ChatGPT

Replicating FloraFauna AI Architecture in Floranodus Monorepo

1. Overview of FloraFauna.ai Architecture and Design System

FloraFauna.ai introduced a powerful **AI-driven node editor interface** built with React, integrated with a Figma-based design system and multiple local AI services. Key architectural elements include:

- **Infinite Canvas Node Editor:** A React Flow–based canvas for arranging nodes, each representing an AI function or data element [1]. Nodes can be connected via input/output handles to form pipelines. The canvas supports panning/zooming and shows a minimap and controls.
- **Specialized Node Types:** A variety of custom React components represent different node types for multimodal AI operations [1]. These include:
- *Data Input Nodes* – for uploading or storing data (JSON, CSV, text, images).
- *Local LLM Nodes (OllamaNode)* – for text generation using local large language models (via Ollama).
- *Image Generation Nodes (ComfyUINode)* – for image synthesis using Stable Diffusion (via ComfyUI).
- *AI Workflow Nodes (FlowiseNode)* – for executing saved AI chains or flows (via Flowise).
- *Generic AI Processing Nodes* – a general processor that can be extended for various models.
- *Figma Design Nodes* – for linking to UI designs in Figma and syncing design changes.

These node types are registered in a central mapping and categorized for easy addition [2] [3]. For example, the **"AI Processing"** category includes nodes like *Local LLM* (Ollama) and *AI Workflow* (Flowise) [4], the **"Creative"** category includes *Image Generation* (ComfyUI) and *Figma Design* nodes [5], and **"Data"** includes a *Data Input* node [6]. - **Shared Design System:** FloraFauna uses a consistent design system derived from Figma. Colors, spacing, and other design tokens are extracted from Figma and used in the React app via CSS custom properties (e.g. `--figma-node-bg`, `--figma-spacing-md`) [7] [8]. These tokens are integrated so that the React components' styling matches the Figma designs (e.g. background, borders, radii). The project's Tailwind CSS is configured with custom classes (e.g. `bg-floranodus-bg-primary`) that map to these Figma-derived values. This ensures a single source of truth for UI design . *For instance, Figma's exported CSS might define* `--figma-node-bg` *and* `--figma-node-border` *colors, which are then applied in Tailwind utility classes for node backgrounds and borders.* The Floranodus monorepo already includes a `design-tokens.css` file with these values [7]. - **Figma Integration (Unified MCP Bridge):** A two-way integration with Figma allows generating UI components in Figma from code and pulling updated designs back. FloraFauna's Figma **Unified Bridge** (using the Model Context Protocol) runs as a local server that the app communicates with. This enables "round-trip" design sync: the app can create or update Figma frames via API, and retrieve changes from Figma into the app. For example, an "AI Node Card" designed in Figma can be linked to its React component, allowing updates in one environment to reflect in the other. The monorepo's Figma bridge supports over 22 design operations (creating frames, setting styles, grouping, etc.) and runs as an MCP server [9]. In Cursor IDE, this appears as the **Figma AI Bridge** server. - **Local AI Service Integrations:** FloraFauna nodes interface with local AI backends: - **Ollama (Local LLM)** – Runs text generation models on the user's machine. The Ollama node lists available models, allows pulling new models, and streams generation results. It supports multimodal models (like LLaVA) by accepting an optional image input. - **ComfyUI (Stable Diffusion)** – Handles image generation. The ComfyUI node lets the user select a diffusion model (e.g. SDXL) and input a prompt. It builds a ComfyUI workflow under the hood, triggers image generation, and displays the resulting image. - **Flowise** – Executes predefined AI workflows (graphs of LLM/toolchain operations) via Flowise's API. The Flowise node allows selecting a saved "chain" and running it to get results. - **(Extensible)** – The architecture is designed to be multimodal: new services for audio or video can be added similarly. For example, one could create a **Speech-to-Text node** using a local Whisper ASR service or a **Text-to-Speech node** using a tool like Coqui TTS, following the same pattern of a React node + service API.

All these pieces work together in FloraFauna.ai to provide an **AI-native creative canvas** – an interface where a developer or designer can drag-and-drop functional AI components, *visually design UI elements via Figma*, and run multimodal AI pipelines on local inference engines.

2. Planning the Integration into Floranodus

Now, we will **replicate and integrate** this entire architecture into the `floranodus-monorepo` project's React frontend. The goal is to rebuild the UI framework, React components, and service integration from FloraFauna inside Floranodus **step by step**, maintaining consistent architecture and design patterns. We will leverage the Cursor IDE environment (with the Figma VSCode extension and a running Unified Figma Bridge) for a seamless development workflow.

High-Level Plan:

1. **Set Up the Design System in Floranodus:** Ensure that Floranodus has the same design tokens and Tailwind classes as FloraFauna so the UI looks consistent. Import Figma-derived CSS and verify the custom Tailwind theme is applied.
2. **Bring Over Core React Components (Node System):** Recreate each key component from FloraFauna in the Floranodus frontend:
3. Base node container and shared UI elements.
4. Specific node components for data input, AI services (Ollama, ComfyUI, Flowise, etc.), and design sync.
5. Auxiliary UI components like the canvas container, controls/toolbars, settings panel, and navigation as needed.
6. **Reproduce the Canvas and Node Editor Behavior:** Set up the React Flow (or equivalent) canvas in Floranodus, register the node types, define initial nodes/edges, and implement drag-and-drop or menu-based node creation.
7. **Integrate Figma Bridge Hooks:** Use the provided `useFigmaSync` hook (or equivalent) to connect to the running Figma MCP bridge and implement **syncing logic** for components. Each component that represents a design should be able to push to Figma (create/update a Figma frame) and pull updates back.
8. **Connect Local AI Services:** Configure Floranodus to communicate with local AI backends (Ollama, ComfyUI, Flowise). This involves copying or implementing service modules (API clients) and ensuring the services are running (with any required environment variables or endpoints configured).
9. **Preserve Authentication Flow:** Floranodus already has Google Sign-In (OAuth) for user login. We must integrate the new features *after* authentication – i.e. ensure that the new canvas UI appears only once the user is logged in, and that nothing in the integration breaks the login logic.

10. **Test Round-Trip Workflows:** After integration, thoroughly test creating components in React, syncing them to Figma (and seeing them in the VSCode Figma panel), making visual tweaks in Figma, and syncing those back into the app. Also test connecting nodes and running the local AI inference to verify end-to-end functionality.

We'll go through each of these steps in detail.

3. Setting Up Floranodus Design System and Figma Bridge

Before adding new UI components, ensure Floranodus has the same **design foundation** as FloraFauna:

- **Import Design Tokens CSS:** Copy over the Figma design tokens CSS (or ensure it's generated via the bridge) into the Floranodus app. In the monorepo, there's likely a file at `packages/floranodus-app/design/figma/design-tokens.css` containing Figma variables. Import this in the React app's entry point or global stylesheet. For example, FloraFauna's quick start guide instructs adding:

```
// Import design tokens from Figma
import '../design/figma/design-tokens.css';
```

This makes variables like `--figma-node-bg` and `--figma-spacing-md` available in CSS [10]. Floranodus already defines Tailwind utility classes (like `.bg-floranodus-bg-secondary`, `.text-floranodus-text-primary`) that likely reference these variables in its Tailwind config or CSS. For instance, in `globals.css` the base node class uses those custom utilities:

```
.floranodus-node {
  @apply bg-floranodus-bg-secondary border-2 rounded-lg p-3 ... text-
floranodus-text-primary;
}
.floranodus-node.selected {
  @apply border-floranodus-accent-primary ...;
}
```

This ensures the **background color, border, text color** etc. come from the Figma-derived theme [11] [12]. Verify that these classes correspond to the CSS variables (e.g. in `design-tokens.css`, `--figma-node-bg` might be mapped to a Tailwind color named `floranodus-bg-secondary`). If not already set up, you can update **tailwind.config.js** to include the Figma token values under custom colors.

- **Ensure Figma Bridge Connectivity:** Since you have the **Unified Figma Bridge server** running (likely via the `floranodus-figma-write MCP` plugin), Floranodus needs to know how to connect. The `useFigmaSync` hook in Floranodus handles this. It will attempt to connect to the bridge at startup and periodically check connection [13]. By default the bridge runs at `http://localhost:3002` (as set in `figmaService.bridgeUrl`) [14] – confirm this matches your actual bridge port (the monorepo's `.cursor/mcp.json` config shows how the server is launched, often on port 3000 or 3001). If needed, adjust the URL or port in `figmaService.ts`.

When the React app starts, `useFigmaSync` will automatically ping the bridge's `/status` endpoint and log a success if connected [13]. In the UI, Floranodus even shows a small indicator "● Figma Connected" on the canvas when the connection is live [15]. Make sure to include that indicator or a similar UI cue so you know the bridge is ready.

- **Figma File Setup:** In your `.env` or config for the Figma plugin, ensure the `FIGMA_API_TOKEN` and possibly a `FIGMA_FILE_ID` are configured (the README snippet shows these variables [16]). The Figma Bridge needs an API token with access to a Figma file where it will create frames/components for your UI. You might have already set this up; just double-check that Floranodus uses the same file or a new one for its designs. The integration will be smoother if you point Floranodus to a Figma file dedicated to this project.

- **Cursor IDE Integration:** With the design tokens and bridge in place, take advantage of Cursor's VSCode features:

- Open the **Figma Panel** in VSCode (`View → Open View → Figma`) to see the Floranodus Figma file live [17]. As you generate or sync components, they will appear here.
- Use the predefined tasks: The monorepo defines tasks such as " Start Figma AI Bridge" and " Generate AI Components" [18]. These can be run via **Terminal > Run Task** in VSCode. For example, "Generate AI Components" likely runs `pnpm generate` (or `npm run generate`) which invokes a script to create a full set of UI frames in Figma. Ensure those tasks are accessible and working. (Since you already have the bridge running, focus on generation tasks).
- Confirm design token syncing: There is a `pnpm figma:sync` command to pull the latest design tokens from Figma into `design-tokens.css` [19]. After you adjust any design in Figma, run this to update the CSS variables in your codebase.

At this stage, Floranodus should have the *same color scheme, spacing, and base styles* as FloraFauna. The Figma bridge connection should be confirmed (you can run `pnpm status` or `pnpm health` as provided [20] to verify the servers). Now we can proceed to integrate the React components.

4. Rebuilding Core React Components in Floranodus

Next, integrate **all key React components and node types** from FloraFauna into Floranodus. We will create each component in `packages/floranodus-app/frontend/src/components/...` following FloraFauna's architecture. The major components and their structure are:

4.1 BaseNode Component (Node Container)

Begin with a `BaseNode` component – this is the foundation that defines how every node is rendered on the canvas. In FloraFauna, BaseNode was a reusable wrapper that provides the node's **visual container, title, description, icon, and connectors**. Reimplement it similarly in Floranodus:

- **File**: `src/components/nodes/BaseNode.tsx` (if not already present).
- **Purpose**: Wrap node content with a styled card and input/output handles.
- **Structure**: Use a `<div>` for the outer node with a Tailwind class like `.floranodus-node` (which applies background, padding, border-radius, etc. from the design system [11]). Inside it, include:

- A left-side handle for input connections and a right-side handle for output connections. These are provided by the React Flow library. For example, in FloraFauna's code:

```
<Handle type="target" position={Position.Left} className="...!w-3 !h-3" />
... node content ...
<Handle type="source" position={Position.Right} className="...!w-3 !h-3" />
```

ensures every node has a small circular port on its left and right [21] [22]. Use similar styling (`bg-floranodus-accent-primary` etc.) for the handle so it's visible.
- A header area displaying an icon (if any) and the node label. FloraFauna uses Lucide icons; ensure those are installed. For example:

```
{IconComponent && <IconComponent size={16} className="text-floranodus-
accent-primary" />}
<span className="font-medium text-sm">{data.label}</span>
```

wrapped in a flex row [23]. The `data.label` will be a short title like "Image Generation" or "Local LLM".
- An optional description text below the title, for a subtitle or status. In BaseNode, this was rendered if `data.description` exists, in a smaller, secondary-colored font [24].
- A slot for child elements (`{children}`) – this is where specialized node content (like buttons, fields) will go in the node's body [25].
- **Visual States**: BaseNode should reflect selection and processing state. In CSS, FloraFauna's `.floranodus-node.selected` class added a highlighted border and shadow [26]. They also used a `data.status` field to color the border: e.g., nodes in a "processing" state get a pulsating accent border [27] [28]. Reuse these patterns:
- Define status classes: e.g., idle (default border), processing (animate pulse in border color), success (green border), error (red border) [27].
- Apply `selected ? 'selected' : ''` class when the node is clicked, and set the border class based on `data.status` [29].

Best Practice: Use the **design tokens** for all styling in BaseNode – either via Tailwind classes or CSS-in-JS. This ties the component to Figma's design. For example, ensure the padding equals `var(--figma-spacing-md)` and border radius uses `--figma-radius-md` (the quick-start snippet shows using these variables in a styled component [30]). In Tailwind, this might already be baked into classes like `rounded-lg` if configured, but confirm the values match.

Once BaseNode is in place, all specific node components will extend it.

4.2 DataNode – Data Input Component

File: `src/components/nodes/DataNode.tsx`
Purpose: Allow users to import data (text, images, CSV, JSON) into the canvas, which can then feed into AI nodes.

Implementation: The DataNode will use BaseNode as a wrapper, passing in a database icon and setting type `'data'`. Inside BaseNode's children, implement a small form: - A dropdown to select the data type (JSON, CSV, Image, or Text). In FloraFauna's example, it looks like:

```
<select className="w-full ... text-xs" value={data.dataType || 'json'}>
  <option value="json">JSON</option>
  <option value="csv">CSV</option>
  <option value="image">Image</option>
  <option value="text">Text</option>
</select>
```

This allows the user to specify how to interpret an uploaded file [31]. - If no data is loaded yet (`!data.content`): show an upload dropzone. FloraFauna's DataNode displayed a dashed border box with an upload icon and text "Click to upload [dataType]" [32]. You can use an `<input type="file" hidden>` and a `<label>` to style the drop area. For example:

```
<label className="block">
  <input type="file" onChange={handleFileUpload} className="hidden"
         accept={data.dataType === 'image' ? 'image/*' : '*'} />
  <div className="border-2 border-dashed border-floranodus-border rounded p-4
text-center cursor-pointer hover:border-floranodus-accent-primary">
    <UploadIcon ... className="mx-auto mb-2 text-floranodus-text-secondary" />
    <p className="text-xs text-floranodus-text-secondary">
      Click to upload {data.dataType}
    </p>
  </div>
</label>
```

This matches the FloraFauna UI [32]. When the user clicks, it triggers file selection; `handleFileUpload` should read the file (e.g., using FileReader for text, or generating an image preview) and store it in the node's state. - If data *is* loaded (`data.content` exists): show a preview. FloraFauna's DataNode toggled between a short message "Data loaded (json)" and a preview mode that displays the content or image [33] [34]. Implement a `showPreview` state to toggle: - If `showPreview` is false: just show a line indicating data is loaded (perhaps the file name or type). - If true and the data is text/JSON: show a `<pre>` block with the JSON stringified or text content [35]. - If it's an image, you could render an `<img>` preview (or for simplicity, just state "Image loaded"). Provide a "Show Preview/Hide Preview" button to toggle this [36], and maybe a Download button (FloraFauna had a small download icon button to export the data) [36]. - Use a **Database** icon (or similar) for the DataNode's icon. The lucide-react set has `Database` which was likely used [37].

Make sure to call `BaseNode` with `type: 'data'` and the label (e.g., "Data Input") and description if any. For example:

```
<BaseNode {...props} data={{ ...data, icon: Database, type: 'data' }}>
  ...children as above...
</BaseNode>
```

This ensures the node has the correct styling and shows the database icon and label [38] [39].

Figma sync considerations: A DataNode might not need a corresponding Figma design (it's more functional). However, you *could* create a Figma component for it if you want the visual style mirrored. It would be a card with a dashed box or some representation of data input. For initial integration, focusing on the design nodes and general style might suffice – syncing every functional node to Figma isn't strictly necessary. But the option is there if desired (we'll discuss how to sync components to Figma in Section 5).

4.3 OllamaNode – Local LLM Text Generation

File: `src/components/nodes/OllamaNode.tsx`
Purpose: Provide a UI for prompting a local large language model (LLM) and getting a generated text response. Uses the Ollama backend (which must be running locally).

UI Elements: - **Model Selection:** A dropdown listing available local models. On mount, the component should call `ollamaService.listModels()` to fetch model names (e.g., "Llama2 7B", "GPT4All", etc.). Populate a `<select>` with these. FloraFauna's code did this and also showed a **Pull Model** button if a model is selected but not yet downloaded [40] [41]. You can replicate that: - If `data.model` is set and it's not in the list of `models` (meaning the model isn't downloaded), show a small button (with a download icon) that calls `ollamaService.pullModel(modelName)` to download it [41]. Indicate progress (Ollama can report download percent) – FloraFauna tracked `pullProgress` state and rendered a progress bar while pulling [42] [43]. - **System Prompt and User Prompt:** Two text areas – one for an optional system prompt (instructions to the AI, like persona or context) and one for the user's prompt. Keep them small (maybe 2-3 rows) with placeholders like "System prompt (optional)..." [44] and "Enter prompt..." [45]. Bind their values to `data.systemPrompt` and `data.prompt` respectively (likely managed via state or store). - **Parameters:** FloraFauna included a **Temperature** slider to adjust randomness [46]. You can include an input type="range" from 0 to 2 (with step 0.1), bound to `data.temperature`. Show the numeric value beside it [46]. - **Image input indicator:** For multimodal LLMs (like LLaVA or Vision-enabled models), the OllamaNode accepted an image input. In the UI, if the chosen model name includes `"llava"` or `"vision"`, FloraFauna displayed a placeholder for an image input handle [47] [48]. In their code they show a small div with an image icon and text "Connect image input" as a visual cue inside the node [49]. You should also add a second target handle for images (besides the main prompt handle). We'll cover handles in a moment. - **Generate/Stop Buttons:** A primary button to run the generation. If no prompt or model is selected, it should be disabled. When clicked, call an `ollamaService.generate(model, prompt, options, callback)` method. FloraFauna's implementation streamed tokens to update the `response` state as they arrived [50]. You can do something similar: append tokens to a `response` string state to simulate streaming. While generating, show a "Stop" button (to allow cancellation) [51] [52]. The button label toggles from "Generate" (with a bot icon) to "Stop" (with a stop icon) depending on `isGenerating` state [53]. Also possibly disable the prompt fields while generating. - **Response Display:** When a response is received, display it in a scrollable area. FloraFauna used a `<pre>` with class to preserve whitespace, inside a `<div>`

with max height and overflow scroll [54]. Add this below the inputs, and auto-scroll to bottom as new tokens append (they did this with a ref and useEffect).

- **Handles (Connections):** To allow connecting other nodes:
- Left-side **"prompt"** handle: so that an upstream node (e.g. a Data node containing text, or another LLM node) could supply the prompt. Give it an `id="prompt"` and position it normally on the left [55]. In React Flow, you might handle incoming connections by updating `data.prompt` when something is connected; for now, the presence of this handle is enough for the UI.
- Left-side **"image"** handle: if the model supports images, include a second target handle at a lower position (e.g. 70% from top) and perhaps a different color to distinguish (FloraFauna used `accent-secondary` for image inputs) [56]. Give it `id="image"`. This will allow an Image node's output to feed in. You don't have to implement the logic fully yet (unless you connect it to ComfyUI output), but structurally it should be there.
- Right-side **"response"** handle: an output handle with `id="response"` to send the generated text to other nodes (e.g., maybe to a design node that displays text, or another processing node) [57].

Use BaseNode to wrap this UI: e.g.,

```
<BaseNode {...props} data={{ ...data, icon: Bot, type: 'ai', status:
isGenerating ? 'processing' : (response ? 'success' : 'idle') }}>
  ... (inputs, texts, buttons as above) ...
  <Handle type="target" position={Position.Left} id="prompt" ... />
  {modelSupportsImages && <Handle type="target" position={Position.Left}
id="image" style={{ top: '70%' }} ... />}
  <Handle type="source" position={Position.Right} id="response" ... />
</BaseNode>
```

In this snippet, we dynamically set the node's `status` to "processing" when generating and to "success" if a response is available [58] [59]. We also include the handles inside BaseNode (so they render in the node UI) [60] [61].

Service Integration: Implement an `ollamaService` similar to FloraFauna's: - If not already present, create `src/services/ollamaService.ts`. This should handle: - Listing models: possibly by calling `GET http://localhost:11434/models` or using Ollama CLI (depending on Ollama's API). In FloraFauna, `listModels()` was used on mount to populate the dropdown [62] [63]. - Pulling a model: likely `POST /models/pull` or using `ollama pull <model>`. FloraFauna's UI expected a progress callback [64]. - Generating text: Ollama offers a streaming API (e.g., `POST /generate` with events). The service can wrap this and invoke a callback for each token (as FloraFauna did with `(token) => setResponse(prev + token)` on each chunk [50]). Also provide a `cancelGeneration()` to abort (perhaps calling an AbortController or Ollama's cancel if available) [65] [51]. - **Note:** Ensure the Ollama service base URL or CLI usage is configured. Possibly set `VITE_OLLAMA_URL` in `.env` (for example, default to `http://localhost:11434`). If needed, update accordingly.

Figma Sync: The OllamaNode is primarily a functional node, so you might not need to mirror its internals in Figma. However, the overall node style (the card with title "Local LLM", etc.) is part of the design system. You could create a Figma component for it (a container with a bot icon and some dummy text fields), but it's optional. The styling is already consistent via BaseNode. Focus on Figma sync for the DesignNode (covered later) and maybe overall canvas layout.

4.4 ComfyUINode – Image Generation Component

File: `src/components/nodes/ComfyUINode.tsx`
Purpose: Interface with ComfyUI (Stable Diffusion) to generate images from text prompts (and possibly image inputs for img2img or inpainting in future).

UI Elements: - **Model Selection:** Similar to Ollama, list available diffusion models. ComfyUI can provide a list of installed models. The `comfyuiService.listModels()` should fetch model names (FloraFauna attempts to read them from the Checkpoint loader node info [66]). Populate a dropdown with options like "SDXL Base", "SDXL Turbo", "DreamShaper", etc. Provide some defaults if list fetch fails [67] . The component's `data.model` holds the selected model filename (e.g. `sd_xl_base_1.0.safetensors`). - **Prompt Input:** A textarea for the prompt describing the image. Keep it a bit smaller (maybe 2 rows) since image prompts are often one line. Use placeholder "Describe the image..." [68] . - **(Optionally) Negative Prompt:** FloraFauna did not explicitly surface negative prompts in the UI, but they hard-coded one ("text, watermark") in the workflow builder [69] . For a manual guide, we might skip negative prompt UI for now to keep it simple, unless you want to add another textarea. - **Image Preview:** Once an image is generated, display it. FloraFauna's node, upon generation completion, created a blob URL for the image and set `imageUrl`. In the UI, if `imageUrl` exists, show an `<img>` with that source [70] . Wrap it in a container with perhaps a download button overlay (they had a small download icon at the corner to save the image) [71] . Style the image with a border and rounded corners to match the node card. - **Generate/Stop Buttons:** Like the LLM node, provide a button that toggles between "Generate" (with a magic wand icon) and "Stop" (with a stop icon) [72] . Disable it if no model or prompt is set. When clicked: - On Generate: Construct a workflow for ComfyUI and submit it. FloraFauna's `comfyuiService.buildText2ImageWorkflow(prompt, model)` returns a JSON graph for ComfyUI's API [73] [74] . Then they call `queueWorkflow(workflow)` to enqueue it [75] , then poll `getHistory(promptId)` every second until outputs are ready [76] . You would implement similar logic: basically poll until an image file is available, then fetch that image via `getImage(filename)` [77] . - On Stop: call `comfyuiService.cancelGeneration()` to abort (FloraFauna likely uses an AbortController to cancel the fetch loop) [78] . - While generating, you can show a progress bar or at least change node status to processing (which BaseNode will indicate with a pulsing border). FloraFauna tracked a `progress` percentage but in their simple implementation they set it to 0 or 100% only (since ComfyUI API might not give intermediate progress easily) [79] [80] . You could update progress if the history indicates it (or omit the progress bar to start). - **Image Input indicator:** If a model that supports image conditioning is selected (none of the default SD models do, except maybe if using ControlNet or something), you might include a second input handle for an image. FloraFauna reused the `modelSupportsImages` check (which was really for LLMs) here too [81] [49] , but for standard SD models this won't apply. You can include the UI cue "Connect image input" similarly for consistency, or skip it. It might be forward-looking if integrating an *img2img* model. - **Handles:** - Left **"prompt"** handle: allow connecting a Text node or other prompt source to this image generator. Define `id="prompt"` on a target handle [82] . - Left **"image"** handle: if supporting image inputs, add `id="image"` target (position it lower on the left) [83] . - Right **"image"** handle (output): on the right side, provide a source handle with `id="image"` (or maybe `"output"` but since it's an image it could be labeled image) [84] . This will allow the generated image to be passed to other nodes (for example, an LLM node that can describe images, or a future image processing node).

Wrap all this in `<BaseNode ... type:'ai' icon: Image ...>` (use an appropriate icon, perhaps a picture icon). Set the status to processing when `isGenerating` is true, and to success if an `imageUrl` is available (meaning generation finished) [85] [86] . For example:

```
<BaseNode {...props} data={{
      ...data, icon: ImageIcon, type: 'ai',
      status: isGenerating ? 'processing' : (imageUrl ? 'success' : 'idle')
}}>
  ...UI as above...
  <Handle type="target" position={Position.Left} id="prompt" ... />
  {modelSupportsImages && <Handle type="target" position={Position.Left}
id="image" style={{ top: '70%' }} ... />}
  <Handle type="source" position={Position.Right} id="image" ... />
</BaseNode>
```

Service Integration: Implement `comfyuiService` (if not already in Floranodus). From FloraFauna, key methods are: - `listModels()` – calls ComfyUI's `object_info` endpoint to get available checkpoint names [66] . - `buildText2ImageWorkflow(prompt, model)` – returns a JSON object representing the pipeline (seed -> model loader -> empty image -> text encode -> KSampler -> VAE decode -> SaveImage) [73] [87] . - `queueWorkflow(workflow)` – `POST /prompt` to start generation, returns a prompt_id [75] . - `getHistory(promptId)` – GET `/history/{promptId}` to check if outputs are ready [76] . - `getImage(filename)` – GET `/view?filename=...` to fetch the image blob [77] . - Also a `cancelGeneration()` to abort (setting an internal abort on fetch) [88] [89] .

Set the base URL for ComfyUI (default is `http://localhost:8188` as in FloraFauna [90]) and ensure ComfyUI's server is running with API enabled.

4.5 FlowiseNode – AI Workflow Executor

File: `src/components/nodes/FlowiseNode.tsx`
Purpose: Integrate with Flowise (an open-source LangChain UI) to run pre-built conversational or data processing flows. This node allows selecting a chain and executing it with an input, returning an output (often text).

UI Elements: - **Chain Selection:** On mount, call `flowiseService.getChains()` to retrieve available flows (each with an id and name). Populate a `<select>` with the chain names [91] . The node's `data.chainId` holds the selected chain's ID. - **Settings/Deploy (Optional):** FloraFauna's UI had a settings icon button, possibly to open Flowise or chain settings (they show a Settings icon in the header) [92] . This was not fully functional in the snippet, so you may omit or leave it static. - **Execute Button:** Once a chain is selected (`data.chainId` is set), show a button to run it [93] . The label toggles between "Execute" and "Executing..." with a loader icon while running [94] . When clicked, call `flowiseService.executeChain(chainId, input)`. FloraFauna hard-coded an input `"Test input"` when calling execute for demonstration [95] . You can improve this by allowing an input: perhaps connect something to the left handle or provide a small text field. For simplicity, you could use a default test prompt or the connection mechanism. - **Output Display:** After execution, show the result. In FloraFauna, they captured the result (which could be JSON or text) and did `setOutput(JSON.stringify(result, null, 2))` [95]. Then they displayed it in a small scrollable `<pre>` area [96]. Implement similar: a `<pre>` with class for small text showing the JSON or string output. - **Handles:** - Left **"input"** handle: allow something like a Text node to feed into the Flowise chain's input. Define `id="input"` on a target handle [97]. - Right **"output"** handle: to output the chain's result to other nodes (e.g., feed into a Design node for display). Define `id="output"` on a source handle [98].

Wrap in `<BaseNode ... icon: Workflow (a flow chart icon) type:'ai' ...>` and set status to processing while executing, and maybe success if `output` is non-empty and the chain is marked as deployed [99]. FloraFauna indicated a `data.deployed` status to color the node when a chain is deployed (green border for success) [99]. If using that, you can update `data.deployed` after a successful execute or if you call a deploy endpoint. To keep it simple, consider status success after getting output.

Service Integration: `flowiseService` should handle: - `getChains()` – GET from Flowise API (likely `GET /api/v1/chains`) to list chains with id and name [100]. - `executeChain(chainId, input)` – POST to Flowise's predict endpoint (`POST /api/v1/prediction/{chainId}`) with a JSON payload containing the input question [101]. - (Optional) `deployChain(chainId)` – Flowise requires deploying a chain to get a persistent endpoint [102]. FloraFauna might have assumed chains are pre-deployed or auto-deployed in dev. You may call `flowiseService.deployChain(id)` after selection if needed (which calls `POST /chains/{id}/deploy`) [103], so that execute works. Check Flowise docs for whether execution works without explicit deploy in the latest version. - The base URL default is `http://localhost:3001` [104], and an `API_KEY` can be set if Flowise requires auth [105] [106].

4.6 AIProcessingNode – Generic AI Processor (Optional)

FloraFauna included a generic `AIProcessingNode` which appears to be a placeholder for an AI operation that isn't specifically defined (it had a model dropdown with dummy values and a Process button that just logs to console) [107] [108]. You may not need to prioritize this if you have Ollama and others working. However, implementing it can be straightforward: - It would be similar to OllamaNode UI (model select, prompt textarea, a button). But since we have real specialized nodes, this might remain as a stub for future custom integrations. - If you include it, register it as type `'aiProcessing'` (which is already in nodeTypes mapping [109]). - It can reuse the Brain icon and simply indicate something like "Process" with a spinner animation on click, without real functionality (or perhaps later connect to a generic AI API).

4.7 DesignNode – Figma Design Sync Component

File: `src/components/nodes/DesignNode.tsx`
Purpose: Represent a UI element from Figma on the canvas, enabling synchronization between the design in Figma and the app. This is crucial for the design system integration: it's how a designer can drop a Figma-created element into the AI flow, or how the app can spawn a design in Figma.

UI Elements: - When a DesignNode is not yet linked to a Figma frame (`!data.figmaNodeId`): show a prompt to connect. FloraFauna's UI displays a message "No Figma node connected" and a button "Connect Figma Node" [110]. Implement this as a placeholder state. - When it **is** linked (`data.figmaNodeId` set): show details and actions: - A label "Figma Node" and an **Open in Figma** link. FloraFauna constructed a URL to the Figma file using the stored `figmaFileKey` and `figmaNodeId`, e.g.:

```
<a href={`https://www.figma.com/file/${data.figmaFileKey}?node-id=$
{data.figmaNodeId}`} target="_blank">
  Open <ExternalLinkIcon/>
</a>
```

This opens the exact component in the Figma web app [111]. Add a small external-link icon for clarity. - A **"Sync from Figma"** button: Clicking this should fetch the latest design from Figma and update the app's representation. In FloraFauna, they wired this to call the hook's `syncFromFigma(fileKey, nodeId)` function [112] [113]. We will set that up soon. The button should indicate when sync is in progress (e.g., disable and maybe show a spinner icon). FloraFauna used a rotating refresh icon during syncing [114]. - A timestamp or note about last sync: e.g., "Last synced: 10:30:15 AM" as in FloraFauna [115]. This helps know if the design is up-to-date. - **Connect Button Behavior:** The "Connect Figma Node" button (when no nodeId) is essentially to **push a new design** from the app to Figma. This should trigger the `syncToFigma` process. You will need to implement an onClick for this button that: 1. Constructs a `designData` object describing what to create in Figma. For example, you might pass an object like `{ name: data.label || "New Component", type: 'FRAME', width: 300, height: 150, ... }` or more simply, some identifier that the Figma plugin's AI can use to generate a component. FloraFauna's `figmaService.createDesignNode(designData)` will send this to the Figma MCP server [116]. 2. Calls `useFigmaSync().syncToFigma(nodeId, designData)`, which in turn uses `figmaService.createDesignNode` and then updates the node's data with the returned `nodeId` and `fileKey` [117]. 3. After this call, the node will get a `figmaNodeId` and `figmaFileKey`, effectively "connecting" it to an actual Figma element. Then you can render the Open/Sync UI.

Wrap the DesignNode UI in `<BaseNode ... type:'design' icon: Figma ...>` (using a Figma logo icon if available in Lucide, or a generic icon). The content inside BaseNode will be the conditional blocks described above [110] [118].

Using useFigmaSync Hook: The `useFigmaSync` hook we set up earlier gives us two crucial functions: `syncToFigma(nodeId, designData)` and `syncFromFigma(fileKey, nodeId)` [117] [119]. Here's how to use them: - **On Connect (Sync** to **Figma):** When the user clicks "Connect Figma Node": - Prepare a minimal `designData` for the new component. This could be as simple as `{ name: data.label || "Design Element" }` and perhaps some default frame properties. Optionally, you could include the current node's dimensions or style if you have any. (Since our nodes are not pixel-measured, you might rely on Figma AI to fill details.) - Call `syncToFigma(nodeId, designData)`. This will send the request to the bridge. The bridge (with its AI generation tools) will create a new frame/component in the Figma file, applying the design system and any AI magic to flesh it out. The result returned includes `result.nodeId` and `result.fileKey` [117]. - The hook automatically updates the canvas node with these identifiers and a timestamp [120]. After this, the DesignNode component will re-render now showing the Open and Sync buttons (because `data.figmaNodeId` is set). - Immediately check the VSCode Figma panel or Figma app – you should see a new component created (perhaps named after the label you gave). This is a one-click way to generate UI from code. - **On Sync** from **Figma:** When the user clicks "Sync from Figma": - Call `syncFromFigma(fileKey, nodeId)` via the hook. This will request the bridge to fetch the specified Figma node (or the whole file's components). The hook then converts any returned Figma nodes into canvas nodes in the app. In FloraFauna, they looped through `figmaData.nodes` and for each, added a new canvas node of type 'design' with the same name and type info [121] [122]. This means if the designer duplicated or created multiple components in Figma, they'd all appear on the canvas. - In our context, for a single DesignNode, you might use this to **update** that node's style or to import new elements. For instance, if you resized or recolored the component in Figma, after sync you could update the corresponding CSS variables or component state. In practice, the current implementation of `syncFromFigma` in the hook adds new nodes rather than updating the existing one (it uses `addNode` with a new ID `figma-...` [123]). You might modify it to instead update if IDs match. But an easy approach: - If you want to import all new Figma components as separate nodes, just let it add nodes. You'll see new DesignNodes appear for each component. You can then manually link them or use them. - If you specifically want to refresh the one component, you might filter the result for that nodeId. - Also run `pnpm figma:sync` after to pull any design token changes. If in Figma you adjusted colors or spacings that are part of your design system, this command will update the `design-tokens.css`. The app (with hot-reload) will then apply those new values.

Testing the DesignNode: - Create a DesignNode on the canvas (the initial nodes include one by default labeled "Figma Design" [124]). It will say "No Figma node connected". Click **Connect Figma Node**. Within a couple of seconds, check the Figma panel – a new frame/component should appear, perhaps a placeholder styled card. The Floranodus Figma Bridge uses AI to generate a reasonable UI element (it might be a styled container with the label text, etc.). - Back in the app, the node now shows "Figma Node" with an Open link. Click **Open** to jump to that component in the Figma web app [111]. Verify its design. - Try editing the design in Figma (for example, change the fill color or text). Then back in the app, click **Sync from Figma**. This should create or update nodes. If a duplicate node appears (with prefix `figma-...` ID), that's the imported one. You can decide to use that going forward (and perhaps remove the old). In a future improvement, the logic could match by name to update, but for now, you have a copy. - Notice any updated styles: if you changed a color to a new one, run the token sync to extract it. Then your Tailwind classes or CSS variables (like `--figma-node-bg`) might update, affecting all nodes globally.

5. Incorporating the Node Editor into Floranodus App

Now that components are created, integrate them into the Floranodus application's UI:

- **Register Node Types:** In Floranodus, there should be a central definition mapping node `type` strings to the React components. In FloraFauna's monorepo, this is done in a file like `src/utils/nodeTypes.ts` [2]. Add all the new components to this map. For example:

  ```
  export const nodeTypes = {
    aiProcessing: AIProcessingNode,
    design: DesignNode,
    data: DataNode,
    flowise: FlowiseNode,
    comfyui: ComfyUINode,
    ollama: OllamaNode,
  };
  ```

  Ensure this is used by the ReactFlow `<ReactFlow nodeTypes={nodeTypes} ...>` prop [125]. This allows React Flow to render the correct component for each node's `type` field.

- **Define Initial Nodes/Edges:** Optionally, set up some starting nodes to show the user. FloraFauna's example initial graph included an AI Processor node, a Figma Design node, and a Data node, with edges connecting data -> AI -> design [126] [127]. You can replicate this:
- Data Input node (id "3") feeding into AI node (id "1"), and AI node feeding into Design node (id "2") [128]. This demonstrates a simple pipeline.
- Adjust the types if you prefer to use Ollama or ComfyUI instead of the generic aiProcessing for the first node. For example, you could start with an Ollama node (type 'ollama') as id "1" with a default model, and a Design node as id "2", etc.
- These initial nodes can be defined in a similar `nodeTemplates` or `initialNodes` constant and fed to `useNodesState(initialNodes)` in the canvas component [129] [130].
- **Canvas Component:** Floranodus likely has a `FloranodusCanvas.tsx` that sets up the ReactFlow canvas. If not, create one in `components/canvas/`. Based on FloraFauna's:

```
<ReactFlow nodes={nodes} edges={edges}
           onNodesChange={onNodesChange} onEdgesChange={onEdgesChange}
           onConnect={onConnect} nodeTypes={nodeTypes} fitView>
  <Background variant={BackgroundVariant.Dots} ... />
  <Controls className="bg-floranodus-bg-secondary ..." />
  <MiniMap ... />
  <Panel position="top-left"><CanvasControls/></Panel>
  {isFigmaConnected && <Panel position="top-right"> ... "Figma Connected"
badge ... </Panel>}
</ReactFlow>
```

This sets up:
- **ReactFlow** container with our nodes, edges, and the event handlers to manage state (you can use `useNodesState` hook to get stateful node list and an `onNodesChange` callback, similar for edges [129]).
- **Background** grid (nice dotted grid for visual aid).
- **Controls** (zoom in/out, etc.) and **MiniMap** – these come from React Flow library. The CSS classes applied ensure they use our theme (see how `.react-flow__controls` and `.react-flow__minimap` classes were styled with floranodus colors [131] [132]).
- **Custom Panels:** Use ReactFlow's Panel to overlay any UI on the canvas. We have two:
    - Left panel: the `CanvasControls` (toolbar with add/save/upload buttons).
    - Right panel: a Figma status indicator, shown only if the Figma bridge is connected. In FloraFauna, they rendered a green dot with text "Figma Connected" [133] using a success color from the theme.
- **Node addition logic:** The `CanvasControls` "+" button currently doesn't open a menu in code (it's a stub with no onClick). You should implement it to add a new node. For example, when + is clicked, you could open a small overlay or context menu showing categories of nodes (using the `nodeTemplates` list we saw in code [134] [135]). For this manual guide, if you want a quick solution:
    - Simply trigger adding a default node, e.g., always add a Data node or a specific node for now to test. Use the zustand store or `setNodes([...nodes, newNode])`.
    - Or integrate a library for a menu. A simple approach: maintain a local state like `showNodeMenu` and if true, render a small div under the plus button with options. Each option onClick calls `addNode(newNode)` to the store.

    - The nodeTemplates array from FloraFauna provides a structured list of node types with default data and labels [3] [136] which you can use to populate this menu. That ensures you insert nodes with sensible defaults (like an Ollama node with a default model value).

- **Edge connection logic:** The `onConnect` handler should add a new edge when the user connects two nodes on the canvas. React Flow's `addEdge` utility can be used as shown [137]. This simply takes the connection params (source, target, etc.) and appends to the edges list. The default edges are straight lines; FloraFauna styled them via CSS to match the theme (the `.react-flow__edge` classes get border color) [138], and set `animated: true` on initial edges for a nice flowing effect [128]. You can animate or style edges as you like (dotted lines, curves, etc., by customizing ReactFlow props or CSS).

- **Navigation and Layout:** The Floranodus app likely has routing (login page -> canvas page). Ensure that after login (when `navigate('/canvas')` is called [139]), the app loads the canvas component where all these nodes live. If using React Router, define a route for `/canvas` that renders `FloranodusCanvas` (which in turn includes all nodes). Also, if you have a top navigation bar or sidebar in your app, you can integrate that with the canvas:

- A simple **Navigation Bar** could be a fixed header with the app name "Floranodus" and maybe a user profile or sign-out button (since Google auth is used). This is not strictly part of FloraFauna's AI interface, but for completeness, you might create a `NavigationBar` component and include it above the canvas. It could also host a toggle for a settings panel or display the project name.

- A **Settings Panel** (as mentioned in your prompt) could be a sidebar to configure global settings (like turning on/off certain services, adjusting API keys, etc.) or to show logs (Console Ninja outputs). If you plan to include one, you can create a component (e.g., `SettingsPanel`) and render it conditionally. For now, you might skip actual settings and just leave a placeholder where it would go (maybe a button in the nav to open a drawer).

- **Console Ninja Integration:** FloraFauna monorepo also integrates Console Ninja for debugging (as indicated by the MCP config [140]). While not requested explicitly, be aware that if it's running, you can see console logs from the app directly in Cursor. Use this to debug your node actions (e.g., log the response from an AI service, etc.). It's a helpful dev feature, especially when testing the flows.

At this point, you should have the **full UI framework** in place in Floranodus: - The user logs in via Google, the app navigates to the canvas. - The canvas displays an infinite grid with some initial nodes. - The user can click the **+** button to add new AI nodes, connect them, configure prompts, and run inference. - The **Figma integration** is live: at least one DesignNode is present and you can generate designs in Figma or import from Figma.

6. Synchronizing React Components with Figma (Round-Trip Workflow)

A critical part of this integration is setting up a smooth **round-trip design workflow** between React (code) and Figma (design). Here we highlight best practices and the typical sequence for using the Unified Figma Bridge with your components:

6.1 Creating React Components First (Code-First Approach)

The recommended approach is **code-first**: define the structure and props of your React component, then use Figma AI to generate the visual design, and finally link them.

- **Start with the React code:** As we did above, create the component files, define their JSX structure and include any dynamic elements (state, etc.). Focus on functionality and using semantic elements (buttons, inputs, etc.) as needed.
- **Use consistent naming:** The name/label you give in the component can be used to identify it in Figma. For example, our DesignNode uses `data.label` which might be "Figma Design". If you pass that as the name in `designData` when calling `syncToFigma`, the Figma component will be named "Figma Design". This helps you trace which Figma frame corresponds to which component.
- **Keep styling minimal in code initially:** Rely on basic structure (like layout using flex, etc.) and use design tokens for spacing, but don't hardcode too many pixel values. The idea is to let Figma's AI propose an ideal styling. For instance, you might not decide the exact width of a node card in code – let the Figma generation create a nicely sized frame, then you adopt those sizes if needed.

6.2 Triggering Visual Design Generation in Figma

Once the component logic exists, use the **Unified Figma Bridge** to create a UI design for it: - Ensure the bridge server is running (`pnpm dev:plugin` or the VSCode task). - In Cursor, you have convenient tasks: - Run **"Generate AI Components"** task to generate all core components in Figma automatically [18]. This likely invokes a script that calls the bridge for each component type (navigation bar, node card, etc.). For example, in the quick start, they show generating components like `'navigation-bar', 'ai-node-card', 'connection-line', 'toolbar', 'settings-panel'` in bulk [141]. Running this could instantly populate your Figma file with a full UI kit. - Alternatively, use the **Figma Write MCP plugin via chat**: In Cursor's chat, you could ask something like *"Generate a Figma design for a card with title and description for my AI node."* The MCP tools might interpret and create a design. However, the scripted approach is more deterministic. - Use the **Connect Figma Node** (syncToFigma) method on individual DesignNodes for ad-hoc generation. For example, if you have a new type of node that the bulk script didn't generate, just placing a DesignNode and clicking connect will create it. - When you trigger generation, watch the Figma panel. Within seconds, you should see frames/components appear, with styling that follows your design system (colors, typography) but also layouts determined by the AI (e.g., padding, alignment of elements). - **Example:** You added a new component file for a **Toolbar** (maybe to hold buttons). Instead of manually drawing it in Figma, you run `await workflow.generateAIComponents(['toolbar'])` via the script or task. The Figma bridge will create a "toolbar" frame with likely a horizontal bar design. You then import the design tokens or measurements from that frame into your code (via token sync or by eyeballing and updating your Tailwind classes to match).

6.3 Using Figma Designs in Code

Once designs are generated: - **Extract design tokens:** Run `pnpm figma:sync` to pull updated design tokens CSS [19]. This will update colors, font sizes, spacing variables, etc., in your code if the Figma design introduced any changes. For example, if the Figma AI decided on a new shade for `--figma-accent-secondary`, it will reflect after sync. Use those tokens in your Tailwind classes or styled components so that the app immediately picks up the new styles. - **Copy specific styles:** Sometimes you might want exact pixel values from Figma (like a component's width). You can use the Figma VSCode extension's inspect feature: click an element in the Figma panel to see its CSS properties. You might then set your component's CSS to match (e.g., give your node container a fixed width if needed). The goal is to make your code's output visually identical to the Figma design.

6.4 Importing Visual Changes from Figma to Code (Sync-From-Figma)

If you or a designer makes manual adjustments in Figma: - Use the **Sync from Figma** button on the DesignNode (or run a script to fetch all components). This will **create canvas nodes** for new components and could be extended to update existing ones. In the current implementation, after syncing, you might end up with duplicate nodes if the same component was already on canvas. In the future, consider enhancing `useFigmaSync.syncFromFigma` to merge changes, but for now, treat it as importing fresh copies. - For each imported design node, you get its name, type, and a reference. You can then decide how to use that: - If it's a brand new component that didn't exist in code, you can create a React component stub for it and assign the design to it (essentially reverse of code-first: design-first then code). - If it's meant to update an existing component's style, you can manually apply the differences. For example, say you have a Button component in code and in Figma you changed its border radius from 4px to 8px. After syncing, a "Button" design node appears with radius 8. You then know to update your code's styling (or token) for button radius to `--figma-radius-md` (which might now be 8px instead of 4px). In fact, if that radius is part of the tokens, simply syncing tokens would do it globally. - Remove or archive any duplicate nodes to avoid confusion, after you've taken the necessary info from them.

Round-Trip Example: Let's walk through a concrete scenario: - You want to create a new **Audio Player node** (just as an example for multimodal extension). You make a new component `AudioNode.tsx` with BaseNode, label "Audio Player", maybe a play button in it. Initially it's plain. - In VSCode, run the task to generate an AI design for "Audio Player". The Figma bridge creates a nice Audio Player frame (with a play icon, timeline, etc.). The DesignNode updates with a Figma ID. - You import any new design tokens and adjust your AudioNode code to include, say, a play icon (matching what Figma put), and ensure classes match (maybe Figma gave it a specific layout). - Later, a designer opens the Figma file and tweaks the Audio Player design – changing the icon color and adding a volume slider. You hit **Sync from Figma**. A new design node or updated data comes in for "Audio Player". - You then update your `AudioNode.tsx` to include a volume slider element and use the new color (which was synced to your tokens). Now code and design are realigned.

Throughout this, keep the Cursor Console Ninja open to catch any errors (for example, if a service call fails or bridge is not connected, those errors will log in the console). The integration can be iterative – design and code informing each other.

7. Maintaining Authentication and Service Configurations

With the new architecture in place, ensure it **plays well with authentication and existing configs**:

- **Google Sign-In Flow:** Floranodus uses Google OAuth for login, via the `@react-oauth/google` component on `LoginPage.tsx`. Continue to use this as the entry point of the app. The login page should remain untouched in terms of logic. Upon successful login, `handleGoogleSuccess` navigates the user to `/canvas` [139]. Confirm that this route now displays your integrated canvas and nodes. There should be no conflict: all the new components are client-side and behind the login, so Google OAuth remains the gate. One thing to check is token handling – the `authService.handleGoogleCallback` likely stores a token or calls backend. Ensure any new network requests you added (to local AI services or Figma) do not require being authenticated via that token (they shouldn't; they're local). They should also not interfere with it (different ports and endpoints entirely, so it's fine).

- **Protecting Routes:** If not already, you might want to protect the `/canvas` route so it's not accessible without login. Possibly the app already does this (e.g., checks a global auth state). Just keep in mind as you integrate that you shouldn't accidentally expose the canvas before login. If using React Router, you can add a redirect on no auth.
- **Open-Source Auth vs AI Services:** The Google Sign-In is for your app's user authentication; it doesn't directly affect services like Ollama or ComfyUI. Those are running locally without auth. So there's no direct integration needed between Google auth and the AI nodes. The only point of potential conflict is port usage: e.g., if you run ComfyUI on 8188 and Flowise on 3001, that's fine. If your frontend dev server is also on 3000, ensure none of these collide. (The Figma bridge we saw might default to 3000 as well [16], but in the config they used 3000 for Figma and separate tasks for others – you might have changed or need to change some ports).
- **Environment Variables:** Make sure to add any required environment configs for the new services:
- In `packages/floranodus-app/frontend/.env` (or wherever Vite reads them), add `VITE_OLLAMA_URL`, `VITE_COMFYUI_URL`, `VITE_FLOWISE_URL` as needed, or use the defaults coded in the services. Floranodus likely already has entries; double-check `ollamaService.ts` default (if it's similar to ComfyUI's pattern).
- Figma plugin env: already set with FIGMA_API_TOKEN and FILE_ID – nothing new to add unless you change file.
- **Service Processes:** Remind the user (or set up in documentation) that to use the multimodal nodes, they must have the respective services running:
- **Ollama:** Install Ollama and run `ollama serve` (if required; newer versions might auto-start on usage). Ensure models are installed or use the UI's pull button to fetch them.
- **ComfyUI:** Run the ComfyUI backend (usually running the `run_comfyui.bat` or `python main.py` in ComfyUI directory). Ensure it's listening on the port your service expects (8188 by default).
- **Flowise:** Start the Flowise server (e.g., `npx flowise start` or a Docker) on port 3001 and get some chains ready.
- Optionally, any other service (if you later add, say, Whisper for audio, you'd note that too).

These services don't require auth tokens from Google, they are separate local tools. So there is no interference with your OAuth, which is good. The integration just needs to ensure the frontend knows where to call (hence env variables or defaults).

- **Concurrency and Performance:** Because all these run locally, be mindful of resource usage. Running a large model in Ollama and a diffusion model in ComfyUI simultaneously can be heavy. In testing, try one at a time to avoid stressing the machine. Also, use the PerformanceMonitor component (if included – I saw a `PerformanceMonitor.tsx` reference in the codebase) to watch app performance if needed.

8. Example End-to-End Usage Scenario

Finally, let’s tie everything together with an illustrative scenario to ensure all aspects are covered:

Step 1: Log In and Open Canvas – The developer/user starts Floranodus, signs in with Google, and is presented with the AI Canvas (dark background infinite grid). The top navigation shows “Floranodus” and maybe user info. On the canvas are a few example nodes: e.g., a *Data Input* node, an *AI Processor* node, and a *Figma Design* node connected in a chain. The Figma status indicator in the top-right shows “Figma Connected” (since our bridge is running) [133].

Step 2: Add Nodes and Connect – The user wants to create an AI image generation flow. They click the **Add Node** button on the toolbar. A menu pops up with categories “AI Processing”, “Creative”, “Data”. Under "Creative", they choose **Image Generation** [135]. The app calls `useCanvasStore.addNode` to add a new node of type 'comfyui' at a default position. A new *ComfyUINode* appears, titled “Image Generation” with a magic wand icon. They drag it next to the Data node. - Next, they connect the Data node’s output handle to the Image Generation node’s prompt handle (draw a line from Data’s right handle to ComfyUI’s left handle). An edge is created and appears as an arrow connecting the two nodes [128]. - They also want to feed the image output into something else. They add another node via , under "AI Processing" choose **Local LLM** (Ollama) [3]. A *OllamaNode* “Local LLM” appears. They connect the Image Generation node’s output (image) handle to the LLM node’s image input handle (so the LLM can describe the image, for example). They also connect the Data node’s output to the LLM node’s prompt input (maybe the data node had some text to prompt the LLM). - Now they have a small graph: Data -> (text) -> LLM, Data -> (prompt) -> ImageGen -> (image) -> LLM. This demonstrates multimodal: the LLM will get both a prompt and an image.

Step 3: Configure Node Parameters – The user uploads a file on the Data node (say an image or a JSON with a caption). The Data node now shows “Data loaded (image)” and the preview option [35] [142]. On the Image Gen node, they select a model (e.g., "SDXL Base") and enter a prompt like *"A scenic mountain landscape at sunrise."* On the LLM node, they select a model (perhaps a vision-enabled model like “LLaVA”) and maybe set temperature.

Step 4: Trigger AI Operations – They click **Generate** on the Image node. The node status changes to “processing” (border pulses) and the ComfyUI service is called. After ~5-10 seconds, an image appears on the node. The node status becomes “success” with a green border [85], and a thumbnail is displayed inside it [70]. They then click **Generate** on the LLM node. The Ollama service runs, taking the image (the node knows an image was connected) and prompt, and outputs a description text. The LLM node streams the text in the response box. Once done, its status shows success (green border) [58] and the full description is visible. - If any issues occurred (say the model wasn’t downloaded), the user could click the pull icon on the LLM node to download it [41], watch the progress bar fill up [43], then retry.

Step 5: Sync a Design to Figma – Now the user wants to make the output more visual. They decide to create a design element to display the LLM’s text (like a nicely styled quote box). They add a **DesignNode** from the "Creative" category (labelled "Figma Design") [5]. They connect the LLM node’s **response** output to this Design node’s input handle. The DesignNode currently says “No Figma node connected”. The user clicks **Connect Figma Node**. The app calls `syncToFigma` for this node, and within a second: - A new component frame appears in the Floranodus Figma file, probably a styled text box. The DesignNode in the app now populates with a Figma icon, an “Open” link and a “Sync from Figma” button [111] [114]. The label might remain "Figma Design" unless we choose to rename it. - The user clicks **Open**, which launches the

Figma file in the browser focused on that new component [111]. They see the design: perhaps a card with the placeholder text. They adjust the design – e.g., apply a background color, adjust font size, or insert an image icon – to make it look nicer. - Back in VSCode, they run **Design Token Sync** (or click Sync from Figma in the app). This pulls any changed tokens (say the background color they set) into the app's CSS. The DesignNode could also update or a new node appears (if the implementation adds a new one). - Now the DesignNode on canvas reflects the styled component. It might not show the content yet (unless your DesignNode component is made to fetch and display from Figma, which could be an advanced feature). But at least, it's linked – clicking *Open* goes to the updated design, and any style tokens are consistent. If you want, you could extend the DesignNode to actually render an image preview of the Figma design (e.g., using Figma API to fetch a render or using the `figmaService.syncFromFigma` data to draw a rough representation). That's an advanced step; initially, it can just serve as a placeholder that signals "the detailed UI is in Figma".

Step 6: Finalize and Ensure No Conflicts – The user logs out and back in to ensure the login flow still works and that the state resets (you might clear the zustand store or reload initialNodes on login). They verify that none of the new functionalities require them to re-login or cause any security issues (they don't, since it's all local). Google auth token is only used for your app's backend (if any) and is unrelated to Figma or local AI.

This scenario covered adding nodes, connecting them, running local AI, and bridging to Figma designs – which is exactly the combined capability of FloraFauna and now Floranodus .

9. Conclusion and Next Steps

By following this guide, you have manually replicated the FloraFauna.ai architecture within the Floranodus monorepo: - All **key components and UI patterns** (node cards, connectors, toolbars, etc.) are now present in Floranodus, leveraging the same design system. - The **multimodal AI nodes** (text, image, workflows, data) are integrated with local inference engines (Ollama, ComfyUI, Flowise), allowing you to build complex AI-driven flows on the canvas. - The **Unified Figma Bridge** is fully connected, enabling you to **generate UI components from code** and **sync design changes back to code** using the `useFigmaSync` hook and associated buttons [117] [114]. This ensures your React components and Figma designs stay in harmony. - The existing Google Sign-In auth remains the entry point, unaffected by the new features (aside from routing the user to the canvas upon login [139]). There are no conflicts, since all new interactions occur post-login on local APIs. - You have a clear workflow to continue development: design new components in Figma with AI assistance, import them, implement their functionality in React, and iterate.

Best Practices Recap: - **Develop component logic first in React**, keeping styling flexible, then use Figma AI generation to get a polished UI design. Always import the latest design tokens so your theme stays consistent between code and design [143] [10]. - **Use `useFigmaSync` for one-click sync:** don't manually copy design properties if you can automate it. The Connect and Sync buttons are your friends – they encapsulate complex Figma API calls into simple actions. - **Leverage the design system:** Whenever possible, use the CSS variables (e.g., `var(--figma-spacing-sm)`) and utility classes (`text-floranodus-text-secondary`, etc.) provided. This not only speeds up styling but guarantees the app looks like the Figma file [11] [12]. - **Keep an eye on performance:** The node editor can grow complex; avoid re-rendering everything unnecessarily. The use of React Flow and zustand store should handle efficiently, but be mindful if adding heavy computations in render. - **Modularity:** Each node component should remain self-contained (handle its own service calls and state). The canvas mainly orchestrates connections. This modularity means you can add new node types (say an **AudioNode** or a **Database query node**) easily by following the same pattern: create component + service, add to nodeTypes, design in Figma if needed.

You now have a **comprehensive AI design and development platform** inside Floranodus, combining the strengths of FloraFauna's UI framework with Floranodus's infrastructure. Enjoy creating with your new setup – design in Figma, develop in code, and let them sync in real-time for a truly unified workflow!

Sources:

- Floranodus Monorepo Code (FloraFauna integration) – e.g., Node type definitions [2] [4], BaseNode and styling [21] [22], DataNode UI logic [142] [144], OllamaNode and ComfyUINode structures [55] [57], FlowiseNode handles [98], DesignNode Figma integration UI [114] [110], useFigmaSync implementation [121] [117].
- Floranodus README and Figma Workflow Docs – project features and usage of MCP (Figma Bridge) [1] [145]. The README outlines the integrated approach of AI nodes with design token sync [143], and the quick-start shows how code ties into Figma generation [10].
- Code excerpts from FloraFauna (via Floranodus code) demonstrating specific behaviors (these were preserved in the integration and cited above for clarity).

---

[1] [9] [16] [17] [19] [20] [140] [143] README.md
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/README.md

[2] [3] [4] [5] [6] [109] [124] [126] [127] [128] [134] [135] [136] nodeTypes.ts
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/utils/nodeTypes.ts

[7] [8] design-tokens.css
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/design/figma/design-tokens.css

[10] [18] [30] [141] [145] quick-start.md
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/design/figma/quick-start.md

[11] [12] [26] [131] [132] [138] globals.css
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/styles/globals.css

[13] [117] [119] [120] [121] [122] [123] useFigmaSync.ts
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/hooks/useFigmaSync.ts

[14] [116] figmaService.ts
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/services/figmaService.ts

[15] [125] [129] [130] [133] [137] FloranodusCanvas.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/canvas/FloranodusCanvas.tsx

[21] [22] [23] [24] [25] [27] [28] [29] BaseNode.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/nodes/BaseNode.tsx

31 32 33 34 35 36 37 38 39 142 144 DataNode.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/nodes/DataNode.tsx

40 41 42 43 44 45 46 47 48 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 OllamaNode.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/nodes/OllamaNode.tsx

49 68 70 71 72 78 79 80 81 82 83 84 85 86 ComfyUINode.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/nodes/ComfyUINode.tsx

66 67 69 73 74 75 76 77 87 88 89 90 comfyuiService.ts
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/services/comfyuiService.ts

91 92 93 94 95 96 97 98 99 FlowiseNode.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/nodes/FlowiseNode.tsx

100 101 102 103 104 105 106 flowiseService.ts
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/services/flowiseService.ts

107 108 AIProcessingNode.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/nodes/AIProcessingNode.tsx

110 111 112 113 114 115 118 DesignNode.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/nodes/DesignNode.tsx

139 LoginPage.tsx
https://github.com/trevoralexanderrobey/floranodus-monorepo/blob/8054bd8c8e6ba1beb5301c0f7725edcb9d61e035/packages/floranodus-app/frontend/src/components/auth/LoginPage.tsx