ChatGPT

# Intelligent Multi-Agent Dashboard Interface

## Project Overview

This project is a **modular AI agent simulation dashboard** combining interactive visualization, multi-agent logic, and AI language/speech capabilities. It provides a **force-directed WebGL graph** of intelligent agents (e.g. *Phi5 Mo-Tsu*, *Octopuce AI*, *Phøebii Zeta*), each agent having an internal neural state (perceptrons, entropy metrics, symbolic attributes). Agents evolve over time, form clusters (including "twin" pair bonds), and exhibit emergent swarm behaviors. A dynamic **control panel** allows real-time tuning of simulation parameters – neural weight adjustments, clustering sensitivity, entropy thresholds, aesthetic modes (color schemes, shapes, sound mappings) – and toggling between **symbolic vs. binary vs. network** visualization modes.

On the backend, an **Express.js server** integrates with the OpenAI SDK to imbue each agent with text and speech intelligence. Agents can receive multimodal input (text or voice) and respond using OpenAI's API (e.g. GPT-4 for text generation, optionally text-to-speech for voice). The server exposes both **RESTful endpoints and WebSocket streams** for real-time interaction and monitoring. Agents collaborate through an evolving **"neural logic engine"** – a coordination layer inspired by swarm intelligence and semantic-symbolic reasoning (blending logical rules with creative language output).

The system supports **exporting simulation data** (e.g. JSON state dumps, SVG snapshots of the graph, logs of neural states) and provides a live monitoring panel for metrics like global entropy, agent status, and emergent pattern logs. All components are packaged as a standalone Node.js app with server-rendered HTML (no heavy frontend framework), Bootstrap 5 UI, and emphasis on extensibility – new agent types, behaviors, or visualization modules can be added with minimal changes.

## Architecture and Components

**Frontend (Visualization & UI):** A single-page dashboard (`index.html`) hosts a WebGL `<canvas>` for rendering the agent network, and an overlay UI panel (HTML/CSS via Bootstrap) for controls and info. The visualization uses a custom **force-directed graph engine** in JavaScript to simulate agents as nodes with spring-like interactions (repulsion, centering) [1] [2]. Each agent is drawn as a point sprite via WebGL shaders, with attributes (position, size, color) passed to the GPU each frame. Shaders ensure efficient rendering of many agents; for example, the fragment shader draws each agent as a glowing circle (fading alpha at edges) rather than a square pixel point [3] [4]. The UI panel (a fixed `<div>` on the left) contains form controls: - Sliders for numerical parameters (e.g. global neural weight factor, cluster detection threshold, entropy cutoff). - Toggle switches for boolean settings (e.g. enable/disable "Entropy influence", "Binary mode", "Symbolic labels", "Mimicry behavior") [5]. - Buttons for actions (e.g. *Add Agent*, *Export JSON*, *Export SVG snapshot*, *Start Voice Input*). - A text input for sending a message to selected agents (for chat/command input). - Sections to display current agent list (each as a badge you can click to select) and real-time metrics/logs.

**Backend (Server & AI Integration):** The Node/Express server (`server.js`) serves the static files and manages all AI and persistence logic. Key responsibilities: - Serving the main dashboard page (server-side rendered HTML or a static file) and static assets (JS, CSS, shader files). - Providing a **REST API** (under `/api/...`) for controlling the simulation and agents. For example, endpoints to list agents, add or

remove an agent, adjust settings, or send a message to an agent. We follow patterns from earlier agent systems (e.g. start/stop endpoints for agents like **AlphaEvolveAgent** and **SigmaAlgeaSwarm** [6]). In this app: - `GET /api/agents` – returns JSON of all agents' state (for external monitoring). - `POST /api/agents` – create a new agent (with parameters in the request body). - `GET /api/agents/:id` – get state of a specific agent. - `POST /api/agents/:id/message` – send a text message to a given agent and get its AI-generated response. - `GET /api/export` – export the entire simulation state (e.g. as JSON file download). *(Additional endpoints can be added for other controls as needed, e.g. toggling global simulation state or getting logs.)* - Managing a **WebSocket server** (using Socket.io or the native WebSocket module) for real-time updates. The server broadcasts events to clients (e.g. *agent-spoke*, *agent-twin-formed*, *metric-update*) so the UI can update immediately. It also allows clients to push events like user voice input streaming or real-time control changes. This duplex channel is ideal for low-latency interactions (e.g. streaming an agent's response token-by-token, or synchronizing multiple dashboard instances). - **OpenAI Integration:** The server uses OpenAI's Node SDK to give agents natural language capabilities. Each agent can have a distinct persona or prompt context. When a message is sent to an agent (via UI or API), the server creates a chat completion request (e.g. using GPT-4 or GPT-3.5) including the agent's persona and conversation history. The OpenAI API key is stored in `.env` and loaded via the `openai` library. The response (agent's reply text) is returned to the client (and optionally broadcast via WebSocket so all connected UIs see it). For speech, if the user sends audio, the server (or client) can use a speech-to-text service (OpenAI's Whisper or browser API) to transcribe it before generating a reply. The client can use Web Speech API for text-to-speech to speak out the agent's reply voice. This way, each agent effectively becomes a chatbot with a voice, accessible through the dashboard or external API. *(The `/api/agents/:id/message` endpoint handles text; for audio input, one could stream audio via WebSocket or send a file to a separate endpoint.)* - **Collaboration Engine:** Beyond individual Q&A, the server can orchestrate multi-agent reasoning sessions. Inspired by swarm intelligence, the **neural logic engine** can prompt multiple agents to collectively solve a problem. For example, one agent could generate a hypothesis, others critique or elaborate, and a final agent synthesizes a conclusion. This can be implemented by sequential OpenAI calls with system prompts assigning roles to each agent (e.g. "You are agent A, respond with a plan…", then "You are agent B, given A's plan, respond with critique…", etc.). The orchestrator code can cycle through agents, akin to a chain-of-thought mediator. (This pattern is similar to having planner/coder/tester roles working in concert [7] [8], but here agents might produce *semantic* outputs like stories or solutions, and a symbolic logic module could extract structured data from these outputs to update the knowledge base.) The result of collective reasoning can be sent back into the simulation – e.g. if agents reach consensus on forming a cluster or taking an action, the simulation can reflect that (joining certain agents together, increasing a "agreement" metric, etc.). This feature is extensible and experimental, demonstrating how semantic (language) and symbolic (structured rules/values) layers interact: the system might parse agent conversations to symbolic facts or use an agent's entropy (a measure of uncertainty/creativity in its responses) as input to the shader visuals (as seen in the HyperGuardian demo where entropy influenced a shader's wave frequency [9]).

**Data Model:** Each agent has a structured state, including: - **Identity**: a unique ID and a name (e.g. "Phi5 Mo-Tsu"). Optionally a type/category to differentiate agent classes (for extensibility). - **Visual properties**: position (x,y in the force graph), velocity (vx, vy for simulation), color, size (radius). Many of these derive from other attributes – e.g. size could map to entropy value, color could represent agent type or recent sentiment. - **Neural state**: a simple neural network or perceptron. We include a basic **Perceptron** class for each agent as a stand-in for a "brain" that can learn a binary classification or decision boundary [10]. (In practice this could be extended to a small multi-layer network or more complex model, but a perceptron suffices to demonstrate evolving weights.) The perceptron can be used for internal decision-making or pattern recognition; for example, an agent might use it to decide if another agent is a twin/ally (1) or not (0) based on their attributes. - **Entropy value**: a numeric measure of randomness or uncertainty in the agent's state. This could be computed from the agent's recent actions or

communications (e.g. Shannon entropy of its last message's text [11] or a running entropy of its perceptron weights). In our demo, for simplicity, we might treat entropy as a random value that drifts over time or is influenced by interactions. Entropy is used both in logic (e.g. highly unpredictable agents might explore more connections) and in visualization (e.g. node color/size) [12]. - **Symbolic characteristics**: a set of descriptors or a knowledge base the agent holds. This could be keywords, a role description (e.g. "poet", "analyst"), or a small list of facts (like a mini knowledge graph). Symbolic data isn't heavily used in the basic simulation physics, but it's crucial for the semantic layer (e.g. an agent with `persona: "Philosopher"` might respond in poetic forms, while one with `persona: "Scientist"` responds with factual tone). These symbolic tags can also be used for clustering (agents with similar tags gravitate together) and for **symbolic visualization mode** (e.g. display an icon or abbreviation on each node). - **Relationships**: the agent can track pointers to others, e.g. a reference to its *twin* (if it forms a twin pair with another agent) or membership in a *meta-cluster*. Twins might be formed when two agents achieve a very high similarity in state (e.g. minimal distance in the graph plus similar entropy or same type). We can represent a twin link as a mutual property `agent.twinId` on both agents. Clusters can be identified by grouping algorithms (community detection) or simple heuristics (distance threshold); an agent may have a `clusterId` to denote it's part of a cluster.

All agent states are kept in memory (and optionally could be persisted to a database like MongoDB as in earlier prototypes [13] [14], but for simplicity this sandbox uses in-memory state with export capability instead of a full DB).

**Simulation Loop:** The front-end runs an animation loop (`requestAnimationFrame`) to animate the force-directed graph. On each tick: - **Physics Update:** We compute repulsive forces between every pair of agents (O(n^2) for simplicity, but acceptable for moderate n). Each pair exerts a small repulsion to keep nodes from overlapping [1]. We also apply a centering force that pulls each agent slightly toward the origin (to keep the cloud of agents roughly in view) [15]. Velocities are damped to simulate friction. - **Agent Behavior Update:** After basic physics, each agent can perform internal updates. For example, an agent might adjust its perceptron weights or entropy slightly (simulating "learning" or internal dynamics each frame). If *evolution* is toggled on, agents can introduce random perturbations to their internal state to simulate ongoing change [16]. Agents also execute social behaviors like **mimicry**: if *mimicry* is enabled, an agent looks at its neighbors and slowly adjusts its color or other traits toward those of the closest neighbor [17] [18]. This causes nearby agents to become more alike over time, reinforcing clusters (an emergent behavior where groups of agents adopt a common color/trait). Another behavior is **twinning**: if two agents come very close and share similar attributes (e.g. their difference in entropy or perceptron output falls below a threshold), we designate them as twins – they could then synchronize some of their state (for instance, copy each other's perceptron weights or share a unique highlight color to show they are paired). - **Rendering:** After updating positions and states, the code prepares data for rendering. We fill a GPU buffer with each agent's current position, size and color. For example, we map an agent's properties to visual attributes: perhaps red channel corresponds to entropy (higher entropy = more red) and green channel to another metric (like perceptron output or cluster id). In our shader, blue is fixed just to give a base hue. Each agent is drawn as a point; the vertex shader uses the provided `aPos` (position) and computes gl_PointSize based on `aSize` (which we set equal to the agent's radius) [19]. We pass a projection scale uniform to map simulation coordinates to clip space; here we treat simulation coords as roughly normalized already, so `uScale` is small (on the order of 2/width) to convert to clip-space [19]. The fragment shader checks `gl_PointCoord` (the relative coordinate within the drawn point sprite) to draw a circle (it discards fragments outside a radius) and uses the passed in color [3] [4]. The result is a nicely rendered node for each agent. *(In a future extension, we could render connecting lines between agent twins or cluster members. This could be done by either drawing GL.LINES between certain nodes or by a second rendering pass. For simplicity, this initial version focuses on nodes; cluster relationships can be visually inferred via color/grouping.)*

**Extensibility:** The system is designed to be modular: - Adding a new agent type: Just define a new agent blueprint with its unique properties (e.g. new color mapping, different perceptron size or a custom update function) and instantiate it via the UI or API. The rendering and simulation will automatically include it, since all agents adhere to the same interface. - New behaviors: The `ForceGraph`/simulation class can be extended with new interaction rules (e.g. attraction between certain agent types, or special forces for twins). This could be toggled via new UI switches. - Symbolic overlays: We can easily overlay information on the graph. For example, in *symbolic mode*, the code can display each agent's name or an icon. This can be done by either using HTML elements positioned over the canvas (since we can translate an agent's world position to screen coordinates) or by rendering text in WebGL (more involved). Similarly, a "binary mode" might show a matrix or binary patterns – one could integrate a secondary visualization (like a small `<canvas>` for a matrix of an agent's perceptron weights or ASCII histogram of agent's last message). The code structure keeps these concerns separated (simulation vs. UI vs. rendering), so new modules can plug in. - The OpenAI-driven abilities are abstracted so you could plug in a different AI service or extend agents with new skills (e.g. image recognition if multimodal inputs expand, or connecting agents to external knowledge bases). The public API makes it possible to script the simulation externally (for example, a Python script could create agents via REST calls or listen on the WebSocket feed to analyze the simulation in real-time).

With this high-level picture in mind, let's break down the code structure and key files in the project.

## Code Structure and Files

```
ai-agent-dashboard/
├── package.json            # Node.js project manifest (dependencies,
scripts)
├── server.js               # Express server setup (serves UI, handles API &
WebSocket, OpenAI integration)
├── public/                 # Static frontend files (served by Express)
│   ├── index.html          # Main HTML page containing UI layout and canvas
│   ├── style.css           # CSS for layout and customization (Bootstrap
overrides, etc.)
│   └── js/
│       └── app.js          # Main client-side JavaScript (simulation logic,
rendering, event handlers)
└── shaders/                # (Optional) separate shader files if not inlined
in JS
    ├── agent.vert.glsl     # Vertex shader for agent points
    └── agent.frag.glsl     # Fragment shader for agent points
```

**package.json:** Lists required dependencies such as Express, OpenAI, and Socket.io. For example:

```
{
  "name": "ai-agent-dashboard",
  "version": "1.0.0",
  "description": "Multimodal AI agent simulation dashboard",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
```

```
  },
  "dependencies": {
    "express": "^4.18.2",
    "openai": "^3.2.1",
    "socket.io": "^4.6.1",
    "dotenv": "^16.3.1"
  }
}
```

*(Versions are examples; the actual latest version can be used.)* We include **dotenv** to load configuration from a `.env` file (for secrets like `OPENAI_API_KEY`).

### Server Implementation (`server.js`)

Below is the code for the Express server, which serves the front-end and provides the API and WebSocket. We've annotated key sections with comments for clarity:

```
// server.js
require('dotenv').config();  // Load .env variables (OPENAI_API_KEY, etc)
const express = require('express');
const path = require('path');
const http = require('http');
const { Configuration, OpenAIApi } = require('openai');
const { Server: SocketIOServer } = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = new SocketIOServer(server);  // WebSocket server

const PORT = process.env.PORT || 3000;

// In-memory storage for agents (simple representation)
let agents = [];  // array of agent objects (defined similarly as front-end,
minus functions)

// Utility: find agent by ID
function findAgent(id) {
  return agents.find(a => a.id === id);
}

// Serve static files from /public
app.use(express.static(path.join(__dirname, 'public')));

// Parse JSON request bodies for API usage
app.use(express.json());

////////////////////////////////////////////
// Express Routes (REST API)
////////////////////////////////////////////
```

```
// List all agents and their core state
app.get('/api/agents', (req, res) => {
  res.json(agents);
});

// Create a new agent (with optional type or parameters in req.body)
app.post('/api/agents', (req, res) => {
  const data = req.body || {};
  // Basic agent template
  const newAgent = {
    id: `agent-${Date.now().toString(36)}`,
    name: data.name || `Agent${agents.length+1}`,
    type: data.type || 'generic',
    entropy: data.entropy || Math.random() * 5,      // random initial
entropy (0-5 range)
    color: data.color || null,  // will be set by client if null (color
picking logic on front-end)
    perceptron: { weights: [] },// placeholder, actual Perceptron exists
client-side
    twinId: null,
    clusterId: null,
    // position and velocity will be managed client-side; we include for
completeness:
    x: 0, y: 0, vx: 0, vy: 0
  };
  agents.push(newAgent);
  res.json(newAgent);
  io.emit('agentAdded', newAgent);  // notify all clients in real-time
});

// Get one agent's status
app.get('/api/agents/:id', (req, res) => {
  const agent = findAgent(req.params.id);
  if (!agent) {
    return res.status(404).send({ error: 'Agent not found' });
  }
  res.json(agent);
});

// Send a message to an agent and get response (text-based)
app.post('/api/agents/:id/message', async (req, res) => {
  const agent = findAgent(req.params.id);
  const userMessage = req.body.message || '';
  if (!agent || !userMessage) {
    return res.status(400).send({ error: 'Missing agent or message' });
  }
  try {
    // OpenAI Chat Completion call
    const openai = new OpenAIApi(new Configuration({ apiKey:
process.env.OPENAI_API_KEY }));
    const systemPrompt = agent.persona || `You are ${agent.name}, an
```

```
intelligent agent in a simulation. Respond succinctly.`;
    const response = await openai.createChatCompletion({
      model: process.env.OPENAI_MODEL || "gpt-3.5-turbo",
      messages: [
        { role: "system", content: systemPrompt },
        { role: "user", content: userMessage }
      ]
    });
    const reply = response.data.choices[0]?.message?.content?.trim() || "";
    // (Optionally, we could maintain a conversation history per agent for
context)
    // Emit the agent's response via WebSocket so the client can display it
or speak it
    io.emit('agentResponse', { agentId: agent.id, message: reply });
    res.json({ reply });
  } catch (err) {
    console.error("OpenAI API error:", err);
    res.status(500).send({ error: "Agent failed to respond" });
  }
});

// Export simulation state (all agents and perhaps recent logs, in one JSON)
app.get('/api/export', (req, res) => {
  const exportData = {
    timestamp: new Date().toISOString(),
    agents: agents
    // We could include more, e.g., clusters or interaction logs
  };
  res.header('Content-Type','application/json');
  res.header('Content-Disposition','attachment; filename="simulation-
state.json"');
  res.send(JSON.stringify(exportData, null, 2));
});

// (Optional) Additional routes for toggling behaviors or retrieving metrics
could be added.

////////////////////////////////////////////
// WebSocket event handlers
////////////////////////////////////////////
io.on('connection', socket => {
  console.log("Client connected to WebSocket.");
  // Send current agents list to new client
  socket.emit('init', agents);

  // Example: handle real-time control events from client (if any)
  socket.on('tweakSettings', params => {
    // e.g., user adjusted a global parameter
    // (In this basic implementation, most simulation logic runs on client,
    // so we might just broadcast to other clients or update server-side
config if needed)
```

```
    io.emit('settingsChanged', params);
  });

  // Handle client requesting to remove an agent
  socket.on('removeAgent', id => {
    agents = agents.filter(a => a.id !== id);
    io.emit('agentRemoved', id);
  });

  // If we wanted to handle voice streaming, we could receive audio data
chunks here
  // and forward to a STT service or process accordingly.
});

// Start the server
server.listen(PORT, () => {
  console.log(`Server listening on http://localhost:${PORT}`);
});
```

**Key points in the server code:** - We use `express.json()` to parse JSON bodies for POST requests. This allows the client to easily send JSON payloads (for creating agents or messaging). - For the OpenAI integration in `/api/agents/:id/message`, we construct a prompt with a system message (defining the agent's persona) and the user's message, then get a completion. We broadcast the result via WebSocket (`agentResponse` event) so that if multiple clients (or multiple components) are connected, all can hear the agent's reply. The client listening will display the reply in the chat UI and optionally use the Web Speech API to synthesize speech. - The server maintains a simple `agents` array that mirrors the agents in the simulation. In a real deployment, the simulation might run server-side or share state, but here the source of truth for physics is the client. So why keep `agents` on server? Mainly for API responses and for any server-side logic (like the chat persona or recording stats). We update and broadcast when new agents are added or removed so all clients stay in sync. The client also updates the server via API or WS when certain changes happen (in this design, the client might send a REST request or WS message when an agent's twin forms or an entropy changes significantly, so the server can update `agents` state or log it). - We included an `init` event on connection to send the current agents to a newly connected client, ensuring it can immediately render the existing simulation state. - CORS isn't explicitly handled here, assuming the UI is served from same origin. If external apps call the API, enabling CORS might be needed.

### Front-End: HTML Structure (`public/index.html`)

The HTML page sets up the layout: a full-screen app with a left control panel and a canvas filling the rest. We include Bootstrap 5 for easy styling of forms and use a custom CSS for specific positioning.

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>AI Agent Simulation Dashboard</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/
bootstrap.min.css" rel="stylesheet">
```

```
  <link href="style.css" rel="stylesheet">
</head>
<body>
  <!-- Control UI Panel -->
  <div id="ui" class="text-light">
    <h4 class="mb-3">AI Agent Dashboard  </h4>
    <!-- Agent addition form -->
    <form id="addForm" class="mb-2 d-flex gap-2">
      <input id="agentNameInput" class="form-control form-control-sm"
placeholder="Agent name (optional)" />
      <select id="agentTypeSelect" class="form-select form-select-sm"
style="max-width: 6rem;">
        <option>Type</option>
        <option value="philosopher">Philosopher</option>
        <option value="analyst">Analyst</option>
        <option value="artist">Artist</option>
      </select>
      <button class="btn btn-sm btn-primary" type="submit">Add</button>
    </form>
    <!-- Simulation parameter toggles/sliders -->
    <div class="mb-2">
      <label class="form-label fw-bold">Simulation Settings</label>
      <div class="form-check form-switch">
        <input class="form-check-input" type="checkbox" id="toggleEvolution"
checked>
        <label class="form-check-label" for="toggleEvolution">Evolution
(agents mutate)</label>
      </div>
      <div class="form-check form-switch">
        <input class="form-check-input" type="checkbox" id="toggleMimic"
checked>
        <label class="form-check-label" for="toggleMimic">Mimicry (converge
traits)</label>
      </div>
      <div class="form-check form-switch">
        <input class="form-check-input" type="checkbox" id="toggleEntropyViz"
checked>
        <label class="form-check-label" for="toggleEntropyViz">Entropy →
Color</label>
      </div>

<!-- Add more toggles as needed for symbolic/binary modes, sounds, etc. -->
      <div class="mt-2">
        <label class="form-label">Cluster Threshold</label>
        <input type="range" id="clusterThreshold" min="0" max="1" step="0.01"
value="0.1" class="form-range">
      </div>
      <div>
        <label class="form-label">Learning Rate</label>
        <input type="range" id="learningRate" min="0" max="0.1" step="0.005"
value="0.01" class="form-range">
```

```
      </div>
    </div>
    <hr class="border-secondary">
    <!-- Agent list display -->
    <h6 class="mb-1">Agents</h6>
    <div id="agentList" class="mb-2 small"></div>
    <!-- Chat interface -->
    <div id="chatInterface" class="mt-2">
      <div id="chatLog" class="border rounded p-2 mb-1" style="max-height:
100px; overflow:auto; font-size:0.9em;"></div>
      <div class="d-flex">
        <input id="chatInput" class="form-control form-control-sm me-1"
placeholder="Say something to agent...">
        <button id="sendBtn" class="btn btn-sm btn-success">Send</button>
      </div>
    </div>
    <hr class="border-secondary">
    <!-- Export and monitoring buttons -->
    <button id="exportJsonBtn" class="btn btn-outline-light btn-sm">Export
JSON</button>
    <button id="exportSvgBtn" class="btn btn-outline-light btn-sm">Export
SVG</button>
  </div>
  <!-- WebGL Canvas for simulation -->
  <canvas id="glCanvas"></canvas>

  <script src="/socket.io/socket.io.js"></script> <!-- Socket.io client lib
-->
  <script src="js/app.js"></script>
</body>
</html>
```

A few notes on this HTML: - The `#ui` panel is absolutely positioned via CSS to sit on the left, with a translucent background. It has a fixed width (we set in CSS) and can scroll if content overflows vertically. - The form at the top allows adding an agent. The user can enter a name and select a type (just examples: philosopher, analyst, artist – these could affect the agent's persona or color). - Under settings, we have toggles for Evolution and Mimicry (matching what we use in code) [5]. We also added a toggle "Entropy → Color" to let the user turn off coloring by entropy if they want a different visual mapping (in code we'll conditionally use entropy in color). - Two range sliders: *Cluster Threshold* could be used to set how close agents must be to be considered in a cluster or twin (though we might not implement full cluster logic beyond mimicry, we expose it for experimentation). *Learning Rate* could tie into the perceptron training speed if we implement online learning (not heavily used in this demo, but slider is there to showcase neural tuning capability). - Agent list (`#agentList`): this will be populated by script with a list of agents (each as a badge or pill). We'll make each badge clickable to select that agent for chatting. - Chat interface: `#chatLog` will show a small transcript of interactions with the selected agent. `#chatInput` and send button allow the user to send a message. We'll only enable these when an agent is selected (or we could allow broadcasting to all if no specific agent chosen). - Export buttons: trigger client-side logic to download JSON or SVG. - We include the Socket.io script for real-time comm and our `app.js` script where all client logic resides.

Finally, the `<canvas id="glCanvas">` is where WebGL draws the agents. Notice we set no explicit size here; the CSS will make it fill the window (as we'll see in CSS).

### CSS Styling (`public/style.css`)

We keep CSS minimal, mostly for layout:

```
html, body {
  width: 100%;
  height: 100%;
  margin: 0;
  padding: 0;
  overflow: hidden;
  background: #0d1117; /* dark background */
  color: #e6edf3;      /* light text (GitHub dark theme colors) */
  font-family: system-ui, sans-serif;
}
#ui {
  position: fixed;
  inset: 0 0 auto 0;    /* top:0, right:0, bottom:auto, left:0 -> docked to
left */
  width: 360px;
  max-height: 100vh;
  overflow-y: auto;
  background: rgba(0,0,0,0.5);
  backdrop-filter: blur(8px);
  padding: 16px;
  border-right: 1px solid rgba(255,255,255,0.1);
  z-index: 10;
}
#glCanvas {
  position: fixed;
  inset: 0;            /* fill entire viewport */
  display: block;
}
.badge.agent-badge {
  cursor: pointer;
  margin: 2px;
}
#chatLog {
  background: rgba(255,255,255,0.1);
}
```

This CSS: - Makes the body full-screen dark. - The `#ui` panel is translucent black, blurred background (for a glassmorphic effect), scrollable if needed. - The `#glCanvas` covers the whole screen behind the UI. - `.agent-badge` class (applied to agent badges in the list) adds a pointer cursor and a small margin. - `#chatLog` has a semi-transparent background to stand out.

### Client-Side JavaScript (`public/js/app.js`)

This is the core of the front-end logic: it initializes WebGL, runs the simulation loop, handles UI events (form submissions, button clicks), and interacts with the server (via REST fetch and Socket.io).

Key components in the client JS: - **Agent Class**: Represents an agent in the simulation. We define a constructor that either takes an object (from server or newly created) or generates random initial values. The Agent includes properties for position, velocity, entropy, color, perceptron, etc. It might also contain methods for behaviors (like update and mimic). - **ForceGraph Class**: Manages a collection of agents and computes physics each frame (repulsion, centering, etc) [1]. Also calls agent-specific updates (like evolution changes and mimicry) each step [16]. - **WebGL Setup**: Initialize a WebGL2 context, compile shaders (the vertex and fragment shader for point sprites), and prepare buffers and attribute locations [3] [19]. - **Rendering**: In each animation frame, build a vertex buffer of all agents (and possibly sub-elements like twin connections or orbiting parts if any; in this basic version, we just render agents). Then render with `gl.drawArrays(gl.POINTS, ...)`. - **UI event handlers**: - Add agent form: on submit, create a new Agent (and also POST to server to record it). - Toggle switches and sliders: on change, update global flags/values (like enabling/disabling mimicry in the simulation object, adjusting learning rate, etc). - Clicking an agent badge: mark that agent as selected (store selectedAgentId), update UI (highlight the badge, show chat interface). - Send chat: when clicking send, call the server API to get a response for that agent, and optimistically update the chatLog with the user query. - Export JSON: call `/api/export` and trigger download. - Export SVG: generate an SVG string from current agents and prompt download. - **Socket.io events**: - On connection, receive initial agent list (`init` event) to populate local simulation. - On `agentAdded` event, create that agent on client side. - On `agentRemoved`, remove it. - On `agentResponse`, if it matches the selected agent (or any agent, we can show it with name), display it in chat log and optionally speak it out.

Let's write the code in segments for clarity:

**a) Defining classes and global state:**

```
// app.js (client-side)

// ======= Global state and classes =======

// Simulation global toggles/params (will be tied to UI controls)
let settings = {
  evolution: true,
  mimicry: true,
  entropyViz: true,
  clusterThreshold: 0.1,
  learningRate: 0.01
};

// Agent class representing one node/agent in the simulation
class Agent {
  constructor({id, name, type="generic", entropy=Math.random()*5,
color=null} = {}) {
    this.id = id || `agent-${crypto.randomUUID().slice(0,8)}`;
    this.name = name || this.id;
    this.type = type;
```

```
    this.entropy = entropy;
    // Assign a color: if provided use it, otherwise pick semi-random (based
on type or random palette)
    if (color) {
      this.color = color;
    } else {
      // Simple deterministic color by type category:
      const typeColors = { philosopher: "#8EB49B", analyst: "#999DFF",
artist: "#FF9751" };
      this.color = typeColors[type] || pickColor();
    }
    // Position randomly in unit square, velocity 0
    this.x = Math.random()*2 - 1;
    this.y = Math.random()*2 - 1;
    this.vx = 0;
    this.vy = 0;
    // Radius related to entropy (min radius 5)
    this.r = 5 + this.entropy * 2;
    // Perceptron brain (2 inputs for now: perhaps entropy and 1 constant)
just for demonstration
    this.brain = new Perceptron(2);
    // Relationship placeholders
    this.twinId = null;
    this.clusterId = null;
  }
  // Update internal state (called each frame if evolution enabled)
  updateInternal() {
    if (!settings.evolution) return;
    // small random walk in entropy
    this.entropy += (Math.random() - 0.5) * 0.05;
    this.entropy = clamp(this.entropy, 0, 10);
    // update radius and maybe color based on new entropy
    this.r = 5 + this.entropy * 2;
    // Could also adjust perceptron weights slightly (simulating learning/
mutation)
    for (let i = 0; i < this.brain.w.length; i++) {
      this.brain.w[i] += (Math.random() - 0.5) * 0.01;
    }
  }
  // Mimicry: slowly approach a neighbor's traits (e.g., color and size)
  mimic(neighbors) {
    if (!settings.mimicry || neighbors.length === 0) return;
    // find nearest neighbor
    let closest = null;
    let minDistSq = Infinity;
    for (const other of neighbors) {
      if (other === this) continue;
      const dx = other.x - this.x, dy = other.y - this.y;
      const distSq = dx*dx + dy*dy;
      if (distSq < minDistSq) {
        minDistSq = distSq;
```

```
        closest = other;
      }
    }
    if (!closest) return;
    // blend this.color towards closest.color, blend size (entropy) as well
    const targetColor = closest.color;
    this.color = lerpColor(this.color, targetColor, 0.02);
    this.r = lerp(this.r, closest.r, 0.02);
    // Also, we could move entropy towards closest's entropy slightly:
    this.entropy = lerp(this.entropy, closest.entropy, 0.01);
  }
}

// Perceptron class (simple single-layer perceptron for demo)
class Perceptron {
  constructor(dim) {
    // initialize weights randomly between -1 and 1
    this.w = new Float32Array(dim);
    for (let i = 0; i < dim; i++) {
      this.w[i] = Math.random()*2 - 1;
    }
  }
  // output score
  score(inputs) {
    let s = 0;
    for (let i = 0; i < this.w.length; i++) s += this.w[i] * (inputs[i] ||
0);
    return s;
  }
  // one-step training (not heavily used here, but available for future use)
  train(inputs, expected, lr = settings.learningRate) {
    const guess = this.score(inputs) >= 0 ? 1 : -1;
    const label = expected >= 0 ? 1 : -1;
    if (guess !== label) {
      for (let i = 0; i < this.w.length; i++) {
        this.w[i] += lr * (inputs[i] || 0) * label;
      }
    }
  }
}

// Force-directed graph simulation
class ForceGraph {
  constructor() {
    this.nodes = [];
  }
  add(agent) {
    this.nodes.push(agent);
  }
  remove(agentId) {
    this.nodes = this.nodes.filter(node => node.id !== agentId);
```

```
  }
  step() {
    const repulsion = 0.0005;
    const damp = 0.9;
    // pairwise repulsion
    for (let i = 0; i < this.nodes.length; i++) {
      const a = this.nodes[i];
      for (let j = i+1; j < this.nodes.length; j++) {
        const b = this.nodes[j];
        let dx = a.x - b.x;
        let dy = a.y - b.y;
        let dist = Math.hypot(dx, dy) + 0.001;
        // repulsive force magnitude
        let force = repulsion / (dist * dist);
        // direction unit vector
        dx /= dist;
        dy /= dist;
        a.vx += dx * force;
        a.vy += dy * force;
        b.vx -= dx * force;
        b.vy -= dy * force;
      }
    }
    // agent-specific updates and mimicry
    for (const node of this.nodes) {
      node.updateInternal();
      node.mimic(this.nodes);
    }
    // centering force and velocity damping/integration
    for (const node of this.nodes) {
      node.vx += -node.x * 0.001;  // pull back to center
      node.vy += -node.y * 0.001;
      node.x += node.vx;
      node.y += node.vy;
      node.vx *= damp;
      node.vy *= damp;
    }
    // (Optional: cluster detection or twin pairing logic could be added here
based on proximity and similarity)
  }
}

// Utility functions
const clamp = (val, min, max) => val < min ? min : (val > max ? max : val);
const lerp = (a, b, t) => a + (b - a) * t;
const hexToRgb = hex => {
  const bigint = parseInt(hex.slice(1), 16);
  return [(bigint >> 16 & 255)/255, (bigint >> 8 & 255)/255, (bigint & 255)/
255];
};
const lerpColor = (hex1, hex2, t) => {
```

```
  const rgb1 = hexToRgb(hex1), rgb2 = hexToRgb(hex2);
  const r = Math.round(lerp(rgb1[0], rgb2[0], t) * 255);
  const g = Math.round(lerp(rgb1[1], rgb2[1], t) * 255);
  const b = Math.round(lerp(rgb1[2], rgb2[2], t) * 255);
  return `#${((1<<24) + (r<<16) + (g<<8) + b).toString(16).slice(1)}`;
};
// Color palette pick (for random color assignment)
const palette =
["#40f2d0","#999DFF","#FF9751","#8EB49B","#F28443","#71f2ff","#ce2d42","#06a0ba","#F4B53F","#A
function pickColor() {
  return palette[Math.floor(Math.random() * palette.length)];
}
```

This block defines our main classes largely based on the earlier discussion and example code. Notably: - `Agent.updateInternal` and `Agent.mimic` implement the evolution and mimicry behaviors, gated by the settings toggles [16]. The mimic function finds the nearest neighbor and gently moves this agent's color and size toward that neighbor [18] [5]. - The `ForceGraph.step` method applies repulsion between every pair (based on the IP-tag graph logic [1]) and a centering force. It then updates each agent's internal state and mimicry (each agent compares with *all others* for simplicity; optimizing to nearest neighbors only would be more complex but could be done). - We left a placeholder comment for cluster detection or twin pairing – e.g., one could loop through pairs and if distance < some threshold and difference in entropy < threshold, mark them as twins (`a.twinId = b.id` etc.) and perhaps emit an event or highlight them. - The perceptron class is kept, though our simulation isn't explicitly feeding it inputs each frame. It's a feature for future extension – e.g., if agents get classified inputs (maybe from messages or sensor data) we could train it. The UI slider "Learning Rate" is linked to `settings.learningRate` which is used in `Perceptron.train`.

**b) WebGL setup and rendering loop:**

```
// ======= WebGL Initialization =======
const canvas = document.getElementById('glCanvas');
const gl = canvas.getContext('webgl2');
if (!gl) {
  alert("WebGL2 not supported in this browser.");
}

// Adjust canvas size to window
function resizeCanvas() {
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;
  gl.viewport(0, 0, canvas.width, canvas.height);
}
window.addEventListener('resize', resizeCanvas);
resizeCanvas();

// Vertex and Fragment shader source
const vertSrc = `#version 300 es
layout(location=0) in vec2 aPos;
layout(location=1) in float aSize;
layout(location=2) in vec3 aColor;
```

```
uniform vec2 uScale;
out vec3 vColor;
void main() {
    vColor = aColor;
    gl_PointSize = aSize;
    vec2 pos = aPos * uScale;
    gl_Position = vec4(pos, 0.0, 1.0);
}`;
const fragSrc = `#version 300 es
precision highp float;
in vec3 vColor;
out vec4 outColor;
void main() {
    float dist = length(gl_PointCoord - 0.5);
    if (dist > 0.5) {
      discard;  // draw circular point
    }
    // fade out at edges of circle (smooth)
    outColor = vec4(vColor, 1.0 - dist * 2.0);
}`;

// Compile shaders and link program
function compileShader(type, source) {
  const shader = gl.createShader(type);
  gl.shaderSource(shader, source);
  gl.compileShader(shader);
  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    console.error("Shader error:", gl.getShaderInfoLog(shader));
    gl.deleteShader(shader);
  }
  return shader;
}
const vs = compileShader(gl.VERTEX_SHADER, vertSrc);
const fs = compileShader(gl.FRAGMENT_SHADER, fragSrc);
const shaderProg = gl.createProgram();
gl.attachShader(shaderProg, vs);
gl.attachShader(shaderProg, fs);
gl.linkProgram(shaderProg);
if (!gl.getProgramParameter(shaderProg, gl.LINK_STATUS)) {
  console.error("Shader link error:", gl.getProgramInfoLog(shaderProg));
}

// Get attribute/uniform locations
const uScaleLoc = gl.getUniformLocation(shaderProg, "uScale");
const aPosLoc = 0, aSizeLoc = 1, aColorLoc = 2;  // using layout(location)
const vbo = gl.createBuffer();

// ======= Simulation and Rendering =======
const graph = new ForceGraph();

// Render loop
```

```
function animate() {
  requestAnimationFrame(animate);
  graph.step();  // update simulation physics and agent states

  // Prepare vertex buffer data for all agents
  const n = graph.nodes.length;
  if (n === 0) {
    gl.clear(gl.COLOR_BUFFER_BIT);
    return;
  }
  // Each agent contributes 6 floats: (x, y, size, r, g, b)
  const data = new Float32Array(n * 6);
  for (let i = 0; i < n; i++) {
    const agent = graph.nodes[i];
    data[i*6 + 0] = agent.x;
    data[i*6 + 1] = agent.y;
    data[i*6 + 2] = agent.r;
    // Determine color components:
    let rCol=0.9, gCol=0.9, bCol=0.9;  // default white-ish
    if (settings.entropyViz) {
      // Map entropy to color: use entropy (0-? maybe up to ~10) to red,
normalize a bit
      rCol = clamp(agent.entropy / 5, 0, 1);
      gCol = 0.2;
      bCol = clamp(1 - agent.entropy / 10, 0, 1);
    }
    // Alternatively, if symbolic mode is toggled, could assign colors by
agent type or cluster.
    // For simplicity, we already assign agent.color as a hex.
    // We'll use agent.color for final color, but blend it with entropy
mapping if needed:
    const baseRgb = hexToRgb(agent.color);
    if (settings.entropyViz) {
      // combine base color with entropy color (just a simple way: average
them)
      data[i*6 + 3] = (baseRgb[0] + rCol) / 2;
      data[i*6 + 4] = (baseRgb[1] + gCol) / 2;
      data[i*6 + 5] = (baseRgb[2] + bCol) / 2;
    } else {
      data[i*6 + 3] = baseRgb[0];
      data[i*6 + 4] = baseRgb[1];
      data[i*6 + 5] = baseRgb[2];
    }
  }

  // Render points
  gl.clearColor(0.05, 0.07, 0.09, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.useProgram(shaderProg);
  // Upload data to GPU
  gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
```

```
  gl.bufferData(gl.ARRAY_BUFFER, data, gl.DYNAMIC_DRAW);
  // Set uniform scale (project to screen: x in [-1,1] maps to view)
  const scaleX = 2 / canvas.width;
  const scaleY = 2 / canvas.height;
  gl.uniform2f(uScaleLoc, scaleX, scaleY);
  // Enable and define vertex attributes
  gl.enableVertexAttribArray(aPosLoc);
  gl.vertexAttribPointer(aPosLoc, 2, gl.FLOAT, false, 24, 0);
  gl.enableVertexAttribArray(aSizeLoc);
  gl.vertexAttribPointer(aSizeLoc, 1, gl.FLOAT, false, 24, 8);
  gl.enableVertexAttribArray(aColorLoc);
  gl.vertexAttribPointer(aColorLoc, 3, gl.FLOAT, false, 24, 12);
  // Draw all agents as points
  gl.drawArrays(gl.POINTS, 0, n);
}
animate();
```

This code closely mirrors the structure of the earlier IP graph demo: - It compiles a simple vertex shader and fragment shader for point rendering [3] [4]. We use `layout(location=X)` in the shader to fix attribute indices, so we can use those known indices in `vertexAttribPointer` calls. - The vertex format is 2 floats (pos) + 1 float (size) + 3 floats (color) = 24 bytes per agent, matching the usage in the code [19]. - We compute `uScale` such that our simulation coordinates (roughly in [-1,1]) map to the canvas. This is exactly the approach shown in the IP Tag Graph code (they used `gl.uniform2f(uScale, 2/innerWidth, 2/innerHeight)` which we emulate [20]). - Color logic: if `settings.entropyViz` is true, we incorporate the agent's entropy into its color (in this case, making high entropy more red and low entropy more blue, arbitrarily chosen for demo). We combine it with the agent's base color. If entropyViz is off, we just use the agent's assigned color. This provides the *aesthetic settings toggle* for color mapping. - We clear the canvas with a dark color each frame and draw points for all agents.

**c) UI Event Handling and Server Communication:**

```
// ======= UI Interaction & Networking =======

// Connect to WebSocket (Socket.io)
const socket = io();  // connect to same host
socket.on('init', serverAgents => {
  // Initialize agents from server data
  serverAgents.forEach(data => {
    const agent = new Agent(data);
    graph.add(agent);
    addAgentBadge(agent);
  });
});
socket.on('agentAdded', data => {
  const agent = new Agent(data);
  graph.add(agent);
  addAgentBadge(agent);
});
socket.on('agentRemoved', id => {
```

```
  graph.remove(id);
  const badge = document.getElementById(`badge-${id}`);
  if (badge) badge.remove();
  if (selectedAgentId === id) {
    selectedAgentId = null;
    document.getElementById('chatInterface').style.display = 'none';
  }
});
socket.on('agentResponse', ({agentId, message}) => {
  // Display agent's response in chat log (if it's the selected agent or show
with name)
  const agent = graph.nodes.find(a => a.id === agentId);
  if (!agent) return;
  const name = agent.name;
  appendChat(`${name}: ${message}`, 'agent');
  // Optionally, use speech synthesis to speak it
  if ('speechSynthesis' in window) {
    const utter = new SpeechSynthesisUtterance(message);
    utter.lang = 'en-US';
    window.speechSynthesis.speak(utter);
  }
});

// DOM elements for controls
const agentListDiv = document.getElementById('agentList');
let selectedAgentId = null;

// Helper to add an agent badge to list
function addAgentBadge(agent) {
  const span = document.createElement('span');
  span.className = "badge rounded-pill bg-secondary agent-badge";
  span.id = `badge-${agent.id}`;
  span.textContent = agent.name;
  span.style.backgroundColor = agent.color;
  span.onclick = () => {
    // Select this agent for chat
    document.querySelectorAll('.agent-badge').forEach(el =>
el.classList.remove('bg-info'));
    span.classList.add('bg-info');
    selectedAgentId = agent.id;
    document.getElementById('chatInterface').style.display = 'block';
    appendChat(`-- Chatting with ${agent.name} --`, 'system');
  };
  agentListDiv.appendChild(span);
}

// Append a message to the chat log
function appendChat(msg, sender='user') {
  const logDiv = document.getElementById('chatLog');
  const p = document.createElement('div');
  p.className = sender === 'user' ? 'text-warning' : (sender==='agent' ?
```

```
'text-info' : 'text-muted');
  p.textContent = msg;
  logDiv.appendChild(p);
  logDiv.scrollTop = logDiv.scrollHeight;  // auto-scroll
}

// Form: Add Agent
document.getElementById('addForm').addEventListener('submit', e => {
  e.preventDefault();
  const nameInput = document.getElementById('agentNameInput');
  const typeSelect = document.getElementById('agentTypeSelect');
  const name = nameInput.value.trim();
  const type = typeSelect.value;
  // Create agent locally
  const agent = new Agent({ name: name || undefined, type: type && type !==
'Type' ? type : 'generic' });
  graph.add(agent);
  addAgentBadge(agent);
  // Send to server (so it knows about it and can persist/distribute)
  fetch('/api/agents', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ name: agent.name, type: agent.type, entropy:
agent.entropy, color: agent.color })
  }).catch(err => console.error("Failed to add agent on server:", err));
  // clear form inputs
  nameInput.value = '';
  typeSelect.value = 'Type';
});

// Handle chat send
document.getElementById('sendBtn').addEventListener('click', () => {
  sendChat();
});
document.getElementById('chatInput').addEventListener('keypress', e => {
  if (e.key === 'Enter') {
    sendChat();
  }
});
function sendChat() {
  const inputEl = document.getElementById('chatInput');
  const message = inputEl.value.trim();
  if (!message || !selectedAgentId) {
    return;
  }
  appendChat(`You: ${message}`, 'user');
  inputEl.value = '';
  // Send message to server to get agent response
  fetch(`/api/agents/${selectedAgentId}/message`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
```

```
    body: JSON.stringify({ message: message })
  }).catch(err => console.error("Error sending message:", err));
}

// Toggle and slider events
document.getElementById('toggleEvolution').onchange = e => {
  settings.evolution = e.target.checked;
};
document.getElementById('toggleMimic').onchange = e => {
  settings.mimicry = e.target.checked;
};
document.getElementById('toggleEntropyViz').onchange = e => {
  settings.entropyViz = e.target.checked;
};
document.getElementById('clusterThreshold').oninput = e => {
  settings.clusterThreshold = parseFloat(e.target.value);
};
document.getElementById('learningRate').oninput = e => {
  settings.learningRate = parseFloat(e.target.value);
};

// Export buttons
document.getElementById('exportJsonBtn').onclick = async () => {
  try {
    const res = await fetch('/api/export');
    const data = await res.blob();
    const url = URL.createObjectURL(data);
    const link = document.createElement('a');
    link.href = url;
    link.download = 'agents-export.json';
    document.body.appendChild(link);
    link.click();
    link.remove();
    URL.revokeObjectURL(url);
  } catch (err) {
    console.error("Export JSON failed", err);
  }
};
document.getElementById('exportSvgBtn').onclick = () => {
  // Create an SVG snapshot of current agent positions
  const svgNS = "http://www.w3.org/2000/svg";
  const svgElem = document.createElementNS(svgNS, "svg");
  svgElem.setAttribute("xmlns", svgNS);
  svgElem.setAttribute("width", canvas.width);
  svgElem.setAttribute("height", canvas.height);
  // Draw each agent as a circle in the SVG
  graph.nodes.forEach(agent => {
    const cx = ((agent.x + 1) / 2) * canvas.width;
    const cy = ((-agent.y + 1) / 2) * canvas.height;  // note: invert y for
SVG coordinate
    const circle = document.createElementNS(svgNS, "circle");
```

```
    circle.setAttribute("cx", cx);
    circle.setAttribute("cy", cy);
    circle.setAttribute("r", agent.r);
    circle.setAttribute("fill", agent.color);
    svgElem.appendChild(circle);
  });
  // Serialize SVG and trigger download
  const svgData = new XMLSerializer().serializeToString(svgElem);
  const blob = new Blob([svgData], { type: "image/svg+xml;charset=utf-8" });
  const url = URL.createObjectURL(blob);
  const link = document.createElement('a');
  link.href = url;
  link.download = 'agents-snapshot.svg';
  document.body.appendChild(link);
  link.click();
  link.remove();
  URL.revokeObjectURL(url);
};
```

In this final part: - We set up the Socket.io client to listen for events: - On `init`: populate the graph with existing agents sent by server (for when a client first connects, so it syncs). We create each agent via our `Agent` class and add to the simulation, and call `addAgentBadge` to update the UI list. - On `agentAdded`: do the same for a newly added agent from another client or via API. - On `agentRemoved`: remove from simulation and UI. - On `agentResponse`: when the server broadcasts an AI response, we add it to the chat log. We also use the browser's Speech Synthesis API to speak the response aloud (so the user hears the agent). We check `speechSynthesis` support before using it. - The `addAgentBadge` function creates a Bootstrap badge for each agent with a background color matching the agent's color. Clicking the badge marks that agent as the "selected" one (highlighting it with a different class, and storing `selectedAgentId`). It also reveals the chat interface (making the chat input visible) and logs a system message indicating the chat target. - The chat send logic sends the user message to the server via a fetch POST to the appropriate `/message` API. We do not wait for the response here; instead we rely on the WebSocket `agentResponse` event to update the chat log when the reply arrives. This asynchronous pattern decouples the UI from waiting on the HTTP response (though we could also handle it via the fetch promise, but using the WS means we also catch replies initiated by other clients). - The toggle handlers simply update the `settings` object which is read each frame in the simulation and rendering. - The export JSON button triggers the `/api/export` route and uses a blob URL to download the returned JSON file. - The export SVG button constructs an SVG by iterating over current agent positions. We convert the normalized coordinates to pixel positions (taking care to invert Y because canvas/WebGL Y=+up whereas SVG Y=+down). We append circle elements for each agent with the same radius and color. Then serialize and force download. This yields a snapshot image that can be opened in Inkscape or any SVG viewer, effectively a vector screenshot of the network.

With the code above, the application is complete. **To run the project:** install dependencies (`npm install`), ensure you have an OpenAI API key in `.env`, then start the server (`npm start` or `node server.js`). Open `http://localhost:3000` in a browser to view the dashboard. You can add agents using the form, see them appear on the canvas, tweak simulation parameters, and click an agent to chat with it. The agents' responses are generated via OpenAI and will display in the chat log (and play via speech). You can observe clusters forming visually if mimicry is on (agents of similar type/traits will drift together and unify color), and export data or SVG snapshots at any time for analysis. The design is modular, so developers can extend agent logic (e.g. add new behaviors like agents forming/

losing twin links based on the `clusterThreshold`, or integrating more sensor data), add new UI controls, or hook the system up to other interfaces using the API/WebSocket.

## Sources

- The force-directed graph and WebGL point rendering are adapted from the project's earlier interactive demos [1] [2], with enhancements for nested sub-nodes and mimicry [5] [16].
- Agent internal logic (perceptron, entropy) and the concept of evolving agent state draw on prior agents like *AlphaEvolveAgent* (which evolved neural weights) [21] and *QuantumJellyfishAgent* (which ticked state variables over time) [22].
- OpenAI integration approach aligns with the system's chat module (requiring an `OPENAI_API_KEY` in environment) and follows the documented API usage for chat completions.
- The dynamic UI and export features were informed by the design of existing sandbox tools and the need for a self-contained, user-friendly interface for creative AI simulations.

---

[1] [2] [3] [4] [10] [11] [12] [15] [19] [20] interactive_ip_tag_graph.html
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/static/hmode/interactive_ip_tag_graph.html

[5] [16] [17] [18] interactive_ip_tag_graph-2.html
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/static/hmode/interactive_ip_tag_graph-2.html

[6] README.md
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/README.md

[7] [8] orchestrator.ts
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/uniphi/interwined-cluster/apps/server/src/agents/orchestrator.ts

[9] hyper-guardian-agent.md
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/docs/hyper-guardian-agent.md

[13] [14] agent-architecture.md
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/docs/agent-architecture.md

[21] alphaEvolveAgent.js
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/agents/alphaEvolveAgent.js

[22] quantum-jellyfish-agent.md
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/docs/quantum-jellyfish-agent.md