Agent Designer - Implementation Plan

Agent Designer - Implementation Plan

Current State Analysis

Working:

- ✅ Visual node placement (Entry Point, Process, Decision, Tool Call, End)
- ✅ Node connections
- ✅ Drag and drop
- ✅ YAML preview generation
- ✅ Backspace delete

Broken:

- ❌ Edit Node panel save
- ❌ Delete button in panel
- ❌ Changes not persisting
- ❌ No versioning
- ❌ No collaboration

Implementation Phases

Phase 1: Fix Core Editor (2-3 hours)

File: `frontend/components/AgentDesigner.tsx`

```
// 1. Fix Edit Node Panel
const [nodes, setNodes] = useNodesState(initialNodes);
const [selectedNode, setSelectedNode] = useState<Node | null>(null);
const onNodeClick = (event: React.MouseEvent, node: Node) ⇒ {
  setSelectedNode(node);
};

const updateNodeData = (nodeId: string, data: Partial<NodeData>) ⇒ {
  setNodes((nds) ⇒
    nds.map((node) ⇒ {
      if (node.id === nodeId) {
        return {
          ...node,
          data: { ...node.data, ...data },
        };
      }
      return node;
    })
  );

  // Persist to backend
  autosave();
};

const deleteNode = (nodeId: string) ⇒ {
  setNodes((nds) ⇒ nds.filter((node) ⇒ node.id !== nodeId));
  setSelectedNode(null);
  autosave();
};
```

EditNodePanel.tsx:

```
export function EditNodePanel({
  node,
  onUpdate,
  onDelete
}: EditNodePanelProps) {
  const [label, setLabel] = useState(node.data.label);
  const [description, setDescription] = useState(node.data.description);

  const handleSave = () ⇒ {
    onUpdate(node.id, { label, description });
  };

  const handleDelete = () ⇒ {
    onDelete(node.id);
  };

  return (
    <div className="edit-panel">
      <input
        value={label}
        onChange={(e) ⇒ setLabel(e.target.value)}
        onBlur={handleSave}  // Auto-save on blur
      />
      <textarea
        value={description}
        onChange={(e) ⇒ setDescription(e.target.value)}
        onBlur={handleSave}
      />
      <button onClick={handleDelete} className="delete-btn">
        🗑️ Delete Node
      </button>
    </div>
  );
}
```

Phase 2: Storage Layer (3-4 hours)

Backend Structure:

```
backend/
├── storage/
```

```
│   ├── redis_store.py      # Drafts/collaboration
│   ├── sqlite_store.py     # Saved versions
│   └── version_control.py  # Git-like checkout
└── api/
    └── designer.py         # Designer endpoints
```

Redis Store (Drafts):

```
# backend/storage/redis_store.py
import redis
import json
from datetime import datetime

class DesignerDraftStore:
    def __init__(self):
        self.redis = redis.Redis(host='redis', port=6379, decode_responses=Tru
e)

    def save_draft(self, agent_id: str, user_id: str, graph_data: dict):
        """Save working draft to Redis"""
        key = f"draft:{agent_id}:{user_id}"

        draft = {
            "graph": graph_data,
            "timestamp": datetime.utcnow().isoformat(),
            "user_id": user_id
        }

        # Set with 24hr expiry
        self.redis.setex(key, 86400, json.dumps(draft))

        # Publish to collaboration channel
        self.redis.publish(f"agent:{agent_id}:updates", json.dumps({
            "type": "draft_update",
            "user_id": user_id,
            "timestamp": draft["timestamp"]
        }))

    def get_draft(self, agent_id: str, user_id: str) → dict | None:
        """Retrieve user's draft"""
        key = f"draft:{agent_id}:{user_id}"
        draft = self.redis.get(key)
        return json.loads(draft) if draft else None

    def get_active_users(self, agent_id: str) → list[str]:
        """Get list of users currently editing"""
        pattern = f"draft:{agent_id}:*"
        keys = self.redis.keys(pattern)
        return [key.split(":")[-1] for key in keys]
```

SQLite Store (Saved Versions):

```
# backend/storage/sqlite_store.py
import sqlite3
from datetime import datetime

class AgentVersionStore:
    def __init__(self, db_path="data/foundry.db"):
        self.db_path = db_path
        self._init_db()

    def _init_db(self):
        with sqlite3.connect(self.db_path) as conn:
            conn.execute("""
                CREATE TABLE IF NOT EXISTS agent_versions (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    agent_id TEXT NOT NULL,
                    version INTEGER NOT NULL,
                    graph_json TEXT NOT NULL,
                    python_code TEXT,
                    commit_message TEXT,
                    author_id TEXT NOT NULL,
                    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    is_checked_out BOOLEAN DEFAULT 0,
                    checked_out_by TEXT,
                    UNIQUE(agent_id, version)
                )
            """)

            conn.execute("""
                CREATE INDEX IF NOT EXISTS idx_agent_versions
                ON agent_versions(agent_id, version DESC)
            """)

    def commit_version(
        self,
        agent_id: str,
        graph_json: str,
        python_code: str,
        commit_message: str,
        author_id: str
    ) → int:
        """Save new version (like git commit)"""
        with sqlite3.connect(self.db_path) as conn:
            # Get next version number
            cursor = conn.execute(
                "SELECT MAX(version) FROM agent_versions WHERE agent_id = ?",
                (agent_id,)
            )
            max_version = cursor.fetchone()[0] or 0
            next_version = max_version + 1

            conn.execute("""
                INSERT INTO agent_versions
                (agent_id, version, graph_json, python_code, commit_message, auth
or_id)
                VALUES (?, ?, ?, ?, ?, ?)
            """, (agent_id, next_version, graph_json, python_code, commit_messag
e, author_id))

            return next_version

    def checkout_version(self, agent_id: str, version: int, user_id: str) → dict:
        """Checkout a version for editing (like git checkout)"""
        with sqlite3.connect(self.db_path) as conn:
            # Check if anyone else has it checked out
            cursor = conn.execute("""
                SELECT checked_out_by FROM agent_versions
                WHERE agent_id = ? AND version = ? AND is_checked_out = 1
            """, (agent_id, version))

            result = cursor.fetchone()
            if result and result[0] != user_id:
                raise ValueError(f"Version checked out by {result[0]}")

            # Check it out
            conn.execute("""
                UPDATE agent_versions
                SET is_checked_out = 1, checked_out_by = ?
                WHERE agent_id = ? AND version = ?
            """, (user_id, agent_id, version))

            # Return version data
            cursor = conn.execute("""
                SELECT graph_json, python_code, version, commit_message
                FROM agent_versions
                WHERE agent_id = ? AND version = ?
            """, (agent_id, version))

            row = cursor.fetchone()
            return {
                "graph": row[0],
                "python": row[1],
                "version": row[2],
                "message": row[3]
            }

    def get_version_history(self, agent_id: str) → list[dict]:
        """Get commit history (like git log)"""
        with sqlite3.connect(self.db_path) as conn:
            cursor = conn.execute("""
                SELECT version, commit_message, author_id, created_at, is_checke
d_out
                FROM agent_versions
                WHERE agent_id = ?
                ORDER BY version DESC
            """, (agent_id,))

            return [
                {
                    "version": row[0],
                    "message": row[1],
                    "author": row[2],
                    "timestamp": row[3],
                    "checked_out": bool(row[4])
                }
                for row in cursor.fetchall()
            ]
```

Phase 3: Real-Time Collaboration (4-5 hours)

WebSocket Server:

```
# backend/api/designer_ws.py
from fastapi import WebSocket, WebSocketDisconnect
from typing import Dict, Set
import json

class CollaborationManager:
    def __init__(self):
        # agent_id → set of WebSocket connections
        self.active_connections: Dict[str, Set[WebSocket]] = {}
        self.user_cursors: Dict[str, dict] = {}  # Track cursor positions

    async def connect(self, websocket: WebSocket, agent_id: str, user_id: str):
        await websocket.accept()

        if agent_id not in self.active_connections:
            self.active_connections[agent_id] = set()

        self.active_connections[agent_id].add(websocket)

        # Broadcast user joined
        await self.broadcast(agent_id, {
            "type": "user_joined",
            "user_id": user_id
        }, exclude=websocket)

    async def disconnect(self, websocket: WebSocket, agent_id: str, user_id: st
r):
        self.active_connections[agent_id].remove(websocket)

        # Broadcast user left
        await self.broadcast(agent_id, {
            "type": "user_left",
            "user_id": user_id
        })

    async def broadcast(self, agent_id: str, message: dict, exclude: WebSocket
= None):
        """Broadcast to all connected users"""
        if agent_id not in self.active_connections:
            return

        for connection in self.active_connections[agent_id]:
            if connection != exclude:
                await connection.send_json(message)

    async def handle_update(self, agent_id: str, user_id: str, update: dict):
        """Handle node/edge update from a user"""
        # Broadcast to others
        await self.broadcast(agent_id, {
            "type": "graph_update",
            "user_id": user_id,
            "update": update
        })

manager = CollaborationManager()

@app.websocket("/ws/designer/{agent_id}")
async def designer_websocket(websocket: WebSocket, agent_id: str, user_id:
str):
    await manager.connect(websocket, agent_id, user_id)

    try:
        while True:
            data = await websocket.receive_json()

            if data["type"] == "node_update":
                await manager.handle_update(agent_id, user_id, data)
            elif data["type"] == "cursor_move":
                await manager.broadcast(agent_id, {
                    "type": "cursor_update",
                    "user_id": user_id,
                    "x": data["x"],
                    "y": data["y"]
                })

    except WebSocketDisconnect:
        await manager.disconnect(websocket, agent_id, user_id)
```

Frontend WebSocket Hook:

```
// frontend/hooks/useCollaboration.ts
import { useEffect, useRef } from 'react';

export function useCollaboration(agentId: string, userId: string) {
  const ws = useRef<WebSocket | null>(null);

  useEffect(() => {
    ws.current = new WebSocket(
      `ws://localhost:8000/ws/designer/${agentId}?user_id=${userId}`
    );

    ws.current.onmessage = (event) => {
      const data = JSON.parse(event.data);

      switch (data.type) {
        case 'user_joined':
          console.log(`${data.user_id} joined`);
          break;

        case 'graph_update':
          // Apply remote changes
          applyRemoteUpdate(data.update);
          break;

        case 'cursor_update':
          // Show other user's cursor
          updateRemoteCursor(data.user_id, data.x, data.y);
          break;
      }
    };

    return () => ws.current?.close();
  }, [agentId, userId]);
  const broadcastUpdate = (update: any) => {
    ws.current?.send(JSON.stringify({
      type: 'node_update',
      ...update
    }));
  };

  return { broadcastUpdate };
}
```

Phase 4: Python ↔ ReactFlow Conversion (5-6 hours)

Python to ReactFlow:

```
# backend/converters/python_to_react_flow.py
import ast
from typing import Dict, List

def parse_langgraph_to_reactflow(python_code: str) -> Dict:
    """
    Parse Python LangGraph code into ReactFlow nodes/edges

    Looks for patterns like:
    - workflow.add_node("name", function)
    - workflow.add_edge("from", "to")
    - workflow.set_entry_point("name")
    """

    tree = ast.parse(python_code)

    nodes = []
    edges = []
    entry_point = None

    for node in ast.walk(tree):
        # Find workflow.add_node() calls
        if isinstance(node, ast.Call):
            if (isinstance(node.func, ast.Attribute) and
                node.func.attr == "add_node"):

                node_name = node.args[0].s  # First arg is node name
                nodes.append({
                    "id": node_name,
                    "type": "process",
                    "position": {"x": 0, "y": 0},  # Layout later
                    "data": {"label": node_name}
                })

            # Find workflow.add_edge() calls
            elif (isinstance(node.func, ast.Attribute) and
                  node.func.attr == "add_edge"):

                from_node = node.args[0].s
                to_node = node.args[1].s

                edges.append({
                    "id": f"{from_node}-{to_node}",
                    "source": from_node,
                    "target": to_node
                })

            # Find entry point
            elif (isinstance(node.func, ast.Attribute) and
                  node.func.attr == "set_entry_point"):

                entry_point = node.args[0].s

    # Auto-layout nodes
    nodes = layout_nodes(nodes, edges, entry_point)

    return {"nodes": nodes, "edges": edges}
```

```
def layout_nodes(nodes: List, edges: List, entry_point: str) → List:
    """Simple vertical layout"""
    # Build dependency graph
    node_order = topological_sort(nodes, edges, entry_point)

    for i, node_id in enumerate(node_order):
        for node in nodes:
            if node["id"] == node_id:
                node["position"] = {
                    "x": 250,
                    "y": i * 150
                }

    return nodes
```

ReactFlow to Python:

```
# backend/converters/react_flow_to_python.py
from jinja2 import Template

LANGGRAPH_TEMPLATE = Template("""
from typing import TypedDict
from langgraph.graph import StateGraph, END
from langchain_anthropic import ChatAnthropic

class AgentState(TypedDict):
    messages: list
    # Add your state fields here

{% for node in nodes %}
def {{ node.id }}(state: AgentState) → AgentState:
    \"\"\"{{ node.data.description or node.data.label }}\"\"\"
    # TODO: Implement node logic
    return state
{% endfor %}
```

```
def create_graph() → StateGraph:
    workflow = StateGraph(AgentState)

    # Add nodes
{% for node in nodes %}
    workflow.add_node("{{ node.id }}", {{ node.id }})
{% endfor %}

    # Set entry point
    workflow.set_entry_point("{{ entry_point }}")

    # Add edges
{% for edge in edges %}
    workflow.add_edge("{{ edge.source }}", "{{ edge.target }}")
{% endfor %}

    return workflow.compile()
""")

def reactflow_to_python(graph_data: dict) → str:
    """Convert ReactFlow graph to Python code"""

    nodes = graph_data["nodes"]
    edges = graph_data["edges"]

    # Find entry point
    entry_point = next(
        (n["id"] for n in nodes if n.get("type") == "entryPoint"),
        nodes[0]["id"]
    )

    return LANGGRAPH_TEMPLATE.render(
        nodes=nodes,
        edges=edges,
        entry_point=entry_point
    )
```

Phase 5: Mermaid Integration (2-3 hours)

```
# backend/converters/mermaid_to_reactflow.py
def mermaid_to_reactflow(mermaid_code: str) → dict:
    """
    Parse Mermaid flowchart into ReactFlow

    Example Mermaid:
    graph TD
        A[Entry] ⟶ B[Process]
        B ⟶ C{Decision}
        C ⟶|Yes| D[End]
        C ⟶|No| B
    """

    nodes = []
    edges = []

    for line in mermaid_code.split("\n"):
        line = line.strip()

        if "⟶" in line:
            # Parse edge: A ⟶ B
            parts = line.split("⟶")
            source = parts[0].strip()
            target = parts[1].strip().split("[")[0].strip()

            edges.append({
                "id": f"{source}-{target}",
                "source": source,
                "target": target
            })
        elif "[" in line and "]" in line:
            # Parse node: A[Label]
            node_id = line.split("[")[0].strip()
            label = line.split("[")[1].split("]")[0]

            nodes.append({
                "id": node_id,
                "type": "process",
                "data": {"label": label},
                "position": {"x": 0, "y": 0}
            })

    return {"nodes": layout_nodes(nodes, edges), "edges": edges}
```

API Endpoints

```
# backend/api/designer.py
from fastapi import APIRouter, HTTPException
from storage.redis_store import DesignerDraftStore
from storage.sqlite_store import AgentVersionStore
from converters.python_to_react_flow import parse_langgraph_to_reactflow
from converters.react_flow_to_python import reactflow_to_python

router = APIRouter(prefix="/api/designer")

draft_store = DesignerDraftStore()
version_store = AgentVersionStore()

@router.post("/agents/{agent_id}/draft")
async def save_draft(agent_id: str, user_id: str, graph_data: dict):
    """Auto-save draft to Redis"""
    draft_store.save_draft(agent_id, user_id, graph_data)
    return {"status": "saved"}
```

```
@router.get("/agents/{agent_id}/draft")
async def get_draft(agent_id: str, user_id: str):
    """Load user's draft"""
    draft = draft_store.get_draft(agent_id, user_id)
    if not draft:
        raise HTTPException(404, "No draft found")
    return draft

@router.post("/agents/{agent_id}/commit")
async def commit_version(
    agent_id: str,
    user_id: str,
    graph_data: dict,
    message: str
):
    """Save version to SQLite (like git commit)"""
    python_code = reactflow_to_python(graph_data)

    version = version_store.commit_version(
        agent_id=agent_id,
        graph_json=json.dumps(graph_data),
        python_code=python_code,
        commit_message=message,
        author_id=user_id
    )

    return {"version": version, "python": python_code}

@router.post("/agents/{agent_id}/checkout")
async def checkout_version(agent_id: str, version: int, user_id: str):
    """Checkout a version (like git checkout)"""
    try:
        data = version_store.checkout_version(agent_id, version, user_id)
        return data
    except ValueError as e:
        raise HTTPException(409, str(e))

@router.get("/agents/{agent_id}/history")
async def get_history(agent_id: str):
    """Get version history (like git log)"""
    return version_store.get_version_history(agent_id)

@router.post("/import/python")
async def import_python(python_code: str):
    """Convert Python LangGraph to ReactFlow"""
    graph = parse_langgraph_to_reactflow(python_code)
    return graph
```

Priority Order

1. **Fix Edit Panel** (today, 2hrs) - blocking users
2. **Autosave to Redis** (tomorrow, 3hrs) - prevent data loss
3. **Python → ReactFlow** (next, 4hrs) - lets users paste code
4. **Commit/Checkout** (next, 3hrs) - version control
5. **Real-time Collab** (later, 5hrs) - nice to have

Want me to start with fixing the Edit Panel and autosave?