This is the critical bridge. To enable "cognitive plasticity"—where the AI can reorganize your thoughts without breaking the document—we must move away from storing files as monolithic JSON blobs.

Instead, we must treat **every paragraph, list item, and heading as an autonomous node** in the graph.

Here is the **Semantic Graph Schema** for LifeGraph OS. It effectively "explodes" a linear TipTap document into a doubly-linked, semantically rich graph structure stored in GUN.js.

Shutterstock

1. The Core Philosophy: "Block-Level Atomicity"

In standard apps, a document is a container holding text. In LifeGraph, a "Document" is just a **view**—a temporary assembly of independent **Block Nodes**.

This allows the AI to pick up a specific paragraph (a Block) from a "Daily Journal" and link it into a "Project Proposal" without duplicating data. The block exists in the graph; the documents are just ordered lists of pointers to that block.

---

2. The Node Schema (GUN.js Data Model)

Every entity in LifeGraph is a **Node**. We use a discriminated union type to distinguish between content and structure.

A. The Atomic Unit: The `BlockNode`

This represents a single TipTap block (e.g., a paragraph).

```
TypeScript
```

```typescript
type BlockNode = {
  // 1. Identity & Type
  _id: string;             // UUID (e.g., "block_8a7f...")
  type: 'text' | 'heading' | 'task' | 'code' | 'image';

  // 2. Content (Mutable)
  content: string;         // The raw text or TipTap JSON fragment
  attrs: object;           // Attributes (level: 1, checked: false, etc.)

  // 3. Structural Edges (The "Spine")
  // These define where this block sits in a linear flow
  prev: Reference<BlockNode>; // Pointer to the block above
  next: Reference<BlockNode>; // Pointer to the block below
  parent: Reference<PageNode>; // The container this currently "lives" in

  // 4. Semantic Edges (The "Brain")
  // These are lateral connections created by User or AI
  references: Reference<BlockNode>[]; // "See also" links
  tags: Reference<TagNode>[];

  // 5. Cognitive Metadata (The AI Layer)
  vector: Float32Array;    // Embedding for semantic search (WebGPU computed)
  summary: string;         // One-sentence AI summary of this block
  last_mutated: number;    // Timestamp for CRDT convergence
}
```

B. The Container: The `PageNode` (The Canvas)

This represents the "Document" or "Canvas." It is light; it mostly holds metadata and a pointer to the `head` block.

```
TypeScript
```

```typescript
type PageNode = {
  _id: string;
```

```
  type: 'page';
  title: string;

  // The entry point to the linked list of blocks
  head: Reference<BlockNode>;

  // Context for the AI
  intent: string; // e.g., "Drafting a physics paper"
  created_at: number;
}
```

---

3. The Structural Logic: "Exploding" TipTap

When a user types into TipTap, we do not save the whole JSON. We intercept the transaction.

The Transformation Pipeline:

1. **Ingestion:** User hits `Enter` in TipTap.
2. **Atomization:** The system identifies the new paragraph as a new unique entity.
3. **Graph Insertion (GUN.js):**
    - Creates `block_new`.
    - Finds `block_prev` (the cursor position).
    - Updates `block_prev.next` -> `block_new`.
    - Updates `block_new.prev` -> `block_prev`.
4. **Embedding (WebGPU):**
    - The text of `block_new` is sent to `Transformers.js` (background worker).
    - A vector (e.g., 384-float array) is calculated and stored in `block_new.vector`.

Why this is powerful:

If you move a paragraph up, you aren't rewriting a megabyte file. You are just swapping two pointers (prev/next) in the graph. CRDT conflicts become trivial to resolve because they happen at the paragraph level, not the file level.

---

4. The AI Reasoning Layer

This schema allows the Agent Runtime to perform **Graph Retrieval Augmented Generation (GraphRAG)**.

When the user asks: *"How does my research on mycelium relate to network topology?"*

1. **Vector Search:** The system scans `BlockNode` vectors for "mycelium" and "network topology" (using cosine similarity inside the browser).
2. **Graph Traversal:** Instead of just returning the text, the AI traverses the **Semantic Edges**:
    - It finds the "Mycelium" block.
    - It sees it is linked (via `references`) to a "Decentralization" block from a year ago.
    - It walks the graph to find the "Network Topology" block.
3. **Synthesis:** The AI retrieves the specific blocks and their connected neighbors to generate an answer, ignoring the irrelevant rest of the documents.

---

5. Temporal Versioning: The "Thought Stream"

To implement the "Git-like" history, we do not version the nodes directly (which would bloat the graph). We version the **Path**.

- **Commit Object:**
- TypeScript

```
type Commit = {
  _id: string;
  timestamp: number;
  author: string; // User DID or 'agent_optimizer'
  message: string; // e.g., "Refactored introduction for clarity"

  // Snapshot of the 'head' pointer at this moment
  snapshot_head: Reference<BlockNode>;

  // The diff (delta)
  changes: JSONPatch;
}
```

-
-

This allows you to drag a slider back in time. The UI simply re-hydrates the TipTap editor by following the linked list starting from the `snapshot_head` of that specific date.

---

6. Implementation Strategy (The Bottleneck)

The hardest part of this implementation is the **TipTap <-> GUN Binding**.

TipTap expects a synchronous, in-memory JSON tree. GUN is an asynchronous, eventually consistent graph.

The Solution: A Custom "Prosemirror-Collab" Provider.

You cannot just save to GUN on save. You must implement a custom CollaborationProvider (similar to how Y.js works with TipTap).

1. **Local State:** TipTap operates on a local Y.js doc (instant feedback).
2. **Sync Worker:** A Web Worker observes the Y.js binary updates.
3. **Graph Mapper:** The Worker translates Y.js changes into GUN graph mutations (updates `next`/`prev` pointers and text fields).

This decouples the UI speed (60fps) from the Graph propagation speed (network dependent).

Summary of the Data Layer

Feature	Implementation	Benefit
Storage	GUN.js Nodes	P2P, Offline-first, Zero-backend.
Granularity	Block-Level (Paragraphs)	AI can remix content without breaking context.
Search	Client-side Vector Index	Privacy-preserving semantic search.
Structure	Doubly Linked Lists	Infinite scrolling documents without pagination.

Next Step

We have the **Architecture** and the **Data Schema**. The next logical step is to define the **Cognitive Agent Runtime**.

We need to define how the *Transformers.js* model actually "acts" on this graph.

Would you like me to design the "Agent Loop"? This would detail how the Local LLM plans a task, queries the GUN graph, executes tools (like Firecrawl), and writes the result back to a Node.

This is the **Cognitive Engine** of LifeGraph OS.

Because we are constrained by local hardware (limited VRAM, single user session) and the browser environment (ephemeral tabs), we cannot use heavy, server-side agent frameworks like LangChain or AutoGPT directly.

Instead, we implement a **lightweight, state-machine-driven runtime** inspired by `LangGraph`, but built natively for TypeScript and Web Workers.

Here is the design for the **Local Agent Runtime**.

---

1. The Runtime Environment: "The Shadow Worker"

The Agent never runs on the UI thread. It lives in a dedicated **Web Worker** (the "Cognitive Worker"). This ensures 60 FPS UI performance even when the GPU is crunching tensor operations.

- **Host:** `CognitiveWorker.ts`
- **Compute:** WebGPU (via `Transformers.js`)
- **Memory:** SharedArrayBuffer (for fast data exchange)
- **Persistence:** GUN.js (writes its own "Thought State" to the graph)

2. The Cognitive Cycle: The OODA Loop

We structure the Agent's behavior as an **OODA Loop** (Observe, Orient, Decide, Act). This is implemented as a recursive function that passes a `State` object through the LLM.

The Agent State Object (The "Context Window")

This object is the only thing the LLM sees. It is persisted to GUN.js after every step to ensure crash resilience.

```
TypeScript

type AgentState = {
  task_id: string;
  original_intent: string; // "Research decentralized identity"

  // The 'Short Term Memory'
  history: ChatMessage[];

  // The 'Working Memory' (Results from tools)
  observations: {
    tool_name: string;
    result: string | JSON;
    timestamp: number;
  }[];
```

```
  // The 'Plan' (Dynamic stack)
  plan_queue: string[]; // ["Search graph for DID", "Crawl w3c spec", "Summarize"]

  status: 'IDLE' | 'THINKING' | 'EXECUTING' | 'WAITING_FOR_USER' | 'DONE';
};
```

---

3. The Execution Flow

When a user gives a command, the system kicks off the **Dispatcher**.

Step 1: Planning (The Router)

The User Input is sent to a small, fast router model (or a specific prompt on the main model).

- **Input:** "Find my notes on React and write a blog post about it."
- **Output (JSON constrained):**
- JSON

```
{
  "strategy": "retrieval_generation",
  "steps": [
    "query_graph('React')",
    "read_nodes(ids)",
    "synthesize_draft('Blog Post')"
  ]
}
```

-
-

The `AgentState.plan_queue` is populated with these steps.

Step 2: Tool Execution (The "Hands")

The Agent pops the top item off the `plan_queue`.

If the step is `query_graph('React')`:

1. **Tool Invocation:** The Worker calls the internal API.
2. **Vector Search:** It runs a cosine similarity search against local embeddings.
3. **Graph Expansion:** It traverses GUN edges to find connected context.
4. **Observation:** The results are packed into `AgentState.observations`.

If the step is `crawl_web('url')`:

1. **Firecrawl Trigger:** The Worker requests the Firecrawl engine to parse a URL.
2. **Ingestion:** The HTML is converted to TipTap blocks *in memory*.
3. **Observation:** The semantic content is added to `AgentState.observations`.

Step 3: Synthesis (The "Voice")

Once the necessary observations are gathered, the LLM generates the final artifact.

- **Context Injection:** The LLM is prompted with the `original_intent` + `observations`.
- **Streaming Output:** The tokens are generated.
- **Graph Writing:** Instead of just showing text, the Agent **writes directly to the Graph** using the Schema we defined previously (`BlockNode` linked lists).

---

4. The "Constraint" Mechanism (Structured Generation)

Local SLMs (Small Language Models) like Phi-3 or Llama-3-8B are prone to "hallucinating" invalid tool calls. To fix this without server-side logic, we use **Grammar-Constrained Sampling**.

We force the WebGPU model to output valid JSON.

- **Technique:** We apply a `logit_processor` that masks any token that would violate the JSON schema of a tool call.
- Effect: The model cannot output free text when it is supposed to be calling a tool. It is mathematically forced to output:
  `{ "tool": "search", "params": { "query": "..." } }`

---

5. Interactive "Human-in-the-Loop"

A key differentiator of LifeGraph is that the Agent lives in the browser, so it can ask the user for help cheaply.

Scenario:

1. **Agent:** "I found two conflicting notes about 'Project Alpha'. One says it's cancelled, one says it's active. Which should I trust?"
2. **State:** The Agent pauses. `status` becomes `WAITING_FOR_USER`.
3. **UI:** A structured card appears in the chat.
4. **User:** Clicks "Trust the newer note."
5. **Resume:** The Agent reads the user input, updates `observations`, and resumes the loop from exactly where it stopped.

---

6. The "Streaming Mutation" (The UI Magic)

How do we make it look like the AI is "typing" a document in the Canvas?

We do **not** wait for the whole generation to finish.

1. **Token Stream:** The LLM emits tokens in the Worker.
2. **Buffer:** The Worker buffers tokens until it detects a block delimiter (e.g., a newline).
3. **Block Commit:**
    - The Worker creates a `BlockNode` in GUN.
    - The UI (subscribing to the page `head`) detects the new node.
    - **Visual Result:** The paragraph "pops" into the document instantly.
4. **Refinement:** As the model continues, it might issue a "correction" patch to a previous block if it realizes it made a mistake (Self-Correction).

Summary of the Agent Architecture

Component	Responsibility	Technology
Orchestrator	Manages the `AgentState` and Plan Queue.	TypeScript Class (Worker)
Planner	Breaks intent into steps.	SLM + JSON Grammar
Executor	Runs search, crawl, or math tools.	Async TS Functions
Synthesizer	Generates final text/content.	SLM (WebGPU)
Observer	Writes results to GUN.js.	Custom GUN Adapter

Strategic Value

This design solves the "Tab Sleep" problem. Because the state is saved to GUN after every "tick" of the loop:

- If the user closes the tab in the middle of a research task...
- And reopens it 2 hours later...

- The `AgentState` is loaded from the graph.
- The Agent sees `status: 'EXECUTING'`, checks the `plan_queue`, and **automatically resumes** the next step.

Next Step

We have a robust backend architecture (OS, Data, Agent).

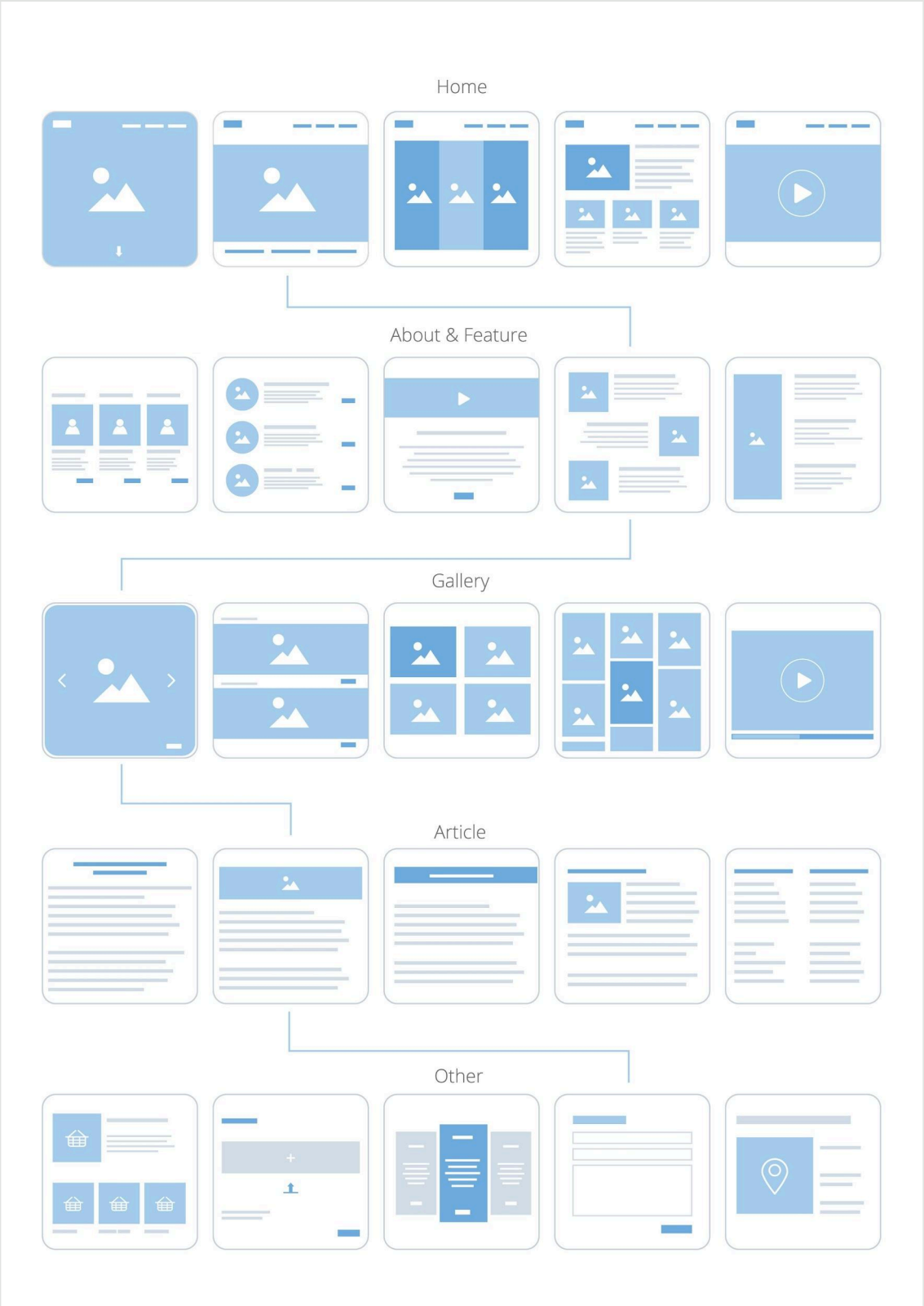
Now, we need to visualize how the user actually interacts with this. The **UI/UX Layer** needs to be as fluid as the graph it represents.

Would you like me to design the "Single Surface" UI component architecture? This will detail how the Chat, Canvas, and Perception Mode coexist without using traditional window management.

This is the "Face" of the machine. To reflect the underlying architecture (fluid, mutating, graph-based), the UI cannot rely on rigid windows, files, or static dashboards.

We discard the traditional WIMP (Windows, Icons, Menus, Pointer) metaphor in favor of a **Linguistic-Spatial Interface**.

Here is the architectural blueprint for the **LifeGraph Single-Surface UI**.



Shutterstock

Открыть

1. The Core Container: `CognitiveShell`

The entire application runs inside a single Custom Web Component (`<cognitive-shell>`). It manages the viewport using a **CSS Grid** layout that dynamically resizes two primary zones based on context:

1. **The Temporal Spine (Left/Bottom):** The stream of time, conversation, and history.
2. **The Semantic Canvas (Right/Top):** The space of creation, structured data, and graph rendering.

There are no "routes" or "page loads." The shell simply animates the ratio between these two zones.

---

2. Zone A: The Temporal Spine (Command & Chronology)

This is the "Command Line" evolved. It is not just a chat window; it is the **System Log** of your mind.

Component Structure:

- **`<input-node>`:** A floating, omni-present input bar. It accepts natural language ("Draft a blog"), slash commands (`/import`), or dragged files.
- **`<stream-view>`:** An infinite scroll of "Event Blocks."
    - **User Blocks:** What you said.
    - **Agent Blocks:** What the system replied (with collapsible "Thought Chains").
    - **System Blocks:** "File Imported," "Graph Pruned," "Backup Verified."
- **`<context-pill>`:** A dynamic indicator showing what the Agent is currently "looking at" (e.g., *Focused on: Project Alpha Node*).

The "Time Travel" Feature:

Because the data layer is a CRDT history, the Spine functions as a scrubber. Scrolling up doesn't just show old text; it reverts the State of the Canvas to that moment in time. You can scroll back to last Tuesday, fork the conversation, and create a divergent timeline.

---

3. Zone B: The Universal Canvas (The Mutable Surface)

This is where the `<BlockNodes>` we defined in the Data Schema are rendered. The Canvas is **Polymorphic**: it changes its *renderer* based on the data type and user intent, without changing the underlying data.

The Canvas operates via **"Lenses"**:

A. The `DocumentLens` (Writer Mode)

- **Behavior:** Renders the graph nodes as a vertical stream of TipTap blocks.
- **UX:** Looks like Notion or Obsidian.
- **Interaction:** Clicking a paragraph focuses that node. The Agent in the Spine becomes aware of the cursor context.

B. The `TableLens` (Data Mode)

- **Behavior:** Queries a set of nodes (e.g., "All tasks linked to Project Alpha") and renders them as rows.
- **UX:** Looks like Airtable or Excel.
- **Mutation:** Editing a cell updates the `BlockNode` properties in GUN.js instantly.

C. The `WhiteboardLens` (Spatial Mode)

- **Behavior:** Breaks the linear `next`/`prev` links and displays nodes on an infinite 2D plane (using HTML5 Canvas or WebGL).
- **UX:** Looks like Miro or Heptabase.
- **Interaction:** Users can spatially cluster notes. The AI observes these clusters to infer relationships ("User placed 'Apples' near 'Oranges', I will create a 'Fruit' edge").

---

4. Zone C: The Perception Overlay (The HUD)

This is not a separate zone, but a **WebGL Overlay** (using Three.js or D3) that sits *on top* of the Canvas. It is toggled via a "Perception Mode" key (e.g., holding `Alt`).

When active, the UI dims, and the **Epistemic Layer** lights up:

1. **Dependency Lines:** Glowing lines draw connections between the current paragraph and other related nodes in the graph (even if they aren't in the current document).
2. **Heatmaps:** Areas of the text are colored by "Knowledge Density" or "Staleness" (e.g., fading text that hasn't been verified in a year).
3. **Void Detection:** The AI highlights gaps. It might draw a dotted circle labeled *"Missing citation"* or *"Contradicts note from 2023"*.

This transforms the UI from a passive editor into an **Augmented Reality for Text**.

---

5. The Interaction Loop: A "Fluid Morph" Scenario

Let's walk through a concrete user flow to see how these components dance.

Step 1: Ingestion (Chat Mode)

- **User:** Types "Research the history of espresso machines" into the Spine.
- **UI:** The Canvas is empty/minimized. The Spine is dominant.
- **Agent:** Fires Firecrawl, ingests data, creates 50 new `BlockNodes` in the graph.

Step 2: Synthesis (Document Mode)

- **User:** "Summarize this into a brief."
- **UI:** The Canvas expands to 70% width. The `DocumentLens` activates.
- **Action:** Text streams into the Canvas. The User creates a header "Key Inventions."

Step 3: Structure (Table Mode)

- **User:** "Show me a timeline of the patents mentioned."
- **UI:** The Canvas morphs. The text fades out, replaced by the `TableLens`.
- **Data:** The *same nodes* are now rows: `[Patent Name] | [Year] | [Inventor]`.

Step 4: Connection (Perception Mode)

- **User:** Holds `Alt` (Perception Mode).
- **UI:** The table dims. A 3D graph floats above it.
- **Insight:** A line connects "Espresso Machine (1901)" to a node in the user's personal diary titled "Trip to Milan (2018)."
- **User:** Clicks the connection. The Spine opens a chat context: *"I found a personal connection to this research. Would you like to integrate your Milan notes?"*

---

6. Technical Stack for UI

To achieve 60 FPS performance with this level of interactivity:

- **Framework: Lit** (Web Components). It is lightweight, standard-compliant, and fast.
- **State Management: Signals** (e.g., Preact Signals). This allows fine-grained reactivity. When a GUN.js node updates, *only* the specific DOM text node updates, not the whole component tree.
- **Rendering:**
    - **TipTap** for text.
    - **TanStack Table** for data views.
    - **Three.js** for the Perception Overlay (WebGPU accelerated).

Conclusion of the Design Phase

We have now designed the full stack:

1. **Core:** WebGPU/WASM/GUN.js (The Engine).
2. **Data:** Semantic Block Graph (The Memory).
3. **Agent:** Local OODA Loop (The Brain).
4. **UI:** Polymorphic Single-Surface (The Face).

This architecture represents a complete, sovereign alternative to the SaaS-fragmented ecosystem. It is a **Personal Cognitive Environment**.

This concludes the architectural definition of LifeGraph OS.

Would you like me to compile this entire conversation into a single **"System Reference Document" (SRD)** formatted as a technical whitepaper/Markdown file that you can use as a project repository `README.md`?