedding collaborative tools (though that's far off and depends on research needs).
- Analytical tools: e.g. automatically compute some metrics from the chat (who sent more messages, sentiment, etc.) for the researcher's convenience.
- Better moderation: if using in uncontrolled environments, features like profanity filters or a way to remove a problematic participant from a session might be needed.

In conclusion, this new repository will generalize and modernize the AI Team App into a **platform for multi-agent team experiments**. It will enable researchers to easily run studies where any number of humans and AI models work together (or compete) on various tasks. By focusing on a solid back-end with real-time capabilities and a flexible scenario configuration, and by improving user-friendliness in deployment and interface, we will create a tool that can facilitate cutting-edge research on human-AI collaboration, team dynamics, and comparative model performance. This platform will allow experiments such as: - *Can people tell they are working only with bots versus with humans?* (by toggling team composition) - *Are teams more effective when they include AI agents?* (by measuring task outcomes across conditions) - *How do different AI models behave in group settings?* (by letting different LLMs converse and observing emergent behavior like leadership or conflict) - *What team structures yield the best decisions?* (all human vs mixed, or varying the number of AI assistants)

By implementing the plan above, we ensure the system is **versatile** for all these questions, while remaining **accessible** to the researchers who formulate them. The end result will be a research platform that can potentially contribute to high-impact findings in social science and AI, enabling publications in venues like *Science* or *Nature* as the user aspires, and generally advancing our understanding of human-AI teaming in a rigorous experimental manner.

Sources:

- Original AI team app code (Node/Express backend and static frontend) [57] [58] [6] [3], which provided the basis for scenario and agent behavior design.
- Observations from the prototype's front-end implementation [51] [45] and logic for agent decision-making [8] [59], which informed our improvements in the new architecture.
- Logging and data schema from the original (session CSV, finish code, etc.) [23] [38] that we plan to preserve and extend in the new platform.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [19] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [50] [53] [54] [55] [56] [57] [58] [59] app.js
https://github.com/actonbp/ai_team_app/blob/39b63215889a68b291711b9d9ee45a702d454625/backend/app.js

[16] [45] [48] [49] chat.js
https://github.com/actonbp/ai_team_app/blob/39b63215889a68b291711b9d9ee45a702d454625/frontend/js/chat.js

[17] [18] [46] [47] [51] [52] chat.html
https://github.com/actonbp/ai_team_app/blob/39b63215889a68b291711b9d9ee45a702d454625/frontend/chat.html

20 39 40 welcome.js

https://github.com/actonbp/ai_team_app/blob/39b63215889a68b291711b9d9ee45a702d454625/frontend/js/welcome.js

41 42 43 44 login.js

https://github.com/actonbp/ai_team_app/blob/39b63215889a68b291711b9d9ee45a702d454625/frontend/js/login.js