```
'use client';

import { useState, useCallback, useEffect, useMemo, useRef, useContext,createContext } from 'react';
import ReactFlow, {
  Node,
  Edge,
  Controls,
  Background,
  useNodesState,
  useEdgesState,
  addEdge,
  Connection,
  NodeProps,
  MarkerType,
  EdgeTypes,
  EdgeProps,
  getBezierPath,
  Handle,
  Position,
  useReactFlow,
  ReactFlowProvider,
} from 'reactflow';
import 'reactflow/dist/style.css';
import { Button } from '../spread_sheet_swarm/ui/button';
import { Textarea } from '../ui/textarea';
import {
  Dialog,
  DialogContent,
  DialogDescription,
  DialogFooter,
  DialogHeader,
  DialogTitle,
  DialogTrigger,
} from '../ui/dialog';
import {
  Tooltip,
  TooltipContent,
  TooltipProvider,
  TooltipTrigger,
} from '../ui/tooltip';
import { Label } from '../ui/label';
import {
  Select,
  SelectContent,
  SelectItem,
  SelectTrigger,
  SelectValue,
} from '../ui/select';
import { Tabs, TabsContent, TabsList, TabsTrigger } from '../ui/tabs';
import {
  Table,
  TableBody,
  TableCell,
  TableHead,
  TableHeader,
  TableRow,
} from '../ui/table';
import {
  DropdownMenu,
  DropdownMenuContent,
  DropdownMenuItem,
  DropdownMenuLabel,
  DropdownMenuSeparator,
  DropdownMenuTrigger,
} from '../spread_sheet_swarm/ui/dropdown-menu';
import {
  Plus,
  Send,
  Save,
  Share,
  Upload,
  MoreHorizontal,
  X,
  Settings,
  Sparkles,
  Loader2,
} from 'lucide-react';
```

```
import { motion, AnimatePresence } from 'framer-motion';
import { anthropic } from '@ai-sdk/anthropic';
import { createOpenAI } from '@ai-sdk/openai';
import {
  experimental_createProviderRegistry as createProviderRegistry,
  generateText,
} from 'ai';
import { Card } from '../spread_sheet_swarm/ui/card';
import { Input } from '../spread_sheet_swarm/ui/input';
import { trpc as api } from '@/shared/utils/trpc/trpc';
import debounce from 'lodash/debounce';
import { useRouter, useSearchParams } from 'next/navigation';
import { useToast } from '@/shared/components/ui/Toasts/use-toast';
import { useEnhancedAutosave } from './autosave';
import AutoGenerateSwarm from './auto_generate_swarm';

// Create provider registry
const registry = createProviderRegistry({
  anthropic,
  openai: createOpenAI({
    apiKey: process.env.NEXT_PUBLIC_OPENAI_API_KEY,
  }),
});

type AgentType = 'Worker' | 'Boss';
type AgentModel = 'gpt-3.5-turbo' | 'gpt-4o' | 'claude-2' | 'gpt-4o-mini';
type SwarmArchitecture = 'Concurrent' | 'Sequential' | 'Hierarchical';

type ReactFlowNode = Node<AgentData>;


// Define types that exactly match your Zod schema

type NodeData = {

  id: string;

  name: string;

  type: string;

  model: string;

  systemPrompt: string;

  clusterId?: string;

  isProcessing?: boolean;

  lastResult?: string;

  dataSource?: string;

  dataSourceInput?: string;

  [key: string]: unknown; // Add index signature to match passthrough behavior

};


type SaveFlowNode = {

  id: string;

  type: string;

  position: {

    x: number;

    y: number;

  };

  data: NodeData;
  [key: string]: unknown; // Add index signature to match passthrough behavior
};


interface AgentData {
  description: string;
  id: string;
  name: string;
  type: AgentType;
  model: AgentModel;
  systemPrompt: string;
  clusterId?: any; // Made optional but explicit in the type
  isProcessing?: boolean;
  lastResult?: string;
  hideDeleteButton?: boolean;
}

interface SwarmVersion {
  id: string;
  timestamp: number;
  nodes: Node<AgentData>[];
  edges: Edge[];
  architecture: SwarmArchitecture;
  results: { [key: string]: string };
}
```

```
declare global {
  interface Window {
    updateNodeData: (id: string, updatedData: AgentData) => void;
    updateGroupData?: (id: string, updatedData: GroupData) => void;
    addAgentToGroup?: (groupId: string) => void;
    addAgent?: (agent: AgentData) => void;
    createGroup?: (groupData: Omit<GroupData, 'agents'>) => void;
  }
}
interface EdgeParams {
  source: string | null;
  target: string | null;
  sourceHandle: string | null;
  targetHandle: string | null;
  animated?: boolean;
  label?: string;
}

const AnimatedHexagon = motion.polygon;

// Add this new component for the delete button
const DeleteButton: React.FC<{ onClick: () => void }> = ({ onClick }) => (
  <Button
    variant="ghost"
    size="icon"
    onClick={(e) => {
```

```
   e.stopPropagation();
   onClick();
  }}
  className="absolute -top-3 -right-3 h-8 w-8 rounded-full bg-destructive text-destructive-foreground hover:bg-destructive/90 shadow-md border-2 border-background z-10 transition-all duration-200 hover:scale-110"
 >
  <X className="h-5 w-5" />
 </Button>
);


const AgentLoadingOverlay = () => {
 return (
  <motion.div
   initial={{ opacity: 0 }}
   animate={{ opacity: 1 }}
   exit={{ opacity: 0 }}
   className="fixed inset-0 bg-background/80 backdrop-blur-sm z-50 flex items-center justify-center"
  >
   <div className="flex flex-col items-center space-y-4">
    <motion.div
     animate={{ rotate: 360 }}
     transition={{ duration: 2, repeat: Infinity, ease: "linear" }}
     className="relative"
```

```
        >
          <div className="w-16 h-16">
            <Loader2 className="w-16 h-16 text-primary animate-spin" />
          </div>
        </motion.div>
        <motion.div
          initial={{ opacity: 0, y: 10 }}
          animate={{ opacity: 1, y: 0 }}
          className="text-lg font-medium text-foreground"
        >
          Processing Agent Tasks...
        </motion.div>
      </div>
    </motion.div>
  );
};

export default AgentLoadingOverlay;

const generateSystemPrompt = async (agentName: string, agentDescription: string) => {
  try {
    const { text } = await generateText({
      model: registry.languageModel('openai:gpt-4o'),
      prompt: `You are an expert AI prompt engineer specializing in creating advanced system prompts for AI agents in a swarm architecture. Your task is to create a highly effective system prompt for an AI agent with the following details:
```

Agent Name: ${agentName}

Agent Description: ${agentDescription || 'No description provided'}

To create a reliable and effective system prompt, follow these instructions:

1. Begin by defining the agent's role and responsibilities, ensuring they are clear and concise.

2. Establish explicit boundaries and constraints for the agent's operation, including any limitations or restrictions.

3. Provide context awareness and collaboration guidelines, outlining how the agent should interact with other agents and systems.

4. Specify the required output format, including any necessary data structures or formatting requirements.

5. Incorporate error handling and edge cases, ensuring the agent can recover from unexpected events or inputs.

6. Enable dynamic adaptation to different tasks, allowing the agent to adjust its behavior in response to changing requirements.

7. Include memory and context management, ensuring the agent can retain and utilize relevant information.

8. Define interaction patterns with other agents, outlining how they should communicate and coordinate.

9. Establish quality control measures, ensuring the agent's output meets the required standards.

10. Implement task prioritization logic, allowing the agent to manage multiple tasks and prioritize them effectively.

To showcase the agent's capabilities, provide the following multi-shot examples:

Example 1:

Input: [Provide a sample input for the agent]

Expected Output: [Describe the expected output from the agent]

Explanation: [Explain the reasoning behind the expected output]

Example 2:

Input: [Provide a sample input for the agent]

Expected Output: [Describe the expected output from the agent]

Explanation: [Explain the reasoning behind the expected output]

Example 3:

Input: [Provide a sample input for the agent]

Expected Output: [Describe the expected output from the agent]

Explanation: [Explain the reasoning behind the expected output]

Use these advanced prompt engineering techniques:

- Chain-of-thought reasoning
- Few-shot examples
- Role-based conditioning
- Task decomposition
- Output structuring
- Context window management
- Error recovery protocols
- Multi-shot examples for reliability and showcasing the agent

```
      Return only the optimized system prompt without any additional text or explanations.`,
    });

    return text;
  } catch (error) {
    console.error('Failed to generate system prompt:', error);
    throw new Error('Failed to generate system prompt');
  }
};

const cleanupBodyStyles = () => {
  document.body.style.removeProperty('pointer-events');
  // Also remove any other unwanted styles that might be added
  document.body.style.removeProperty('overflow');
};

// Create a hook to manage body styles
const useBodyStyleCleanup = (isOpen: boolean) => {
  useEffect(() => {
    // Clean up styles when modal closes
    if (!isOpen) {
      cleanupBodyStyles();
    }

    // Always clean up on unmount
    return () => {
      cleanupBodyStyles();
    };
  }, [isOpen]);
};


// Add this utility function near the top with other utility functions
const optimizePrompt = async (currentPrompt: string): Promise<string> => {
  if (!currentPrompt?.trim()) {
    throw new Error('System prompt is required for optimization');
  }

  try {
    const { text } = await generateText({
      model: registry.languageModel('openai:gpt-4o'),
      prompt: `
        Your task is to optimize the following system prompt for an AI agent. The optimized prompt should be highly reliable, production-grade, and tailored to the specific needs of the agent. Consider the following guidelines:

      1. Thoroughly understand the agent's requirements and capabilities.
      2. Employ diverse prompting strategies (e.g., chain of thought, few-shot learning).
      3. Blend strategies effectively for the specific task or scenario.
      4. Ensure production-grade quality and educational value.
      5. Provide necessary constraints for the agent's operation.
      6. Design for extensibility and wide scenario coverage.
      7. Aim for a prompt that fosters the agent's growth and specialization.

      Original prompt to optimize:
      ${currentPrompt}

      Please provide an optimized version of this prompt, incorporating the guidelines mentioned above. Only return the optimized prompt, no other text or comments.
      `,
    });

    return text;
  } catch (error) {
    console.error('Failed to optimize prompt:', error);
    throw new Error('Failed to optimize system prompt');
  }
};

const AgentNode: React.FC<NodeProps<AgentData> & { hideDeleteButton?: boolean }> = ({ data, id, hideDeleteButton }) => {
  const [isEditing, setIsEditing] = useState(false);
  const [isGenerating, setIsGenerating] = useState(false);
  const [isOptimizing, setIsOptimizing] = useState(false);
  const [localSystemPrompt, setLocalSystemPrompt] = useState(data.systemPrompt || '');
  const { setNodes, setEdges } = useReactFlow();
  const { toast } = useToast(); // Add toast import if not already present
```

```
useEffect(() => {
  const cleanup = () => {
    // Check if no modals are open before cleaning up
    const anyModalOpen = document.querySelector('[role="dialog"]');
    if (!anyModalOpen) {
      cleanupBodyStyles();
    }
  };

  // Listen for escape key
  const handleEscape = (e: KeyboardEvent) => {
    if (e.key === 'Escape') {
      cleanup();
    }
  };

  window.addEventListener('keydown', handleEscape);

  // Add a mutation observer to detect DOM changes
  const observer = new MutationObserver(() => {
    // Check if any dialog was removed
    const anyModalOpen = document.querySelector('[role="dialog"]');
    if (!anyModalOpen) {
      cleanup();
    }
  });

  observer.observe(document.body, {
    childList: true,
    subtree: true,
  });

  return () => {
    window.removeEventListener('keydown', handleEscape);
    observer.disconnect();
    cleanup();
  };
}, []);

useBodyStyleCleanup(isEditing);

// Update localSystemPrompt when data changes
useEffect(() => {
  setLocalSystemPrompt(data.systemPrompt || '');
}, [data.systemPrompt]);

const handleDelete = useCallback(() => {
  setNodes(nodes => {
    // Create a new array of nodes
    return nodes.map(node => {
      // If this is a group node, check if it contains the agent
      if (node.type === 'group' && node.data.agents) {
        return {
          ...node,
          data: {
            ...node.data,
            // Remove the agent from the group's agents array
            agents: node.data.agents.filter((agent: AgentData) => agent.id !== data.id)
          }
        };
      }
      // Remove the agent node itself
      return node.id !== id ? node : null;
    }).filter(Boolean) as Node[];
  });

  // Remove any edges connected to this agent
  setEdges(edges => edges.filter(edge =>
    edge.source !== id && edge.target !== id
  ));
}, [id, data.id, setNodes, setEdges]);

// Add the generate prompt handler


// Add the optimize prompt handler
const handleOptimizePrompt = async () => {
```

```
if (!localSystemPrompt) {
  toast({
    title: "Error",
    description: "System prompt is required for optimization",
    variant: "destructive"
  });
  return;
}

setIsOptimizing(true);
try {
  const optimizedPrompt = await optimizePrompt(localSystemPrompt);

  // Update both local state and node data
  setLocalSystemPrompt(optimizedPrompt);

  // Update the node data in the flow
  setNodes((nodes) =>
    nodes.map((node) => {
      if (node.id === id) {
        return {
          ...node,
          data: {
            ...node.data,
            systemPrompt: optimizedPrompt,
          },
          };
        }
        return node;
      })
    );

    toast({
      description: "System prompt optimized successfully",
    });
  } catch (error) {
    toast({
      title: "Error",
      description: "Failed to optimize system prompt",
      variant: "destructive"
    });
  }
  setIsOptimizing(false);
};

return (
  <div className="relative">
    {!hideDeleteButton &&
    <DeleteButton onClick={handleDelete} />}
    <TooltipProvider>
      <Tooltip>
        <TooltipTrigger>
```

```
<div className="relative" onClick={() => setIsEditing(true)}>
  <svg width="80" height="80" viewBox="0 0 100 100">
    <AnimatedHexagon
      points="50 1 95 25 95 75 50 99 5 75 5 25"
      fill={data.type === "Boss"
        ? "hsl(var(--card))"
        : "hsl(var(--secondary))"
      }
      stroke={document.documentElement.classList.contains('dark') ? "#333" : "hsl(var(--border))"}
      strokeWidth="2"
      initial={{ scale: 0 }}
      animate={{
        scale: 1,
        rotate: data.isProcessing ? 360 : 0
      }}
      transition={{
        duration: 0.5,
        repeat: data.isProcessing ? Infinity : 0
      }}
      style={{
        fill: document.documentElement.classList.contains('dark')
          ? data.type === "Boss"
            ? "#0F0F10"
            : "#1A1A1B"
          : data.type === "Boss"
```

```
                      ? "hsl(var(--card))"
                      : "hsl(var(--secondary))"
                }}
              />
            </svg>
            <div className="absolute p-2 inset-0 flex flex-col items-center justify-center text-xs">
              <div className="font-bold text-card-foreground text-[0.6rem]">
                {data.name}
              </div>
              <div className="text-muted-foreground">{data.type}</div>
            </div>
            {data.lastResult && (
              <motion.div
                className="absolute -top-2 -right-2 bg-primary text-primary-foreground text-xs rounded-full w-6 h-6 flex items-center justify-center"
                initial={{ scale: 0 }}
                animate={{ scale: 1 }}
                transition={{ type: 'spring', stiffness: 500, damping: 30 }}
              >

              </motion.div>
            )}
          </div>
        </TooltipTrigger>
        <TooltipContent className="max-w-[500px] bg-popover text-popover-foreground border-border">
```

```
     <div className="text-sm">
      <p>
       <strong>Name:</strong> {data.name}
      </p>
      <p>
       <strong>Type:</strong> {data.type}
      </p>
      <p>
       <strong>Model:</strong> {data.model}
      </p>
      <p>
       <strong>System Prompt:</strong> {data.systemPrompt}
      </p>
      {data.lastResult && (
       <p>
        <strong>Last Result:</strong> {data.lastResult}
       </p>
      )}
     </div>
    </TooltipContent>
   </Tooltip>
  </TooltipProvider>
  <AnimatePresence>
  {isEditing && (
<motion.div
 initial={{ opacity: 0, scale: 0.9 }}
```

```
  animate={{ opacity: 1, scale: 1 }}
  exit={{ opacity: 0, scale: 0.9 }}
  className="fixed inset-0 z-50 flex items-center justify-center bg-background/95"
 >
   <Card className="w-96 min-w-[500px] bg-card text-card-foreground p-6 rounded-lg shadow-xl border-border relative z-[60]">
   <div className="flex justify-between items-center mb-4">
        <h3 className="text-lg font-semibold text-card-foreground antialiased">
         Edit Agent
        </h3>
        <Button
         variant="ghost"
         size="icon"
         onClick={() => setIsEditing(false)}
         className="text-muted-foreground hover:text-card-foreground"
        >
         <X className="h-4 w-4" />
        </Button>
       </div>
       <form
        onSubmit={(e) => {
         e.preventDefault();
         const formData: any = new FormData(
          e.target as HTMLFormElement,
         );
         const updatedAgent: AgentData = {
```

```
      ...data,
      name: formData.get('name') as string,
      type: formData.get('type') as AgentType,
      model: formData.get('model') as AgentModel,
      systemPrompt: formData.get('systemPrompt') as string,
    };
    window.updateNodeData(id, updatedAgent);
    setIsEditing(false);
  }}
>
  <div className="space-y-4">
    <div>
      <Label htmlFor="name" className="text-card-foreground">
        Name
      </Label>
      <Input
        id="name"
        name="name"
        defaultValue={data.name}
        className="bg-card border-border text-card-foreground"
      />
    </div>
    <div>
      <Label htmlFor="type" className="text-card-foreground">
        Type
      </Label>
  <Select name="type" defaultValue={data.type}>
    <SelectTrigger className="bg-card border-border text-card-foreground">
      <SelectValue />
    </SelectTrigger>
  </Select>
</div>
<div>
  <Label htmlFor="model" className="text-card-foreground">
    Model
  </Label>
  <Select name="model" defaultValue={data.model}>
    <SelectTrigger className="bg-card border-border text-card-foreground">
      <SelectValue />
    </SelectTrigger>
    <SelectContent className="bg-popover border-border">
      <SelectItem value="gpt-3.5-turbo">
        GPT-3.5 Turbo
      </SelectItem>
      <SelectItem value="gpt-4o">GPT-4o</SelectItem>
      <SelectItem value="gpt-4o-mini">GPT-4o-Mini</SelectItem>
      <SelectItem value="claude-2">Claude 2</SelectItem>
    </SelectContent>
  </Select>
</div>
<div>
  <Label
    htmlFor="systemPrompt"
    className="text-card-foreground"
  >
    System Prompt
  </Label>
  <div className="relative mt-1.5">
    <Textarea
      id="systemPrompt"
      name="systemPrompt"
      value={localSystemPrompt}
      onChange={(e) => {
        const newValue = e.target.value;
        setLocalSystemPrompt(newValue);
        // Update node data
        setNodes((nodes) =>
          nodes.map((node) => {
            if (node.id === id) {
              return {
                ...node,
                data: {
                  ...node.data,
                  systemPrompt: newValue,
                },
              };
            }
            return node;
```

```
            })
          );
        }}
        className="pr-12 min-h-[100px] bg-card border-border text-card-foreground"
      />
      <Button
        type="button"
        size="sm"
        variant="ghost"
        onClick={handleOptimizePrompt}
        disabled={isOptimizing || !localSystemPrompt}
        title={!localSystemPrompt ? "System prompt required" : "Optimize prompt"}
        className="absolute right-2 top-2 h-8 w-8 p-0"
      >
        {isOptimizing ? (
          <Loader2 className="h-4 w-4 animate-spin" />
        ) : (
          <Sparkles className="h-4 w-4" />
        )}
      </Button>
    </div>
  </div>
</div>
<DialogFooter className="mt-6">
  <Button
    type="submit"
```

```
                    className="bg-primary text-primary-foreground hover:bg-primary/90"
                  >
                    Save Changes
                  </Button>
                </DialogFooter>
              </form>
            </Card>
          </motion.div>
        )}
      </AnimatePresence>
    </div>
  );
};



// Make groupProcessingStates available globally for the GroupNode component
const GroupNodeContext = createContext<{
  groupProcessingStates: any;
}>({ groupProcessingStates: {} });

const GroupNode: React.FC<NodeProps<GroupData>> = ({ data, id }) => {
  const { groupProcessingStates } = useContext(GroupNodeContext);
  const processingState = groupProcessingStates[id];
  const [isEditing, setIsEditing] = useState(false);
  const [isOptimizing, setIsOptimizing] = useState(false);
  const { setNodes, setEdges } = useReactFlow();
```

```
useBodyStyleCleanup(isEditing);


const handleDelete = useCallback(() => {
  setNodes(nodes => {
    // Get all agent IDs in this group
    const agentIds = data.agents?.map(agent => agent.id) || [];


    // Update nodes: Remove the group and update agents to be standalone
    return nodes.map(node => {
      if (node.id === id) {
        // Remove the group
        return null;
      }
      if (agentIds.includes(node.id)) {
        // Update agent to remove group association
        return {
          ...node,
          data: {
            ...node.data,
            groupId: undefined
          }
        };
      }
      return node;
    }).filter(Boolean) as Node[];
  });


  setEdges(edges => edges.filter(edge =>
    edge.source !== id && edge.target !== id
  ));
}, [id, data.agents, setNodes, setEdges]);


// Add a new function to remove agent from group
const removeAgentFromGroup = (groupId: string, agentId: string) => {
  setNodes(nodes => nodes.map(node => {
    if (node.id === groupId) {
      // Remove agent from group's agents array
      return {
        ...node,
        data: {
          ...node.data,
          agents: node.data.agents.filter((agent: AgentData) => agent.id !== agentId)
        }
      };
    }
    if (node.id === agentId) {
      // Remove group association from agent
      return {
        ...node,
        data: {
          ...node.data,
            groupId: undefined
          }
        };
      }
      return node;
    }));
  };

  return (
    <div className="relative">
      <DeleteButton onClick={handleDelete} />
      <div className="min-w-[300px] min-h-[200px] relative">
        {/* Semi-transparent backdrop with more visible styling */}
        <div className="absolute inset-0 bg-card/80 dark:bg-background/80 backdrop-blur-sm border-2 border-border dark:border-gray-800 rounded-lg shadow-lg" />

        {/* Input handle */}
        <Handle
          type="target"
          position={Position.Left}
          className="w-2 h-2 !bg-muted-foreground"
        />

        {/* Group Content */}
        <div className="relative p-4">
          {/* Group Header */}
```

```
<div className="flex justify-between items-center mb-4 border-b border-border pb-2">
  <div>
    <h3 className="font-semibold text-lg text-card-foreground">{data.teamName}</h3>
    <p className="text-sm text-muted-foreground">{data.swarmType}</p>
  </div>
  <DropdownMenu>
    <DropdownMenuTrigger asChild>
      <Button variant="ghost" size="sm" className="h-8 w-8 p-0">
        <MoreHorizontal className="h-4 w-4" />
      </Button>
    </DropdownMenuTrigger>
    <DropdownMenuContent>
      <DropdownMenuItem onClick={() => setIsEditing(true)}>
        <Settings className="w-4 h-4 mr-2" />
        Edit Group
      </DropdownMenuItem>
      <DropdownMenuItem onClick={() => window.addAgentToGroup?.(id)}>
        <Plus className="w-4 h-4 mr-2" />
        Add Agent
      </DropdownMenuItem>
    </DropdownMenuContent>
  </DropdownMenu>
</div>

{/* Agents Container */}
<div className="flex flex-wrap gap-4">
```

```
            {data.agents?.map((agent) => (
              <motion.div
                key={agent.id}
                initial={{ scale: 0.9, opacity: 0 }}
                animate={{ scale: 1, opacity: 1 }}
                className="relative"
              >
                <Button
                  variant="ghost"
                  size="icon"
                  onClick={(e) => {
                    e.stopPropagation();
                    removeAgentFromGroup(id, agent.id);
                  }}
                  className="absolute -top-2 -right-2 h-6 w-6 rounded-full bg-destructive text-destructive-foreground hover:bg-destructive/90 shadow-md border-2 border-background z-10"
                >
                  <X className="h-3 w-3" />
                </Button>
                <div className="transform scale-75 origin-top-left">
                  <AgentNode
                    data={agent}
                    id={agent.id}
                    type="agent"
                    xPos={0}
                    yPos={0}
            selected={false}
            zIndex={0}
            isConnectable={true}
            dragging={false}
            hideDeleteButton={true}
          />
        </div>
      </motion.div>
    ))}
  </div>
</div>

{/* Output handle */}
<Handle
  type="source"
  position={Position.Right}
  className="w-2 h-2 !bg-muted-foreground"
/>

{/* Edit Group Dialog */}
<Dialog open={isEditing} onOpenChange={setIsEditing}>
  <DialogContent className="sm:max-w-[425px]">
    <DialogHeader>
      <DialogTitle className="text-left">Edit Group</DialogTitle>
      <DialogDescription className="text-left">
        Make changes to the group configuration here.
  </DialogDescription>
</DialogHeader>
<form onSubmit={(e) => {
  e.preventDefault();
  const formData = new FormData(e.target as HTMLFormElement);
  const updatedData = {
    ...data,
    teamName: formData.get('teamName') as string,
    swarmType: formData.get('swarmType') as string,
    description: formData.get('description') as string,
  };
  window.updateGroupData?.(id, updatedData);
  setIsEditing(false);
}}>
  <div className="grid gap-4 py-4">
    <div className="grid grid-cols-4 items-center gap-4">
      <Label htmlFor="teamName" className="text-left">Team Name</Label>
      <Input
        id="teamName"
        name="teamName"
        defaultValue={data.teamName}
        className="col-span-3"
      />
    </div>
    <div className="grid grid-cols-4 items-center gap-4">
      <Label htmlFor="swarmType" className="text-left">Swarm Type</Label>
    <Select name="swarmType" defaultValue={data.swarmType}>
      <SelectTrigger className="col-span-3 text-left">
        <SelectValue />
      </SelectTrigger>
      <SelectContent align="start">
        <SelectItem value="Marketing">Marketing</SelectItem>
        <SelectItem value="Research">Research</SelectItem>
        <SelectItem value="Development">Development</SelectItem>
        <SelectItem value="Analytics">Analytics</SelectItem>
        <SelectItem value="Custom">Custom</SelectItem>
      </SelectContent>
    </Select>
  </div>
  <div className="grid grid-cols-4 items-center gap-4">
    <Label htmlFor="description" className="text-left">Description</Label>
    <Textarea
      id="description"
      name="description"
      defaultValue={data.description}
      className="col-span-3"
    />
  </div>
</div>
<DialogFooter>
  <Button type="submit">Save Changes</Button>
</DialogFooter>
```

```
      </form>
     </DialogContent>
    </Dialog>

    {/* Add processing indicator */}
    {processingState?.isProcessing && (
          <div className="absolute inset-0 bg-primary/20 backdrop-blur-sm flex items-center justify-center">
      <div className="text-primary-foreground">Processing...</div>
     </div>
    )}

    {/* Add completion indicator */}
    {processingState?.result && (
     <div className="absolute top-2 right-2">
       <div className="bg-primary text-primary-foreground rounded-full w-6 h-6 flex items-center justify-center">

      </div>
     </div>
    )}

   </div>
  </div>
 );
};
```

```
// eslint-disable-next-line react-hooks/rules-of-hooks
const nodeTypes ={
  agent: AgentNode,
  group: GroupNode,
};

const CustomEdge = ({
  id,
  sourceX,
  sourceY,
  targetX,
  targetY,
  sourcePosition,
  targetPosition,
  style = {},
  data,
  markerEnd,
}: EdgeProps) => {
  const { setEdges } = useReactFlow();

  // Calculate midpoint for the delete button
  const midX = (sourceX + targetX) / 2;
  const midY = (sourceY + targetY) / 2;
  const [edgePath] = getBezierPath({
    sourceX,
    sourceY,
    sourcePosition,
    targetX,
    targetY,
    targetPosition,
  });

  const handleDelete = (event: React.MouseEvent) => {
    event.stopPropagation();
    setEdges((edges) => edges.filter((e) => e.id !== id));
  };

  return (
    <>
      <path
        id={id}
        style={style}
        className="react-flow__edge-path"
        d={edgePath}
        markerEnd={markerEnd}
      />
      {/* Delete button */}
      <foreignObject
```

```
        width={24}
        height={24}
        x={midX - 12}
        y={midY - 12}
        requiredExtensions="http://www.w3.org/1999/xhtml"
      >
        <div className="flex items-center justify-center h-full">
          <button
            className="h-6 w-6 rounded-full bg-destructive/90 hover:bg-destructive flex items-center justify-center text-destructive-foreground"
            onClick={handleDelete}
          >
            <X className="h-4 w-4" />
          </button>
        </div>
      </foreignObject>
    </>
  );
};

// Make sure to update the edgeTypes
const edgeTypes: EdgeTypes = {
  custom: CustomEdge,
};

type TaskResults = { [key: string]: string };
```

```
// Add this utility function at the top level
const isEqual = (prev: any, next: any) =>
  JSON.stringify(prev) === JSON.stringify(next);

// Add new interfaces
interface GroupData {
  teamName: string;
  swarmType: string;
  agents: AgentData[];
  description?: string;
}

// Add this type definition near the top with other interfaces
interface AddAgentToGroupDialogProps {
  groupId: string;
  open: boolean;
  onOpenChange: (open: boolean) => void;
  nodes: Node[];
  setNodes: (nodes: any) => void;
}

// Add this new component before the main component
const AddAgentToGroupDialog: React.FC<AddAgentToGroupDialogProps> = ({
  groupId,
  open,
  onOpenChange,
  nodes,
  setNodes,
}) => {
  const [selectedAgents, setSelectedAgents] = useState<string[]>([]);
  useBodyStyleCleanup(open);
  // Filter out agents that are already in groups
  const availableAgents = nodes.filter((node) =>
    node.type === 'agent' &&
    !node.data.groupId &&
    node.id !== groupId
  );

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();

    setNodes((currentNodes: Node[]) =>
      currentNodes.map((node) => {
        if (node.id === groupId) {
          return {
            ...node,
            data: {
              ...node.data,
              agents: [
                ...(node.data.agents || []),
                ...selectedAgents.map(agentId =>
        nodes.find(n => n.id === agentId)?.data
       ).filter(Boolean)
      ]
     }
    };
   }
   if (selectedAgents.includes(node.id)) {
    return {
     ...node,
     data: {
      ...node.data,
      groupId
     }
    };
   }
   return node;
  })
 );

 setSelectedAgents([]);
 onOpenChange(false);
};

return (
 <Dialog open={open} onOpenChange={onOpenChange}>
    <DialogContent className="fixed top-1/2 left-1/2 transform -translate-x-1/2 -translate-y-1/2
```

```
w-[425px] bg-background border border-border rounded-lg shadow-lg z-[100]">
        <DialogHeader>
          <DialogTitle className="text-left">Add Agents to Group</DialogTitle>
          <DialogDescription className="text-left">
            Select agents to add to this group.
          </DialogDescription>
        </DialogHeader>
        {availableAgents.length === 0 ? (
          <div className="py-6 text-center text-muted-foreground">
            No available agents to add. Create new agents first.
          </div>
        ) : (
          <form onSubmit={handleSubmit} className="space-y-4">
            <div className="max-h-[300px] overflow-y-auto px-4">
              {availableAgents.map((agent) => (
                <div key={agent.id} className="flex items-center gap-2 py-2">
                  <input
                    type="checkbox"
                    id={agent.id}
                    value={agent.id}
                    checked={selectedAgents.includes(agent.id)}
                    onChange={(e) => {
                      if (e.target.checked) {
                        setSelectedAgents([...selectedAgents, agent.id]);
                      } else {
                        setSelectedAgents(selectedAgents.filter(id => id !== agent.id));
```

```
           }
          }}
          className="h-4 w-4 rounded border-border"
         />
         <Label htmlFor={agent.id} className="flex-grow text-left">
          {agent.data.name} ({agent.data.type})
         </Label>
        </div>
       ))}
      </div>
      <DialogFooter className="px-4 pb-4">
       <Button
        type="submit"
        disabled={selectedAgents.length === 0}
        className="w-full"
       >
        Add Selected Agents
       </Button>
      </DialogFooter>
     </form>
    )}
   </DialogContent>
  </Dialog>
 );
};
```

```
// Create a wrapped component for the main content
const FlowContent = () => {
  const router = useRouter();
  const searchParams = useSearchParams();

  // Make sure your useNodesState is properly typed
  const [nodes, setNodes, onNodesChange] = useNodesState([]);
  const [isLoading, setIsLoading] = useState(true);
  const [isCreatingGroup, setIsCreatingGroup] = useState(false);
  const [addToGroupDialogState, setAddToGroupDialogState] = useState<{
    open: boolean;
    groupId: string | null;
  }>({
    open: false,
    groupId: null
  });
  // const [nodes, setNodes, onNodesChange] = useNodesState<AgentData>([]);
  const [edges, setEdges, onEdgesChange] = useEdgesState([]);
  const [task, setTask] = useState('');
  const [swarmJson, setSwarmJson] = useState('');
  const [taskResults, setTaskResults] = useState<{ [key: string]: string }>({});
  const [isOptimizing, setIsOptimizing] = useState(false);
  const { toast } = useToast();
  const [systemPrompt, setSystemPrompt] = useState('');
  const [currentFlowId, setCurrentFlowId] = useState<string | null>(null); // Move this line above the useEnhancedAutosave call
  const saveFlowMutation = api.dnd.saveFlow.useMutation(); // Move this line above the useEnhancedAutosave call
  const { lastSaveStatus, forceSave, isSaving } = useEnhancedAutosave({
    nodes,
    edges,
    currentFlowId,
    taskResults,
    onSave: saveFlowMutation.mutateAsync,
    debounceMs: 1000, // Adjust as needed
    maxRetries: 3,
    enabled: true
  });
  const [isGenerating, setIsGenerating] = useState(false);
  const [isLoadingFlow, setIsLoadingFlow] = useState(true);
  const [hasInitialized, setHasInitialized] = useState(false);

  const [swarmArchitecture, setSwarmArchitecture] =
    useState<SwarmArchitecture>('Concurrent');
  const [popup, setPopup] = useState<{
    message: string;
    type: 'success' | 'error';
  } | null>(null);
  useBodyStyleCleanup(isCreatingGroup);
```

```
const updateGroupState = (groupId: string, update: Partial<GroupProcessingState>) => {
  setGroupProcessingStates(prev => ({
    ...prev,
    [groupId]: {
      ...prev[groupId],
      ...update
    }
  }));
};

// Add TRPC mutations and queries
const getCurrentFlowQuery = api.dnd.getCurrentFlow.useQuery(
  {
    flowId: searchParams?.get('flowId') || undefined,
  },
  {
    enabled: !!searchParams?.get('flowId'),
  },
);

useEffect(() => {
  if (getCurrentFlowQuery.data) {
    const swarmData = {
      nodes: getCurrentFlowQuery.data.nodes,
      edges: getCurrentFlowQuery.data.edges,
      architecture: getCurrentFlowQuery.data.architecture,
      results: getCurrentFlowQuery.data.results,
    };
    setSwarmJson(JSON.stringify(swarmData, null, 2));
  }
}, [getCurrentFlowQuery.data]);

const getAllFlowsQuery = api.dnd.getAllFlows.useQuery();
const setCurrentFlowMutation = api.dnd.setCurrentFlow.useMutation();
const reactFlowInstance = useReactFlow();
// Add a ref to track initial load
const initialLoadRef = useRef(false);

// Update the stateRef type and assignment
const stateRef = useRef<{
  nodes: ReactFlowNode[];
  edges: Edge[];
  taskResults: { [key: string]: string };
}>({
  nodes: [],
  edges: [],
  taskResults: {},
});

// Update the assignment
stateRef.current = {
  nodes: nodes as any,
  edges,
  taskResults,
};

// Inside the component, add these state tracking refs
const previousStateRef = useRef({
  nodes: [] as ReactFlowNode[],
  edges: [] as Edge[],
});

const handleOptimizePrompt = async () => {
  if (!systemPrompt) {
    toast({
      title: "Error",
      description: "System prompt is required for optimization",
      variant: "destructive"
    });
    return;
  }

  setIsOptimizing(true);
  try {
    const optimizedPrompt = await optimizePrompt(systemPrompt);
    setSystemPrompt(optimizedPrompt);
    toast({
      description: "System prompt optimized successfully",
    });
  } catch (error) {
    toast({
      title: "Error",
      description: "Failed to optimize system prompt",
      variant: "destructive"
    });
  }
  setIsOptimizing(false);
};

const saveInProgressRef = useRef(false);


useEffect(() => {
  const mainWrapperElements = document.getElementsByClassName('main-wrapper-all');
  const panelLayoutElements = document.getElementsByClassName('panel-layout-wrapper');
  const originalClasses: { [key: string]: string[] } = {
    mainWrapper: [],
    panelLayout: []
  };

  // Save original classes
  for (let i = 0; i < mainWrapperElements.length; i++) {
    originalClasses.mainWrapper[i] = mainWrapperElements[i].className;
  }
```

```
for (let i = 0; i < panelLayoutElements.length; i++) {
  originalClasses.panelLayout[i] = panelLayoutElements[i].className;
  // Remove specific classes from panel layout
  const classes = panelLayoutElements[i].className
    .split(' ')
    .filter(cls => !['w-screen', 'h-screen', 'min-h-screen'].includes(cls))
    .join(' ');
  panelLayoutElements[i].className = classes;
}

const timer = setTimeout(() => {
  if (mainWrapperElements.length >= 1) {
    for (let i = 0; i < mainWrapperElements.length; i++) {
      mainWrapperElements[i].className = 'main-wrapper-all spreadsheet-swarm';
    }
  }
  // Set loading to false after classes are updated
  setTimeout(() => {
    setIsLoading(false);
  }, 500);
}, 500);

// Restore original classes on unmount
return () => {
  clearTimeout(timer);
```

```
    const mainWrapperElements = document.getElementsByClassName('main-wrapper-all');
    const panelLayoutElements = document.getElementsByClassName('panel-layout-wrapper');

    for (let i = 0; i < mainWrapperElements.length; i++) {
      if (mainWrapperElements[i]) {
        mainWrapperElements[i].className = originalClasses.mainWrapper[i];
      }
    }

    for (let i = 0; i < panelLayoutElements.length; i++) {
      if (panelLayoutElements[i]) {
        panelLayoutElements[i].className = originalClasses.panelLayout[i];
      }
    }
  };
}, []);


// Add new function to handle new flow creation
const createNewFlow = useCallback(async () => {
  try {
    // Clear all states
    setNodes([]);
    setEdges([]);
    setTaskResults({});
    setSwarmArchitecture('Concurrent');
```

```
setSwarmJson('');
setTask('');
setCurrentFlowId(null);

// Reset refs
previousStateRef.current = {
  nodes: [],
  edges: [],
};
initialLoadRef.current = false;

// Save new empty flow to get an ID
const result: any = await saveFlowMutation.mutateAsync({
  nodes: [],
  edges: [],
  architecture: 'Concurrent',
  results: {},
});

if (result) {
  // Update URL with new flow ID
  const newUrl = new URL(window.location.href);
  newUrl.searchParams.set('flowId', result.id);
  router.replace(newUrl.pathname + newUrl.search);

  // Refresh flows list
```

```
      await getAllFlowsQuery.refetch();

      setPopup({ message: 'New flow created', type: 'success' });
    }
  } catch (error) {
    console.error('Error creating new flow:', error);
    setPopup({ message: 'Error creating new flow', type: 'error' });
  }
}, [router, setNodes, setEdges, saveFlowMutation, getAllFlowsQuery]);

// Update state ref whenever state changes
useEffect(() => {
  stateRef.current = {
    nodes: nodes as any[],
    edges,
    taskResults,
  };
}, [nodes, edges, taskResults]);

// Modify the useEffect for loading initial flow data
useEffect(() => {
  const flowId = searchParams?.get('flowId');

  if (flowId && getCurrentFlowQuery.data && !initialLoadRef.current) {
    // Set the flag to prevent multiple loads
    initialLoadRef.current = true;
```

```
try {
  const flowData = getCurrentFlowQuery.data;

  // Update all relevant state with the loaded flow data
  setNodes(flowData.nodes || []);
  setEdges(flowData.edges || []);
  setSwarmArchitecture(flowData.architecture || 'Concurrent');
  setTaskResults(flowData.results || {});
  setCurrentFlowId(flowId);

  // Initialize previous state
  previousStateRef.current = {
    nodes: flowData.nodes || [],
    edges: flowData.edges || [],
  };

  // Update the JSON representation
  const swarmData = {
    nodes: flowData.nodes,
    edges: flowData.edges,
    architecture: flowData.architecture,
    results: flowData.results,
  };
  setSwarmJson(JSON.stringify(swarmData, null, 2));
```

```
      setPopup({ message: 'Flow loaded successfully', type: 'success' });
    } catch (error) {
      console.error('Error loading flow data:', error);
      setPopup({ message: 'Error loading flow data', type: 'error' });
    }
  }
}, [searchParams, getCurrentFlowQuery.data]);

// Add new type for group processing state
interface GroupProcessingState {
  isProcessing: boolean;
  completedAgents: string[];
  result?: string;
  error?: string;
}

// Add processing state tracking
const [groupProcessingStates, setGroupProcessingStates] = useState<{
  [groupId: string]: GroupProcessingState;
}>({});

// Add function to check for circular connections
const hasCircularConnection = (
  source: string,
  target: string,
  visitedGroups = new Set<string>()
```

```
): boolean => {
  if (visitedGroups.has(target)) return true;
  visitedGroups.add(source);

  const outgoingEdges = edges.filter(edge => edge.source === target);
  return outgoingEdges.some(edge =>
    hasCircularConnection(target, edge.target, new Set(visitedGroups))
  );
};

// Modify onConnect to handle group connections
const onConnect = useCallback(
  (params: Connection) => {
    if (!params.source || !params.target) return;

    // Get source and target nodes
    const sourceNode = nodes.find(n => n.id === params.source);
    const targetNode = nodes.find(n => n.id === params.target);

    // Only allow connections between groups
    if (sourceNode?.type !== 'group' || targetNode?.type !== 'group') {
      setPopup({
        message: 'Only groups can be connected to other groups',
        type: 'error'
      });
      return;
    }


    // Check for circular connections
    if (hasCircularConnection(params.source, params.target)) {
      setPopup({
        message: 'Circular connections between groups are not allowed',
        type: 'error'
      });
      return;
    }


    const newEdge: Edge = {
      id: `e${params.source}-${params.target}`,
      source: params.source,
      target: params.target,
      type: 'custom', // Set the edge type to custom
      animated: true,
      style: { stroke: '#8E8E93' },
      markerEnd: {
        type: MarkerType.ArrowClosed,
        color: '#8E8E93',
      },
      data: { label: 'Group Flow' },
    };


    setEdges((eds) => addEdge(newEdge, eds));
```

```
  },
  [nodes, edges, setEdges]
);

const addAgent = useCallback(
  (agent: AgentData) => {
    const newNode: ReactFlowNode = {
      id: `agent-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`, // Generate unique ID
      type: 'agent',
      position: {
        // Get viewport center for better positioning
        x: Math.random() * 500,
        y: Math.random() * 300
      },
      data: agent,
    };

    if (agent.type === 'Boss') {
      const clusterId = `cluster-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`;
      newNode.data = { ...newNode.data, clusterId };
    } else if (agent.type === 'Worker') {
      const availableBosses = nodes.filter((node) => {
        if (node.data.type !== 'Boss') return false;
        const workerCount = nodes.filter(
          (n) => n.data?.clusterId === node.data?.clusterId,
        ).length;
  return workerCount < 3;
});

if (availableBosses.length > 0) {
  const closestBoss = availableBosses.reduce((prev, curr) => {
    const prevDistance =
      Math.abs(prev.position.x - newNode.position.x) +
      Math.abs(prev.position.y - newNode.position.y);
    const currDistance =
      Math.abs(curr.position.x - newNode.position.x) +
      Math.abs(curr.position.y - newNode.position.y);
    return currDistance < prevDistance ? curr : prev;
  });

  if (closestBoss.data.clusterId) {
    newNode.data = {
      ...newNode.data,
      clusterId: closestBoss.data.clusterId,
    };

    // Add edge to connect worker to boss
    const newEdge: Edge = {
      id: `e${closestBoss.id}-${newNode.id}`,
      source: closestBoss.id,
      target: newNode.id,
      type: 'custom',
            animated: true,
            style: { stroke: '#8E8E93' },
            markerEnd: {
              type: MarkerType.ArrowClosed,
              color: '#8E8E93',
            },
            data: { label: 'Hierarchy' },
          };

          setEdges((eds) => [...eds, newEdge]);
        }
      }
    }

    // Use functional update to preserve existing nodes
    setNodes((prevNodes) => [...prevNodes, newNode]);
  },
  [nodes, setNodes, setEdges],
);

const updateNodeData = (id: string, updatedData: AgentData) => {
  setNodes((nds: Node<ReactFlowNode[], string | undefined>[]) =>
    nds.map((node) =>
      node.id === id
        ? { ...node, data: { ...node.data, ...updatedData } }
        : node,
    ),
  );
  //saveVersion();
};

// Expose the updateNodeData function to the window object
useEffect(() => {
  (window as any).updateNodeData = updateNodeData;
}, []);

// Modify runTask to handle both grouped and standalone agents
const runTask = async () => {
  try {
    // Reset all states
    setGroupProcessingStates({});
    setTaskResults({});

    // 1. Process standalone agents based on swarm architecture
    const standaloneAgents = nodes.filter(node =>
      node.type === 'agent' && !node.data.groupId
    );

    let standaloneResults: { id: string; result: string }[] = [];

    if (standaloneAgents.length > 0) {
      switch (swarmArchitecture) {
```

```
case 'Concurrent':
  standaloneResults = await Promise.all(
    standaloneAgents.map(async (agent) => {
      const { text } = await generateText({
        model: registry.languageModel(`openai:${agent.data.model}`),
        prompt: `${agent.data.systemPrompt}

        Task: ${task}

        Response:`,
      });
      return { id: agent.id, result: text };
    })
  );
  break;

case 'Sequential':
  let context = '';
  for (const agent of standaloneAgents) {
    const { text } = await generateText({
      model: registry.languageModel(`openai:${agent.data.model}`),
      prompt: `${agent.data.systemPrompt}

      Previous context: ${context}

      Task: ${task}
      Response:`,
    });
    standaloneResults.push({ id: agent.id, result: text });
    context += `\n${agent.data.name}: ${text}`;
  }
  break;

case 'Hierarchical':
  const bosses = standaloneAgents.filter(node => node.data.type === 'Boss');
  const workers = standaloneAgents.filter(node => node.data.type === 'Worker');

  // Process bosses first
  const bossPrompts = await Promise.all(
    bosses.map(async (boss) => {
      const { text } = await generateText({
        model: registry.languageModel(`openai:${boss.data.model}`),
        prompt: `${boss.data.systemPrompt}

        You are a Boss agent. Create a subtask based on the following main task:

        Task: ${task}

        Subtask for your team:`,
      });
      standaloneResults.push({ id: boss.id, result: text });
```

```
        return { bossId: boss.id, subtask: text };
      })
    );

    // Then process workers
    await Promise.all(
      workers.map(async (worker) => {
        const boss = bosses.find(b => b.data.clusterId === worker.data.clusterId);
        if (!boss) return null;

        const bossPrompt = bossPrompts.find(bp => bp.bossId === boss.id);
        if (!bossPrompt) return null;

        const { text } = await generateText({
          model: registry.languageModel(`openai:${worker.data.model}`),
          prompt: `${worker.data.systemPrompt}

          Task from your boss: ${bossPrompt.subtask}

          Response:`,
        });
        standaloneResults.push({ id: worker.id, result: text });
      })
    );
    break;
}
```

```
  // Store standalone results
  standaloneResults.forEach(({ id, result }) => {
    setTaskResults(prev => ({
      ...prev,
      [id]: result
    }));
  });
}

// 2. Process connected groups
const startingGroups = nodes.filter(node =>
  node.type === 'group' &&
  !edges.some(edge => edge.target === node.id)
);

if (startingGroups.length > 0) {
  // Process each starting group chain sequentially
  for (const startGroup of startingGroups) {
    await processGroupChain(startGroup.id, task);
  }
}


setPopup({
  message: 'Task completed successfully',
        type: 'success'
      });

      await forceSave(); // Save the new results

    } catch (error) {
      console.error('Error running task:', error);
      setPopup({
        message: 'Error processing task',
        type: 'error'
      });
    }
  };


  // Keep the existing processGroupChain function as is
  const processGroupChain = async (groupId: string, currentTask: string, previousResults: string = '', processedGroups = new Set<string>()) => {
    // Prevent infinite loops and reprocessing
    if (processedGroups.has(groupId)) {
      return;
    }
    processedGroups.add(groupId);

    try {
      updateGroupState(groupId, {
```

```
  isProcessing: true,
  completedAgents: [],
});

const group = nodes.find(n => n.id === groupId);
if (!group || group.type !== 'group') return;

// Create a more detailed context from previous results
const contextPrompt = previousResults
  ? `Previous Team's Results:
     ${previousResults}

     Using these results as context, your team should build upon this work.

     Main Task: ${currentTask}`
  : `You are the first team working on this task.

     Main Task: ${currentTask}`;

// Process all agents within the group
const results = await Promise.all(
  group.data.agents.map(async (agent: AgentData) => {
    const { text } = await generateText({
      model: registry.languageModel(`openai:${agent.model}`),
      prompt: `${agent.systemPrompt}
      Context and Task:

      ${contextPrompt}


      Your Role: ${agent.type}

      Team: ${group.data.teamName}

      Team Type: ${group.data.swarmType}


      Provide your specialized contribution based on the context and your role.`,

    });


    // Update the agent's lastResult in the nodes state

    setNodes(nodes => nodes.map(node => {

      if (node.type === 'group' && node.id === groupId) {

        return {

          ...node,

          data: {

            ...node.data,

            agents: node.data.agents.map((a: AgentData) =>

              a.id === agent.id ? { ...a, lastResult: text } : a

            )

          }

        };

      }

      return node;

    }));
```

```
        updateGroupState(groupId, {
          completedAgents: [...(groupProcessingStates[groupId]?.completedAgents || []), agent.id],
        });

        return { agentId: agent.id, result: text };
      })
    );

    // Combine results from all agents in the group
    const groupResult = `Team ${group.data.teamName} Results:\n${results.map(r => r.result).join('\n\n')}`;

    // Update group state with results
    updateGroupState(groupId, {
      isProcessing: false,
      result: groupResult,
    });

    // Store individual agent results
    results.forEach(({ agentId, result }) => {
      setTaskResults(prev => ({
        ...prev,
        [agentId]: result,
      }));
    });
    // Find next groups in chain
    const nextGroups = edges
      .filter(edge => edge.source === groupId)
      .map(edge => edge.target);

    // Process all next groups in parallel
    await Promise.all(
      nextGroups.map(nextGroupId =>
        processGroupChain(nextGroupId, currentTask, groupResult, processedGroups)
      )
    );

  } catch (error) {
    console.error(`Error processing group ${groupId}:`, error);
    updateGroupState(groupId, {
      isProcessing: false,
      error: 'Error processing group',
    });
  }
};

// Update GroupNode component to show processing state
const GroupNode: React.FC<NodeProps<GroupData>> = ({ data, id }) => {
  const { groupProcessingStates } = useContext(GroupNodeContext);
  const processingState = groupProcessingStates[id];
  return (
    <div className="min-w-[300px] min-h-[200px] relative">
      {/* Existing group node code ... */}

      {/* Add processing indicator */}
      {processingState?.isProcessing && (
        <div className="absolute inset-0 bg-primary/20 backdrop-blur-sm flex items-center justify-center">
          <div className="text-primary-foreground">Processing...</div>
        </div>
      )}

      {/* Add completion indicator */}
      {processingState?.result && (
        <div className="absolute top-2 right-2">
          <div className="bg-primary text-primary-foreground rounded-full w-6 h-6 flex items-center justify-center">

          </div>
        </div>
      )}

      {/* Show group result if available */}
      {processingState?.result && (
        <div className="mt-2 p-2 bg-muted/50 rounded text-sm">
          <strong>Group Result:</strong>
          <div className="max-h-20 overflow-y-auto">
            {processingState.result}
          </div>
        </div>
      )}

      {/* Existing group node content ... */}
    </div>
  );
};

// Expose the updateNodeData function to the window object
useEffect(() => {
  (window as any).updateNodeData = updateNodeData;
}, []);


 // Add this function inside the component
 const loadVersion = async (flowId: string) => {
  try {
    // Prevent loading if a save is in progress
    if (saveInProgressRef.current) {
      setPopup({
        message: 'Please wait for current save to complete',
        type: 'error',
      });
  return;
}

// Set current flow as active
await setCurrentFlowMutation.mutateAsync({ flow_id: flowId });

// Update URL with new flow ID
const newUrl = new URL(window.location.href);
newUrl.searchParams.set('flowId', flowId);
router.replace(newUrl.pathname + newUrl.search);

// Fetch the specific flow data directly with the flowId
const { data: flowData } = await getCurrentFlowQuery.refetch({
  //    queryKey: ['dnd.getCurrentFlow', { flowId }]
});

if (flowData) {
  // Update all state
  setNodes(flowData.nodes || []);
  setEdges(flowData.edges || []);
  setSwarmArchitecture(flowData.architecture || 'Concurrent');
  setTaskResults(flowData.results || {});
  setCurrentFlowId(flowId);

  // Update previous state to prevent immediate save
  previousStateRef.current = {
        nodes: flowData.nodes || [],
        edges: flowData.edges || [],
      };

      // Update JSON representation
      const swarmData = {
        nodes: flowData.nodes,
        edges: flowData.edges,
        architecture: flowData.architecture,
        results: flowData.results,
      };
      setSwarmJson(JSON.stringify(swarmData, null, 2));

      // Reset save in progress flag
      saveInProgressRef.current = false;

      setPopup({
        message: 'Flow version loaded successfully',
        type: 'success',
      });
    } else {
      throw new Error('No flow data found');
    }
  } catch (error) {
    console.error('Error loading flow version:', error);
    setPopup({ message: 'Error loading flow version', type: 'error' });
    // Reset save in progress flag on error
    saveInProgressRef.current = false;
  }
};



const updateSwarmJson = () => {
  const swarmData = {
    nodes: nodes.map((node) => ({
      id: node.id,
      type: node.type,
      data: node.data,
      position: node.position,
    })),
    edges: edges,
    architecture: swarmArchitecture,
    results: taskResults,
  };
  setSwarmJson(JSON.stringify(swarmData, null, 2));
};

const saveSwarmConfiguration = () => {
  updateSwarmJson();
  const blob = new Blob([swarmJson], { type: 'application/json' });
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a');
  a.href = url;
  a.download = 'swarm-configuration.json';
  a.click();
  URL.revokeObjectURL(url);
};

const shareSwarmConfiguration = () => {
  updateSwarmJson();
  navigator.clipboard.writeText(swarmJson).then(
    () => {
      setPopup({
        message: 'Swarm configuration copied to clipboard',
        type: 'success',
      });
    },
    (err) => {
      console.error('Could not copy text: ', err);
      setPopup({ message: 'Failed to copy configuration', type: 'error' });
    },
  );
};

const loadSwarmConfiguration = (
  event: React.ChangeEvent<HTMLInputElement>,
) => {
  const file = event.target.files?.[0];
  if (file) {

    const reader = new FileReader();

    reader.onload = (e) => {

      try {

        const content = e.target?.result as string;

        const swarmData = JSON.parse(content);

        setNodes(swarmData.nodes);

        setEdges(swarmData.edges);

        setSwarmArchitecture(swarmData.architecture || 'Concurrent');

        setTaskResults(swarmData.results || {});

        updateSwarmJson();

        //saveVersion()

        setPopup({

          message: 'Swarm configuration loaded successfully',

          type: 'success',

        });

      } catch (error) {

        console.error('Error parsing JSON:', error);

        setPopup({ message: 'Invalid JSON file', type: 'error' });

      }

    };

    reader.readAsText(file);

  }

};


const saveVersionStable = useCallback(async () => {
```

```
try {
  // Transform nodes to match the expected schema
  const validNodes: SaveFlowNode[] = nodes.map((node) => {
    const { id, type, position, data } = node;

    // Set default values for required fields if they're missing
    const validatedData = {
      id: data?.id || id,
      name: data?.name || '',
      type: data?.type || type || 'default', // Ensure type is never null
      model: data?.model || '',
      systemPrompt: data?.systemPrompt || '',
      clusterId: data?.clusterId,
      isProcessing: data?.isProcessing || false,
      lastResult: data?.lastResult || '',
      dataSource: data?.dataSource,
      dataSourceInput: data?.dataSourceInput,
      ...data,
    };

    return {
      id,
      type: type || 'default', // Ensure node type is never null
      position: {
        x: position.x,
        y: position.y,
    },
    data: validatedData,
  };
});

// Rest of the function remains the same...
const validEdges: any[] = edges.map((edge) => {
  const { id, source, target, ...rest } = edge;
  return {
    id,
    source,
    target,
    type: rest.type || 'default',
    animated: rest.animated,
    style: rest.style
      ? {
          stroke: rest.style.stroke || '#000000',
        }
      : undefined,
    markerEnd: rest.markerEnd
      ? {
          type:
            typeof rest.markerEnd === 'string'
              ? rest.markerEnd
              : (rest.markerEnd as any)?.type || 'arrow',
          color: (rest.markerEnd as any)?.color || '#000000',
              }
            : undefined,
          data: rest.data || { label: 'Connection' },
        };
      });

      const flowData = {
        flow_id: currentFlowId || undefined,
        nodes: validNodes,
        edges: validEdges,
        architecture: swarmArchitecture,
        results: taskResults || {},
      } as const;

      const result: any = await saveFlowMutation.mutateAsync(flowData);

      if (!currentFlowId && result.id) {
        const newUrl = new URL(window.location.href);
        newUrl.searchParams.set('flowId', result.id);
        router.replace(newUrl.pathname + newUrl.search);
        setCurrentFlowId(result.id);
      }

      setPopup({ message: 'Flow saved successfully', type: 'success' });
      await getAllFlowsQuery.refetch();
    } catch (error) {
      console.error('Error saving flow:', error);
      setPopup({ message: 'Failed to save flow', type: 'error' });
    }
    }, [nodes, edges, swarmArchitecture, taskResults, saveFlowMutation, router, currentFlowId, getAllFlowsQuery]);



  const useSaveShortcuts = () => {
    useEffect(() => {
      const handleKeyPress = (e: KeyboardEvent) => {
        if ((e.metaKey || e.ctrlKey) && e.key === 's') {
          e.preventDefault();
          forceSave();
        }
      };

      window.addEventListener('keydown', handleKeyPress);
      return () => window.removeEventListener('keydown', handleKeyPress);
    }, [forceSave]);
  };

  const useUnsavedChangesWarning = () => {
    useEffect(() => {
      const handleBeforeUnload = (e: BeforeUnloadEvent) => {
        if (isSaving) {
```

```
        const message = 'Changes are being saved. Are you sure you want to leave?';
        e.returnValue = message;
        return message;
      }
      if (lastSaveStatus === 'error') {
        const message = 'You have unsaved changes. Are you sure you want to leave?';
        e.returnValue = message;
        return message;
      }
    };

    window.addEventListener('beforeunload', handleBeforeUnload);
    return () => window.removeEventListener('beforeunload', handleBeforeUnload);
  }, [isSaving, lastSaveStatus]);
};


useSaveShortcuts();
useUnsavedChangesWarning();

// Update popup handling to show save status
useEffect(() => {
  if (lastSaveStatus === 'error') {
    setPopup({
      message: 'Failed to save changes. Click save to retry.',
      type: 'error'
    });
  }
}, [lastSaveStatus]);




// Replace the existing save-related code with this implementation
const debouncedSave = useMemo(
  () =>
    debounce(async () => {
      // Prevent concurrent saves
      if (saveInProgressRef.current) {
        return;
      }

      const currentNodes = stateRef.current.nodes;
      const currentEdges = stateRef.current.edges;

      // Check if there are actual changes
      const hasChanges =
        !isEqual(previousStateRef.current.nodes, currentNodes) ||
        !isEqual(previousStateRef.current.edges, currentEdges);

      if (!hasChanges || !currentFlowId) {
        return;
      }

      try {
        saveInProgressRef.current = true;

        // Update previous state before saving
        previousStateRef.current = {
          nodes: JSON.parse(JSON.stringify(currentNodes)),
          edges: JSON.parse(JSON.stringify(currentEdges)),
        };

        await saveVersionStable();
      } finally {
        saveInProgressRef.current = false;
      }
    }, 2000),
  [currentFlowId, saveVersionStable],
);

// Update the effect that triggers saves
useEffect(() => {
  if (!currentFlowId || (nodes.length === 0 && edges.length === 0)) {
    return;
  }

  // Update current state ref
  stateRef.current = {
    nodes: nodes,
    edges,
    taskResults,
  };

  // Only trigger save if not the initial load
  if (previousStateRef.current.nodes.length > 0) {
    debouncedSave();
  } else {
    // Initialize previous state on first load
    previousStateRef.current = {
      nodes: JSON.parse(JSON.stringify(nodes)),
      edges: JSON.parse(JSON.stringify(edges)),
    };
  }

  return () => {
    debouncedSave.cancel();
  };
}, [nodes, edges, currentFlowId, debouncedSave, taskResults]);

useEffect(() => {
  if (popup) {
    const timer = setTimeout(() => {
      setPopup(null);
    }, 3000);
    return () => clearTimeout(timer);
  }
}, [popup]);

// Add share link functionality
const shareFlowLink = () => {
  if (currentFlowId) {
    const shareUrl = new URL(window.location.href);
    shareUrl.searchParams.set('flowId', currentFlowId);
    navigator.clipboard.writeText(shareUrl.toString());
    setPopup({ message: 'Flow link copied to clipboard', type: 'success' });
  }
};

// Add error state handling
if (getCurrentFlowQuery.isError) {
  return (
    <div className="w-full h-screen flex items-center justify-center">
      <div className="flex flex-col items-center space-y-4">
        <div className="text-red-500 text-xl">Error loading flow</div>
        <Button onClick={() => router.push('/')}>Return to Home</Button>
      </div>
    </div>
  );
}
```

```
const createGroup = (groupData: Omit<GroupData, 'agents'>) => {
  // Get viewport info safely
  const viewport = reactFlowInstance?.getViewport();

  // Default position if viewport is not available
  let position = { x: 100, y: 100 };

  if (viewport) {
    // Calculate center position with safety checks
    const centerX = (-viewport.x / viewport.zoom) + ((window.innerWidth / 2) / viewport.zoom);
    const centerY = (-viewport.y / viewport.zoom) + ((window.innerHeight / 2) / viewport.zoom);

    position = {
      x: Math.max(0, centerX - 150), // Prevent negative positions
      y: Math.max(0, centerY - 100)
    };
  }

  // Create new group with unique ID
  const newGroup = {
    id: `group-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`,
    type: 'group',
    position,
    data: {
      ...groupData,
      agents: [],
    },
    style: {
      width: 300,
      height: 200,
    },
  };

  // Update nodes state using functional update
  setNodes(prevNodes => [...prevNodes, newGroup]);
  setIsCreatingGroup(false);
};

// Add the updateGroupData function
// eslint-disable-next-line react-hooks/rules-of-hooks
useEffect(() => {
  window.updateGroupData = (id: string, updatedData: GroupData) => {
    setNodes((nds) =>
      nds.map((node) =>
        node.id === id
          ? { ...node, data: { ...node.data, ...updatedData } }
          : node,
      ),
    );
  };
  window.addAgentToGroup = (groupId: string) => {
    setAddToGroupDialogState({
      open: true,
      groupId
    });
  };
}, [setNodes]);

// Replace the existing Versions TabsContent with the new component
return (
  <div className="w-full h-[calc(100%-10px)] flex flex-col bg-background text-foreground">
    {/* Header */}
    <div className="flex justify-between items-center p-4 border-b border-border">
      <h1 className="text-2xl font-semibold">Swarms No-Code Builder</h1>
      <div className="flex space-x-2">
        <AutoGenerateSwarm
          addAgent={addAgent}
          setPopup={setPopup}
          reactFlowInstance={reactFlowInstance}
        />
        <Button
          variant="outline"
          className="bg-card hover:bg-muted"
          onClick={createNewFlow}
          disabled={saveFlowMutation.isPending}
        >
          <Plus className="w-4 h-4 mr-2" />
          New Swarm
        </Button>

        <Button
          variant="outline"
          className="bg-card hover:bg-muted"
          onClick={() => setIsCreatingGroup(true)}
        >
          <Plus className="w-4 h-4 mr-2" />
          New Team
        </Button>

        <Dialog>
          <DialogTrigger asChild>
            <Button variant="outline" className="bg-card hover:bg-muted">
              <Plus className="w-4 h-4 mr-2" />
              Add Agent
            </Button>
          </DialogTrigger>
          <DialogContent className="sm:max-w-[425px]">
            <DialogHeader>
              <DialogTitle>Add New Agent</DialogTitle>
              <DialogDescription>
                Create a new agent to add to your swarm.
              </DialogDescription>
```

```
</DialogHeader>
<form
  onSubmit={(e: any) => {
    e.preventDefault();
    const formData = new FormData(e.target as HTMLFormElement);
    addAgent({
      id: `${nodes.length + 1}`,
      name: formData.get('name') as string,
      type: formData.get('type') as AgentType,
      model: formData.get('model') as AgentModel,
      systemPrompt: formData.get('systemPrompt') as string,
      description: '',
    });
  }}
>
  <div className="grid gap-4 py-4">
    <div className="grid grid-cols-4 items-center gap-4">
      <Label htmlFor="name" className="text-right">
        Name
      </Label>
      <Input id="name" name="name" className="col-span-3" />
    </div>
    <div className="grid grid-cols-4 items-center gap-4">
      <Label htmlFor="type" className="text-right">
        Type
      </Label>
```

```
</div>

<div className="grid grid-cols-4 items-center gap-4">

  <Label htmlFor="model" className="text-right">

    Model

  </Label>

  <Select name="model">

    <SelectTrigger className="col-span-3">

      <SelectValue placeholder="Select model" />

    </SelectTrigger>

    <SelectContent>

      <SelectItem value="gpt-3.5-turbo">

        GPT-3.5 Turbo

      </SelectItem>

      <SelectItem value="gpt-4o">GPT-4o</SelectItem>

      <SelectItem value="gpt-4o-mini">GPT-4o-Mini</SelectItem>

      <SelectItem value="claude-2">Claude 2</SelectItem>

    </SelectContent>

  </Select>

</div>

<div className="grid grid-cols-4 items-center gap-4">

<Label htmlFor="systemPrompt" className="text-right">

  System Prompt

</Label>

<div className="relative col-span-3">

  <Textarea

    id="systemPrompt"
```

```
      name="systemPrompt"
      value={systemPrompt}
      onChange={(e) => setSystemPrompt(e.target.value)}
      className="pr-12"
    />
    <Button
      type="button"
      size="sm"
      variant="ghost"
      onClick={handleOptimizePrompt}
      disabled={isOptimizing || !systemPrompt}
      title={!systemPrompt ? "System prompt required" : "Optimize prompt"}
      className="absolute right-2 top-2 h-8 w-8 p-0"
    >
      {isOptimizing ? (
        <Loader2 className="h-4 w-4 animate-spin" />
      ) : (
        <Sparkles className="h-4 w-4" />
      )}
    </Button>
  </div>
</div>
</div>
<DialogFooter>
  <Button type="submit">Add Agent</Button>
</DialogFooter>
```

```
    </form>
  </DialogContent>
</Dialog>
<DropdownMenu>
  <DropdownMenuTrigger asChild>
    <Button variant="outline" className="bg-card hover:bg-muted">
      <MoreHorizontal className="w-4 h-4 mr-2" />
      Options
    </Button>
  </DropdownMenuTrigger>
  <DropdownMenuContent>
    <DropdownMenuLabel>Swarm Configuration</DropdownMenuLabel>
    <DropdownMenuSeparator />
    <DropdownMenuItem onClick={saveSwarmConfiguration}>
      <Save className="w-4 h-4 mr-2" />
      Save JSON
    </DropdownMenuItem>
    <DropdownMenuItem onClick={shareSwarmConfiguration}>
      <Share className="w-4 h-4 mr-2" />
      Share JSON
    </DropdownMenuItem>
    <DropdownMenuItem>
      <label
        htmlFor="load-json"
        className="flex items-center cursor-pointer"
      >
```

```
        <Upload className="w-4 h-4 mr-2" />
        Load JSON
      </label>
    </DropdownMenuItem>
    <DropdownMenuItem onClick={shareFlowLink}>
      <Share className="w-4 h-4 mr-2" />
      Share Link
    </DropdownMenuItem>
  </DropdownMenuContent>
</DropdownMenu>
<input
  id="load-json"
  type="file"
  accept=".json"
  className="hidden"
  onChange={loadSwarmConfiguration}
/>
<Select
  value={swarmArchitecture}
  onValueChange={(value: SwarmArchitecture) =>
    setSwarmArchitecture(value)
  }
>
  <SelectTrigger className="w-[180px] bg-card border-border">
    <SelectValue placeholder="Select architecture" />
  </SelectTrigger>
```

```
      <SelectContent>
        <SelectItem value="Concurrent">Concurrent</SelectItem>
        <SelectItem value="Sequential">Sequential</SelectItem>
        <SelectItem value="Hierarchical">Hierarchical</SelectItem>
      </SelectContent>
    </Select>
  </div>
</div>

{/* Add this Dialog */}
<Dialog open={isCreatingGroup} onOpenChange={setIsCreatingGroup}>
  <DialogContent className="sm:max-w-[425px]">
    <DialogHeader>
      <DialogTitle>Create Team</DialogTitle>
    </DialogHeader>
    <form onSubmit={(e) => {
      e.preventDefault();
      const formData = new FormData(e.target as HTMLFormElement);
      createGroup({
        teamName: formData.get('teamName') as string,
        swarmType: formData.get('swarmType') as string,
        description: formData.get('description') as string,
      });
    }}>
      <div className="grid gap-4 py-4">
        <div className="grid grid-cols-4 items-center gap-4">
  <Label htmlFor="teamName" className="text-right">Team Name</Label>

  <Input

    id="teamName"

    name="teamName"

    className="col-span-3"

    required

  />

</div>

<div className="grid grid-cols-4 items-center gap-4">

  <Label htmlFor="swarmType" className="text-right">Swarm Type</Label>

  <Select name="swarmType" required>

    <SelectTrigger className="col-span-3">

      <SelectValue placeholder="Select type" />

    </SelectTrigger>

    <SelectContent>

      <SelectItem value="Marketing">Marketing</SelectItem>

      <SelectItem value="Research">Research</SelectItem>

      <SelectItem value="Development">Development</SelectItem>

      <SelectItem value="Analytics">Analytics</SelectItem>

      <SelectItem value="Custom">Custom</SelectItem>

    </SelectContent>

  </Select>

</div>

<div className="grid grid-cols-4 items-center gap-4">

  <Label htmlFor="description" className="text-right">Description</Label>

  <Textarea
```

```
            id="description"
            name="description"
            className="col-span-3"
          />
        </div>
      </div>
      <DialogFooter>
        <Button type="submit">Create Group</Button>
      </DialogFooter>
    </form>
  </DialogContent>
</Dialog>
<div className="flex-grow flex overflow-hidden">
  {/* Sidebar */}
  <div className="w-96 border-r border-border p-4 overflow-hidden bg-background">
    <Tabs defaultValue="results" className="h-full">
      <TabsList className="grid w-full grid-cols-2">
        <TabsTrigger value="results">Results</TabsTrigger>
        <TabsTrigger value="versions">Swarm History</TabsTrigger>
      </TabsList>
      <TabsContent value="results">
        <h2 className="text-lg font-semibold mb-4">Task Results</h2>
        <div className="h-[calc(100vh-340px)] overflow-y-auto">
          <Table>
            <TableHeader>
              <TableRow>
      <TableHead>Agent</TableHead>
      <TableHead>Group</TableHead>
      <TableHead>Result</TableHead>
    </TableRow>
  </TableHeader>
  <TableBody>
    {Object.entries(taskResults).map(([agentId, result]) => {
      // Find the agent either as a standalone node or within a group
      const agentNode = nodes.find(node =>
        node.id === agentId ||
        (node.type === 'group' && node.data.agents?.some((a:any) => a.id === agentId))
      );

      const agent = agentNode?.type === 'group'
        ? agentNode.data.agents?.find((a:any) => a.id === agentId)
        : agentNode?.data;

      const groupName = agentNode?.type === 'group'
        ? agentNode.data.teamName
        : '-';

      return (
        <TableRow key={agentId}>
          <TableCell className="font-medium">
            {agent?.name || 'Unknown Agent'}
          </TableCell>
```

```
              <TableCell>{groupName}</TableCell>
              <TableCell>{result}</TableCell>
            </TableRow>
          );
        })}
      </TableBody>
    </Table>
  </div>
</TabsContent>
<TabsContent value="versions">
  <h2 className="text-lg font-semibold mb-4">Flows</h2>
  <div className="h-[calc(100vh-340px)] overflow-y-auto">
    <Table>
      <TableHeader>
        <TableRow>
          <TableHead>Swarms</TableHead>
          <TableHead>Date</TableHead>
          <TableHead>Action</TableHead>
        </TableRow>
      </TableHeader>
      <TableBody>
        {getAllFlowsQuery.data?.map((flow: any) => (
          <TableRow key={flow.id}>
            <TableCell>{flow.id}</TableCell>
            <TableCell>
              {new Date(flow.created_at).toLocaleString()}
```

```
                </TableCell>
                <TableCell>
                  <Button
                    variant="ghost"
                    onClick={() => loadVersion(flow.id)}
                    disabled={
                      saveFlowMutation.status === 'pending' ||
                      setCurrentFlowMutation.status === 'pending'
                    }
                  >
                    Load
                  </Button>
                </TableCell>
              </TableRow>
            ))}
          </TableBody>
        </Table>
      </div>
    </TabsContent>
  </Tabs>
</div>

{/* Flow area */}
<div className="flex-1 relative">
  <GroupNodeContext.Provider value={{ groupProcessingStates }}>
    <ReactFlow
```

```
        nodes={nodes}
        edges={edges}
        onNodesChange={onNodesChange}
        onEdgesChange={onEdgesChange}
        onConnect={onConnect}
        nodeTypes={nodeTypes}
        edgeTypes={edgeTypes}
        fitView
      >
        <Background
          color={document?.documentElement.classList.contains('dark') ? "#333" : "#ccc"}
          gap={16}
        />
        <Controls />
      </ReactFlow>
    </GroupNodeContext.Provider>
  </div>
</div>

{/* Footer */}
<div className="p-4 border-t border-border flex justify-center items-center">
  <Input
    type="text"
    placeholder="Enter a task for the swarm..."
    value={task}
    onChange={(e) => setTask(e.target.value)}
```

```
          className="w-1/2 mr-2 bg-input text-foreground placeholder:text-muted-foreground"
        />
        <Button
          onClick={runTask}
          className="bg-primary text-primary-foreground hover:bg-primary/90"
        >
          <Send className="w-4 h-4 mr-2" />
          Run Task
        </Button>
      </div>
      <AnimatePresence>
        {popup && (
          <motion.div
            initial={{ opacity: 0, y: -50 }}
            animate={{ opacity: 1, y: 0 }}
            exit={{ opacity: 0, y: -50 }}
            className={`fixed top-4 right-4 p-4 rounded-md shadow-md max-w-[500px] overflow-hidden text-ellipsis ${
              popup.type === 'success'
                ? 'bg-emerald-600 dark:bg-emerald-500 text-white'
                : 'bg-destructive text-destructive-foreground'
            }`}
          >
            {popup.message}
          </motion.div>
        )}
```

```
</AnimatePresence>
<AddAgentToGroupDialog
  groupId={addToGroupDialogState.groupId || ''}
  open={addToGroupDialogState.open}
  onOpenChange={(open) => setAddToGroupDialogState(prev => ({ ...prev, open }))}
  nodes={nodes}
  setNodes={setNodes}
/>
<style jsx global>{`
  /* Light mode styles */
  .react-flow__node {
    background: transparent !important;
    color: hsl(var(--card-foreground));
  }

  .react-flow__node-default {
    background: transparent !important;
  }

  /* Dark mode styles */
  .dark .react-flow__node {
    background: transparent !important;
    color: hsl(var(--card-foreground));
  }

  .dark .react-flow__node-default {
```

```
  background: transparent !important;
}

/* Node handle styles */
.react-flow__handle {
  background: hsl(var(--muted));
  border-color: hsl(var(--border));
}

.dark .react-flow__handle {
  background: white;
  border-color: white;
}

/* Edge styles */
.react-flow__edge-path {
  stroke: hsl(var(--border));
}

.dark .react-flow__edge-path {
  stroke: white;
}

/* Controls - Light mode */
.react-flow__panel.react-flow__controls {
  background: hsl(var(--card));
  border: 1px solid hsl(var(--border));
  border-radius: 6px;
  padding: 4px;
}

.react-flow__controls-button {
  background: hsl(var(--card)) !important;
  border: 1px solid hsl(var(--border)) !important;
}

.react-flow__controls-button:hover {
  background: hsl(var(--muted)) !important;
}

.react-flow__controls-button svg path {
  fill: hsl(var(--foreground)) !important;
}

/* Controls - Dark mode */
.dark .react-flow__panel.react-flow__controls {
  background: #0F0F10;
}

.dark .react-flow__controls-button {
  background: #0F0F10 !important;
  border: 1px solid white !important;
```

```
}

.dark .react-flow__controls-button:hover {
  background: #1A1A1B !important;
}

.dark .react-flow__controls-button svg path {
  fill: white !important;
}

/* Ensure proper spacing between buttons */
.react-flow__controls-button + .react-flow__controls-button {
  margin-top: 4px;
}

/* Group Node Styles */
.react-flow__node-group {
  background: transparent;
  border: none;
  width: auto !important;
  height: auto !important;
}

.react-flow__node-group .react-flow__handle {
  width: 8px;
  height: 8px;
  border-radius: 50%;
  background-color: hsl(var(--primary));
  border: 2px solid hsl(var(--background));
}

.react-flow__node-group .react-flow__handle-left {
  left: -4px;
}

.react-flow__node-group .react-flow__handle-right {
  right: -4px;
}

/* Dialog and form styles */
.dialog-content {
  text-align: left;
}

.select-trigger {
  text-align: left;
}

.select-content {
  text-align: left;
}
```

```css
/* Node hover card styles */
.react-flow__node-default {
  text-align: left;
}

.react-flow__node-group {
  text-align: left;
}

/* Form field styles */
.form-label {
  text-align: left;
}

.form-input {
  text-align: left;
}

/* Dialog header styles */
.dialog-header {
  text-align: left;
}

/* Select dropdown styles */
.select-dropdown {
  text-align: left;
```

```
        }
      `}</style>
    </div>
  );
}

// Update the main component to use the provider
export function EnhancedAgentSwarmManagementComponent() {
  return (
    <ReactFlowProvider>
      <FlowContent />
    </ReactFlowProvider>
  );
}
```