Frontend Development Handoff Document

Swarm Multi-Agent System

Project: Swarm Multi-Agent AI System
Version: 3.0.0
Date: June 2025
Document Type: Frontend Development Handoff

Table of Contents

Project Overview

The Swarm Multi-Agent System is a sophisticated AI collaboration platform that enables users to interact with multiple specialized AI agents through a chat-based

interface. The system emphasizes real-time collaboration, individual agent capabilities, and seamless multi-agent coordination.

Key Features

- **6 Specialized AI Agents** with distinct roles and capabilities
- **Real-time WebSocket communication** for instant collaboration
- **Cross-agent memory sharing** via SuperMemory integration
- **Filesystem access** through MCP (Model Context Protocol)
- **OpenRouter API integration** for flexible model selection
- **Email automation** and task management capabilities
- **Performance monitoring** and system health tracking

Target Users

- **Primary:** Individual power users seeking AI assistance across multiple domains
- **Future:** Teams and organizations requiring collaborative AI workflows
- **Commercial potential:** Enterprise customers with approval workflow needs

---

Core Vision & Requirements

Primary User Experience Goals

1. Chat-Based Agent Interaction

- **Individual Chat Windows:** Each agent should have its own dedicated chat interface
- **Model Selection Dropdowns:** Users can select different AI models (via OpenRouter) for each agent
- **Persistent Conversations:** Chat history maintained across sessions
- **Real-time Responses:** Immediate feedback and typing indicators

2. Group Collaboration Features

- **@Mention System:** Users can @mention specific agents to bring them into group conversations
- **Multi-agent Coordination:** Agents can collaborate on complex tasks
- **Conversation Threading:** Clear organization of multi-participant discussions
- **Agent Handoffs:** Seamless transfer of context between agents

3. Individual Agent Capabilities

- **Specialized Roles:** Each agent has distinct capabilities and personality
- **Independent Task Execution:** Agents can work on tasks without constant supervision
- **File System Access:** Agents can read/write files through MCP integration
- **Email Management:** Agents can send, read, and organize emails
- **Task Scheduling:** Agents can manage calendars and reminders

Agent Profiles

Communication Agent

- **Role:** Communication Specialist
- **Capabilities:** Text transformation, tone matching, business writing, LinkedIn optimization
- **Use Cases:** Professional communication, content refinement, messaging clarity

Cathy (Personal Assistant)

- **Role:** Personal Assistant
- **Capabilities:** Email management, task scheduling, calendar management, document organization
- **Use Cases:** Daily task management, email automation, scheduling coordination

DataMiner

- **Role:** Data Analysis Specialist

- **Capabilities:** Data analysis, visualization, statistical modeling, report generation
- **Use Cases:** Business intelligence, data insights, executive reporting

Coder

- **Role:** Software Development Expert
- **Capabilities:** Code review, debugging, architecture design, documentation, optimization
- **Use Cases:** Software development, technical problem-solving, code quality

Creative

- **Role:** Content Creation Specialist
- **Capabilities:** Content writing, brainstorming, design concepts, storytelling, brand voice
- **Use Cases:** Marketing content, creative projects, brand development

Researcher

- **Role:** Information Gathering Expert
- **Capabilities:** Web research, fact-checking, synthesis, citation management, competitive analysis
- **Use Cases:** Market research, academic work, competitive intelligence

Feature Prioritization

High Priority (MVP)

1. **Chat-based interface** with individual agent windows
2. **Real-time messaging** with WebSocket integration
3. **Agent selection and @mention** functionality
4. **Basic conversation management**
5. **Model selection dropdowns** for each agent

Medium Priority

1. **File upload and sharing** capabilities
2. **Advanced conversation threading**
3. **Performance monitoring dashboard**
4. **System health indicators**
5. **Notification system**

Low Priority (Nice to Have)

1. **Drag-and-drop UI** for agent management
2. **Advanced visualization** of agent interactions
3. **Custom agent configuration**
4. **Approval workflow UI** (future commercial feature)

Technical Architecture

Technology Stack

Frontend

- **Framework:** React 19.1.0 with Vite 6.3.5
- **Styling:** Tailwind CSS 4.1.7 with custom components
- **UI Components:** Radix UI primitives for accessibility
- **State Management:** React hooks (useState, useEffect, useRef)
- **Real-time Communication:** Socket.IO Client 4.8.1
- **Routing:** React Router DOM 7.6.1
- **Form Handling:** React Hook Form 7.56.3 with Zod validation
- **Icons:** Lucide React 0.510.0
- **Animations:** Framer Motion 12.15.0

Backend Integration

- **API Base URL:** Configurable via environment variables
- **WebSocket Connection:** Real-time bidirectional communication
- **HTTP Requests:** RESTful API calls for data operations
- **Authentication:** JWT-based (future implementation)

Current Frontend Structure

```
frontend/
├── src/
│   ├── App.jsx                 # Main application component
│   ├── main.jsx               # Application entry point
│   ├── components/
│   │   ├── EnhancedComponents.jsx  # Custom agent and message components
│   │   └── ui/                     # Radix UI component library
│   │       ├── accordion.jsx
│   │       ├── alert-dialog.jsx
│   │       ├── button.jsx
│   │       ├── card.jsx
│   │       ├── dialog.jsx
│   │       ├── input.jsx
│   │       ├── select.jsx
│   │       ├── textarea.jsx
│   │       └── ... (30+ UI components)
│   └── styles/                 # Global styles and themes
├── index.html                  # HTML template
├── package.json               # Dependencies and scripts
├── vite.config.js            # Vite configuration
└── dist/                     # Built application (production)
```

State Management Architecture

Core Application State

```
// Agent Management
const [agents, setAgents] = useState([]);
const [selectedAgents, setSelectedAgents] = useState([]);
const [agentPerformance, setAgentPerformance] = useState({});

// Conversation Management
const [conversations, setConversations] = useState([]);
const [currentConversation, setCurrentConversation] = useState(null);
const [messages, setMessages] = useState([]);

// UI State
const [isLoading, setIsLoading] = useState(false);
const [connectionStatus, setConnectionStatus] = useState('disconnected');
const [notifications, setNotifications] = useState([]);
const [darkMode, setDarkMode] = useState(false);

// System Monitoring
const [systemHealth, setSystemHealth] = useState(null);
const [responseTime, setResponseTime] = useState(0);
```

WebSocket Integration

Connection Management

```
// Socket initialization
const socketRef = useRef(null);
socketRef.current = io(API_BASE_URL, {
  transports: ['websocket', 'polling']
});

// Event handlers
socketRef.current.on('connect', () => setConnectionStatus('connected'));
socketRef.current.on('message_response', handleMessageResponse);
socketRef.current.on('agent_status_update', handleAgentStatusUpdate);
socketRef.current.on('performance_update', handlePerformanceUpdate);
```

Real-time Events

- **message_response:** Agent responses to user messages
- **agent_status_update:** Changes in agent availability/status
- **performance_update:** Real-time performance metrics
- **system_health:** System monitoring data

API Documentation

Base Configuration

- **Base URL:** `https://your-render-deployment.onrender.com` (configurable)
- **WebSocket:** `wss://your-render-deployment.onrender.com/socket.io`
- **Content-Type:** `application/json`
- **CORS:** Enabled for all origins

Core API Endpoints

1. System Health & Status

GET `/api/health`

Purpose: Check system health and service status

```
// Response
{
  "status": "success",
  "data": {
    "system": {
      "overall_status": "healthy",
      "healthy_services": 2,
      "total_services": 2,
      "services": {
        "mcp_filesystem": "healthy",
        "websocket": "healthy"
      }
    },
    "configuration": {
      "valid": false,
      "missing_configs": ["DATABASE_URL", "OPENROUTER_API_KEY"]
    },
    "services_initialized": true,
    "version": "3.0.0"
  }
}
```

GET `/api/system/status`

Purpose: Detailed system status with performance metrics

```
// Response
{
  "status": "success",
  "data": {
    "services": {
      "mcp_filesystem": {
        "status": "healthy",
        "message": "Service is operational",
        "response_time_ms": 15.2,
        "last_check": "2025-06-23T15:30:00Z"
      }
    },
    "task_manager": {
      "active_tasks": 0,
      "completed_tasks": 15,
      "failed_tasks": 0
    },
    "orchestrator": {
      "agents_count": 6,
      "active_conversations": 2
    }
  }
}
```

2. Agent Management

GET `/api/agents`

Purpose: Retrieve all available agents with their capabilities

```
// Response
{
  "status": "success",
  "data": [
    {
      "id": "cathy",
      "name": "Cathy",
      "role": "Personal Assistant",
      "personality": "Helpful, organized, and proactive...",
      "capabilities": [
        {
          "name": "email_management",
          "description": "Send, read, and organize emails...",
          "confidence_level": 0.95,
          "execution_time_estimate": 30
        }
      ],
      "status": "idle",
      "current_model": "gpt-4",
      "collaboration_style": "coordinator",
      "performance": {
        "total_tasks": 25,
        "successful_tasks": 24,
        "success_rate": 96.0,
        "avg_response_time": 1250.5,
        "last_active": "2025-06-23T15:25:00Z"
      }
    }
  ]
}
```

GET `/api/agents/{agent_id}/config`

Purpose: Get detailed configuration for a specific agent

```
// Response
{
  "status": "success",
  "data": {
    "id": "cathy",
    "name": "Cathy",
    "role": "Personal Assistant",
    "available_models": ["gpt-4", "gpt-3.5-turbo", "claude-3"],
    "current_model": "gpt-4",
    "capabilities": [...],
    "settings": {
      "max_tokens": 4000,
      "temperature": 0.7,
      "response_format": "markdown"
    }
  }
}
```

3. Conversation Management

GET `/api/conversations`

Purpose: Retrieve user's conversation history

```
// Response
{
  "status": "success",
  "data": {
    "conversations": [
      {
        "id": "conv_123",
        "title": "Data Analysis Project",
        "created_at": "2025-06-23T10:00:00Z",
        "updated_at": "2025-06-23T15:30:00Z",
        "participants": ["cathy", "dataminer"],
        "message_count": 15,
        "status": "active"
      }
    ]
  }
}
```

POST `/api/conversations`

Purpose: Create a new conversation

```
// Request
{
  "title": "New Project Discussion",
  "agent_ids": ["cathy", "creative"],
  "initial_message": "Let's brainstorm ideas for the new campaign"
}

// Response
{
  "status": "success",
  "data": {
    "conversation": {
      "id": "conv_124",
      "title": "New Project Discussion",
      "created_at": "2025-06-23T15:35:00Z",
      "participants": ["cathy", "creative"],
      "status": "active"
    }
  }
}
```

GET `/api/conversations/{conversation_id}/messages`

Purpose: Retrieve messages from a specific conversation

```
// Response
{
  "status": "success",
  "data": {
    "messages": [
      {
        "id": "msg_001",
        "conversation_id": "conv_123",
        "content": "Can you analyze this sales data?",
        "sender_type": "user",
        "timestamp": "2025-06-23T15:30:00Z",
        "attachments": []
      },
      {
        "id": "msg_002",
        "conversation_id": "conv_123",
        "content": "I'll analyze the sales data for you...",
        "sender_type": "agent",
        "agent_name": "DataMiner",
        "agent_id": "dataminer",
        "model_used": "gpt-4",
        "timestamp": "2025-06-23T15:30:15Z",
        "processing_time_ms": 1250
      }
    ]
  }
}
```

POST `/api/conversations/{conversation_id}/messages`

Purpose: Send a message to agents in a conversation

```
// Request
{
  "content": "Please analyze this data and create a report @dataminer",
  "mentions": ["dataminer"],
  "attachments": [
    {
      "type": "file",
      "name": "sales_data.csv",
      "url": "/uploads/sales_data.csv"
    }
  ],
  "agent_ids": ["dataminer"]
}

// Response
{
  "status": "success",
  "data": {
    "message_id": "msg_003",
    "responses": [
      {
        "agent_id": "dataminer",
        "agent_name": "DataMiner",
        "content": "I've analyzed your sales data...",
        "model_used": "gpt-4",
        "processing_time_ms": 2150,
        "attachments": [
          {
            "type": "report",
            "name": "sales_analysis_report.pdf",
            "url": "/generated/sales_analysis_report.pdf"
          }
        ]
      }
    ]
  }
}
```

WebSocket Events

Client → Server Events

`send_message`

```
socket.emit('send_message', {
  conversation_id: 'conv_123',
  content: 'Hello @cathy, can you help me with scheduling?',
  mentions: ['cathy'],
  agent_ids: ['cathy']
});
```

`join_conversation`

```
socket.emit('join_conversation', {
  conversation_id: 'conv_123'
});
```

`agent_typing`

```
socket.emit('agent_typing', {
  conversation_id: 'conv_123',
  agent_id: 'cathy',
  is_typing: true
});
```

Server → Client Events

`message_response`

```
socket.on('message_response', (data) => {
  // data structure:
  {
    conversation_id: 'conv_123',
    message: {
      id: 'msg_004',
      content: 'I can help you with scheduling...',
      sender_type: 'agent',
      agent_name: 'Cathy',
      timestamp: '2025-06-23T15:35:00Z'
    }
  }
});
```

`agent_status_update`

```
socket.on('agent_status_update', (data) => {
  // data structure:
  {
    agent_id: 'cathy',
    status: 'busy', // 'idle', 'busy', 'offline'
    current_task: 'Processing email request'
  }
});
```

`typing_indicator`

```
socket.on('typing_indicator', (data) => {
  // data structure:
  {
    conversation_id: 'conv_123',
    agent_id: 'cathy',
    agent_name: 'Cathy',
    is_typing: true
  }
});
```

Error Handling

Standard Error Response Format

```
{
  "status": "error",
  "error": {
    "message": "Agent not found",
    "code": "AGENT_NOT_FOUND",
    "details": {
      "agent_id": "invalid_agent"
    },
    "timestamp": "2025-06-23T15:35:00Z"
  }
}
```

Common Error Codes

- `VALIDATION_ERROR`: Invalid request data
- `AGENT_NOT_FOUND`: Specified agent doesn't exist
- `CONVERSATION_NOT_FOUND`: Conversation ID invalid
- `SERVICE_UNAVAILABLE`: Backend service temporarily unavailable
- `RATE_LIMIT_EXCEEDED`: Too many requests
- `AUTHENTICATION_REQUIRED`: User authentication needed (future)

UI/UX Specifications

Design System

Color Palette

```
/* Primary Colors */
--primary-blue: #3B82F6;
--primary-blue-dark: #1D4ED8;
--primary-blue-light: #93C5FD;

/* Agent Status Colors */
--status-online: #10B981;    /* Green */
--status-busy: #F59E0B;      /* Amber */
--status-offline: #6B7280;   /* Gray */

/* Semantic Colors */
--success: #10B981;
--warning: #F59E0B;
--error: #EF4444;
--info: #3B82F6;

/* Neutral Palette */
--gray-50: #F9FAFB;
--gray-100: #F3F4F6;
--gray-200: #E5E7EB;
--gray-300: #D1D5DB;
--gray-500: #6B7280;
--gray-700: #374151;
--gray-900: #111827;
```

Typography

```
/* Font Family */
font-family: 'Inter', -apple-system, BlinkMacSystemFont, sans-serif;

/* Font Sizes */
--text-xs: 0.75rem;    /* 12px */
--text-sm: 0.875rem;   /* 14px */
--text-base: 1rem;     /* 16px */
--text-lg: 1.125rem;   /* 18px */
--text-xl: 1.25rem;    /* 20px */
--text-2xl: 1.5rem;    /* 24px */

/* Font Weights */
--font-normal: 400;
--font-medium: 500;
--font-semibold: 600;
--font-bold: 700;
```

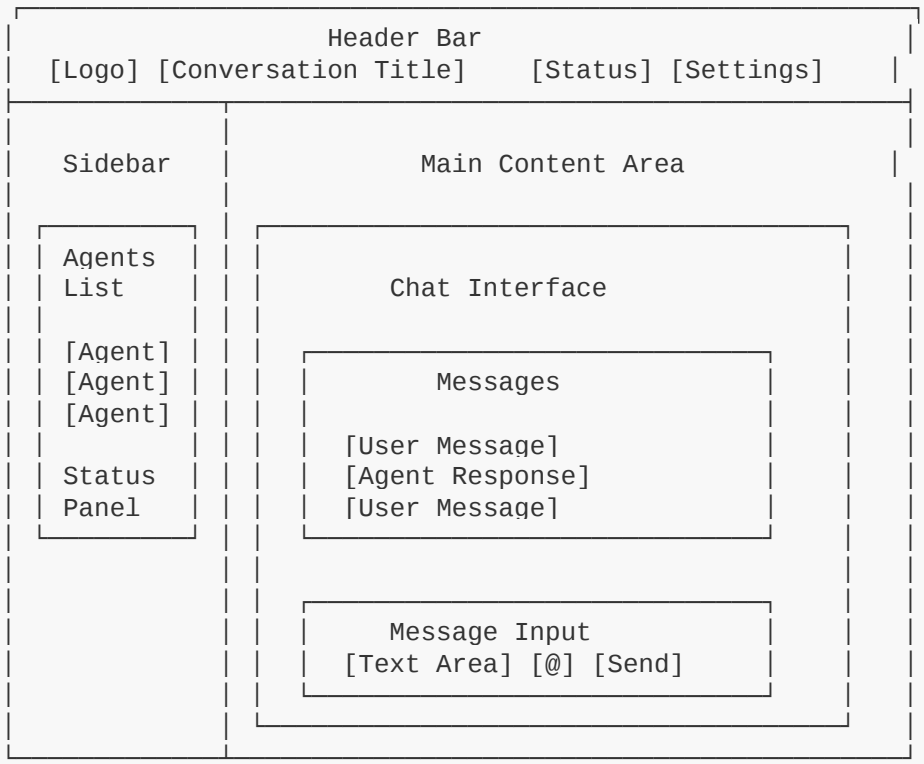
Spacing & Layout

```css
/* Spacing Scale */
--space-1: 0.25rem;   /* 4px */
--space-2: 0.5rem;    /* 8px */
--space-3: 0.75rem;   /* 12px */
--space-4: 1rem;      /* 16px */
--space-6: 1.5rem;    /* 24px */
--space-8: 2rem;      /* 32px */

/* Border Radius */
--radius-sm: 0.375rem;  /* 6px */
--radius-md: 0.5rem;    /* 8px */
--radius-lg: 0.75rem;   /* 12px */
--radius-xl: 1rem;      /* 16px */
```

Layout Structure

Main Application Layout



Component Specifications

1. Agent Card Component

```
<AgentCard
  agent={{
    id: "cathy",
    name: "Cathy",
    role: "Personal Assistant",
    status: "idle", // "idle" | "busy" | "offline"
    avatar: "/avatars/cathy.png",
    capabilities: [...],
    performance: {
      success_rate: 96.0,
      avg_response_time: 1250
    }
  }}
  isSelected={true}
  onSelect={() => {}}
  showPerformance={true}
/>
```

Visual Requirements: - **Size:** 280px width, auto height - **Status Indicator:** Colored dot (green/amber/gray) in top-right corner - **Selection State:** Blue border when selected - **Hover Effect:** Subtle shadow and scale transform - **Performance Metrics:** Small text showing success rate and response time

2. Message Component

```
<Message
  message={{
    id: "msg_001",
    content: "Can you help me with this task?",
    sender_type: "user", // "user" | "agent"
    agent_name: "Cathy",
    timestamp: "2025-06-23T15:30:00Z",
    attachments: [],
    mentions: ["cathy"]
  }}
  showTimestamp={true}
  showAvatar={true}
/>
```

Visual Requirements: - **User Messages:** Right-aligned, blue background - **Agent Messages:** Left-aligned, gray background, agent avatar - **Mentions:** Highlighted @mentions in blue - **Timestamps:** Small gray text below message - **Attachments:** File icons with download links

3. Message Input Component

```
<MessageInput
  onSendMessage={(data) => {}}
  selectedAgents={[...]}
  isLoading={false}
  placeholder="Type your message... Use @agent to mention"
  showAttachments={true}
  showMentions={true}
/>
```

Features: - **Auto-resize:** Text area grows with content - **@Mention Autocomplete:** Dropdown with agent suggestions - **File Upload:** Drag-and-drop or click to upload - **Send Button:** Disabled when empty or loading - **Typing Indicators:** Show when agents are responding

4. Sidebar Component

```
<Sidebar
  agents={[...]}
  selectedAgents={[...]}
  onAgentSelect={(agent) => {}}
  systemHealth={{...}}
  connectionStatus="connected"
  isCollapsed={false}
  onToggleCollapse={() => {}}
/>
```

Sections: 1. **Header:** Logo, title, collapse button 2. **System Status:** Connection, health metrics 3. **Agent Search:** Filter and search agents 4. **Agent List:** Scrollable list of agent cards 5. **Selected Summary:** Show selected agents count

Responsive Design

Breakpoints

```
/* Mobile First Approach */
--mobile: 320px;      /* Small phones */
--tablet: 768px;      /* Tablets */
--desktop: 1024px;    /* Desktop */
--wide: 1440px;       /* Wide screens */
```

Mobile Layout (< 768px)

- **Sidebar:** Overlay modal instead of fixed sidebar

- **Agent Cards:** Full width, stacked vertically
- **Messages:** Reduced padding, smaller avatars
- **Input:** Fixed bottom position

Tablet Layout (768px - 1024px)

- **Sidebar:** 240px width instead of 320px
- **Messages:** Optimized for touch interaction
- **Agent Cards:** 2-column grid when sidebar collapsed

Desktop Layout (> 1024px)

- **Full Layout:** As specified in main layout
- **Sidebar:** Can be collapsed to icon-only mode
- **Multiple Conversations:** Tabbed interface option

Accessibility Requirements

WCAG 2.1 AA Compliance

- **Color Contrast:** Minimum 4.5:1 ratio for normal text
- **Keyboard Navigation:** All interactive elements accessible via keyboard
- **Screen Reader Support:** Proper ARIA labels and roles
- **Focus Indicators:** Clear visual focus states

Specific Requirements

```
 // Agent status with screen reader support
<div role="status" aria-live="polite">
  <span className="sr-only">Agent status:</span>
  <span aria-label={`${agent.name} is ${agent.status}`}>
    {agent.status}
  </span>
</div>

// Message with proper semantics
<article role="article" aria-labelledby="message-author">
  <h3 id="message-author" className="sr-only">
    Message from {message.agent_name}
  </h3>
  <div>{message.content}</div>
</article>
```

Animation & Interactions

Micro-interactions

- **Message Send:** Smooth slide-in animation
- **Agent Selection:** Gentle scale and color transition
- **Typing Indicators:** Pulsing dots animation
- **Status Changes:** Color fade transitions
- **Loading States:** Skeleton screens and spinners

Performance Considerations

- **Virtual Scrolling:** For long message lists
- **Lazy Loading:** Agent avatars and attachments
- **Debounced Search:** 300ms delay for agent filtering
- **Optimistic Updates:** Immediate UI feedback

Current Frontend Status

Existing Implementation

The current frontend is a **basic React application** with the following components:

✅ Implemented Features

- **React 19 + Vite setup** with modern tooling
- **Tailwind CSS** styling framework
- **Radix UI component library** (30+ components available)
- **Socket.IO client** integration for real-time communication
- **Basic agent card** and message components
- **Responsive layout structure** with sidebar and main content
- **System health monitoring** display
- **Agent selection** and filtering functionality

⚠️ Partially Implemented

- **Message interface** - Basic structure exists but needs enhancement
- **WebSocket integration** - Connected but event handling incomplete
- **Agent performance display** - UI exists but data integration needed
- **Conversation management** - Basic state management in place

❌ Missing Critical Features

- **Individual agent chat windows** - Core requirement not implemented
- **@Mention system** - No autocomplete or mention detection
- **Model selection dropdowns** - Not connected to OpenRouter API
- **File upload/attachment** handling
- **Conversation persistence** and history
- **Real-time typing indicators**
- **Notification system**
- **Mobile responsive optimizations**

Technical Debt

1. **State Management:** Currently using basic React hooks, may need Redux/Zustand for complex state

2. **Error Handling:** Limited error boundaries and user feedback
3. **Performance:** No optimization for large message lists
4. **Testing:** No test suite implemented
5. **Accessibility:** Basic structure but needs ARIA improvements

---

Development Priorities

Phase 1: Core Chat Functionality (2-3 weeks)

Goal: Implement basic chat interface with agent interaction

Sprint 1: Individual Agent Windows

- [ ] Create tabbed interface for individual agent chats
- [ ] Implement agent-specific conversation state
- [ ] Add model selection dropdown for each agent
- [ ] Basic message sending and receiving

Sprint 2: Real-time Communication

- [ ] Complete WebSocket event handling
- [ ] Implement typing indicators
- [ ] Add message status indicators (sent, delivered, read)
- [ ] Real-time agent status updates

Sprint 3: Message Enhancement

- [ ] File upload and attachment support
- [ ] Message formatting (markdown support)
- [ ] Message search and filtering
- [ ] Conversation history persistence

Phase 2: Collaboration Features (2-3 weeks)

Goal: Enable multi-agent collaboration and @mention system

Sprint 4: @Mention System

- [ ] Implement @mention autocomplete
- [ ] Agent notification system
- [ ] Cross-conversation agent summoning
- [ ] Mention highlighting and navigation

Sprint 5: Group Conversations

- [ ] Multi-agent conversation management
- [ ] Agent handoff functionality
- [ ] Conversation threading
- [ ] Collaborative task tracking

Sprint 6: Advanced Features

- [ ] Agent performance dashboard
- [ ] System monitoring interface
- [ ] Notification center
- [ ] User preferences and settings

Phase 3: Polish & Optimization (1-2 weeks)

Goal: Production-ready application with full responsive design

Sprint 7: Mobile Optimization

- [ ] Responsive design implementation
- [ ] Touch-friendly interactions
- [ ] Mobile-specific UI patterns
- [ ] Performance optimization

Sprint 8: Accessibility & Testing

- [ ] WCAG 2.1 AA compliance
- [ ] Comprehensive test suite
- [ ] Error handling and recovery
- [ ] Documentation and deployment

Implementation Guidelines

Development Setup

Prerequisites

```
# Node.js 18+ and npm/pnpm
node --version  # v18.0.0+
npm --version   # v8.0.0+

# Git for version control
git --version
```

Local Development

```
# Clone repository
git clone https://github.com/copp1723/multi_uni_merge.git
cd multi_uni_merge/frontend

# Install dependencies
npm install

# Start development server
npm run dev

# Build for production
npm run build
```

Environment Configuration

```
# .env.local
VITE_API_URL=http://localhost:5000
VITE_WS_URL=ws://localhost:5000
VITE_ENVIRONMENT=development
```

Code Standards

File Structure

```
src/
├── components/
│   ├── agents/
│   │   ├── AgentCard.jsx
│   │   ├── AgentList.jsx
│   │   └── AgentSelector.jsx
│   ├── chat/
│   │   ├── ChatWindow.jsx
│   │   ├── MessageList.jsx
│   │   ├── MessageInput.jsx
│   │   └── TypingIndicator.jsx
│   ├── layout/
│   │   ├── Sidebar.jsx
│   │   ├── Header.jsx
│   │   └── MainContent.jsx
│   └── ui/           # Radix UI components
├── hooks/
│   ├── useWebSocket.js
│   ├── useAgents.js
│   └── useConversations.js
├── services/
│   ├── api.js
│   ├── websocket.js
│   └── storage.js
├── utils/
│   ├── formatters.js
│   ├── validators.js
│   └── constants.js
└── styles/
    ├── globals.css
    └── components.css
```

Naming Conventions

```
// Components: PascalCase
const AgentCard = () => {};

// Hooks: camelCase with 'use' prefix
const useWebSocket = () => {};

// Constants: UPPER_SNAKE_CASE
const API_BASE_URL = 'https://api.example.com';

// Functions: camelCase
const formatMessage = (message) => {};

// CSS Classes: kebab-case
.agent-card-container {}
```

Component Patterns

```
// Functional components with hooks
import { useState, useEffect } from 'react';

const AgentCard = ({ agent, onSelect, isSelected }) => {
  const [isHovered, setIsHovered] = useState(false);

  useEffect(() => {
    // Side effects
  }, [agent.id]);

  return (
    <div
      className={`agent-card ${isSelected ? 'selected' : ''}`}
      onMouseEnter={() => setIsHovered(true)}
      onMouseLeave={() => setIsHovered(false)}
      onClick={() => onSelect(agent)}
    >
      {/* Component content */}
    </div>
  );
};

export default AgentCard;
```

State Management Strategy

Local State (useState)

- Component-specific UI state
- Form inputs and validation
- Temporary display states

Context API

- User preferences and settings
- Theme and dark mode
- Authentication state (future)

Custom Hooks

- WebSocket connection management
- API data fetching
- Complex business logic

```
// Example: useAgents hook
const useAgents = () => {
  const [agents, setAgents] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  const fetchAgents = async () => {
    try {
      setLoading(true);
      const response = await api.get('/agents');
      setAgents(response.data);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    fetchAgents();
  }, []);

  return { agents, loading, error, refetch: fetchAgents };
};
```

Performance Optimization

Code Splitting

```
// Lazy load heavy components
const AgentDashboard = lazy(() => import('./components/AgentDashboard'));
const PerformanceMonitor = lazy(() =>
import('./components/PerformanceMonitor'));

// Use Suspense for loading states
<Suspense fallback={<LoadingSpinner />}>
  <AgentDashboard />
</Suspense>
```

Memoization

```
// Memoize expensive calculations
const memoizedAgentStats = useMemo(() => {
  return calculateAgentPerformance(agents, messages);
}, [agents, messages]);

// Memoize components to prevent unnecessary re-renders
const MemoizedAgentCard = memo(AgentCard);
```

Virtual Scrolling

```
// For large message lists
import { FixedSizeList as List } from 'react-window';

const MessageList = ({ messages }) => (
  <List
    height={600}
    itemCount={messages.length}
    itemSize={80}
    itemData={messages}
  >
    {MessageItem}
  </List>
);
```

Testing Requirements

Testing Strategy

Unit Tests (Jest + React Testing Library)

```
// Example: AgentCard.test.jsx
import { render, screen, fireEvent } from '@testing-library/react';
import AgentCard from '../AgentCard';

describe('AgentCard', () => {
  const mockAgent = {
    id: 'cathy',
    name: 'Cathy',
    role: 'Personal Assistant',
    status: 'idle'
  };

  test('renders agent information correctly', () => {
    render(<AgentCard agent={mockAgent} />);

    expect(screen.getByText('Cathy')).toBeInTheDocument();
    expect(screen.getByText('Personal Assistant')).toBeInTheDocument();
  });

  test('calls onSelect when clicked', () => {
    const mockOnSelect = jest.fn();
    render(<AgentCard agent={mockAgent} onSelect={mockOnSelect} />);

    fireEvent.click(screen.getByRole('button'));
    expect(mockOnSelect).toHaveBeenCalledWith(mockAgent);
  });
});
```

Integration Tests

- WebSocket connection and message flow
- API integration with backend
- Multi-component interaction scenarios

E2E Tests (Playwright/Cypress)

- Complete user workflows
- Cross-browser compatibility
- Mobile responsive testing

Test Coverage Requirements

- **Minimum:** 80% code coverage
- **Components:** 90% coverage for critical components
- **Hooks:** 100% coverage for custom hooks
- **Utils:** 100% coverage for utility functions

Deployment & Environment

Build Configuration

Production Build

```
 # Build optimized production bundle
npm run build

# Preview production build locally
npm run preview

# Analyze bundle size
npm run analyze
```

Environment Variables

```
# Production (.env.production)
VITE_API_URL=https://your-backend.onrender.com
VITE_WS_URL=wss://your-backend.onrender.com
VITE_ENVIRONMENT=production
VITE_SENTRY_DSN=your-sentry-dsn
```

Deployment Options

Option 1: Integrated with Backend (Current)

- Frontend built and served by Flask backend
- Single deployment on Render
- Simplified deployment process

Option 2: Separate Frontend Deployment

- Deploy frontend to Vercel/Netlify
- Backend remains on Render
- Better performance and CDN benefits

Option 3: Docker Deployment

```
# Dockerfile for frontend
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
RUN npm run build
EXPOSE 3000
CMD ["npm", "run", "preview"]
```

Monitoring & Analytics

Error Tracking

```
// Sentry integration
import * as Sentry from "@sentry/react";

Sentry.init({
  dsn: import.meta.env.VITE_SENTRY_DSN,
  environment: import.meta.env.VITE_ENVIRONMENT,
});
```

Performance Monitoring

- Core Web Vitals tracking
- API response time monitoring
- WebSocket connection health
- User interaction analytics

Conclusion

This handoff document provides a comprehensive foundation for developing the Swarm Multi-Agent System frontend. The project has significant potential for creating an innovative AI collaboration platform.

Key Success Factors

1. **Focus on Core Chat Experience:** Prioritize individual agent windows and real-time communication
2. **Implement @Mention System Early:** This is a differentiating feature
3. **Maintain Performance:** Optimize for real-time interactions and large message volumes
4. **Plan for Scale:** Design architecture to support future enterprise features

Next Steps

1. **Review and clarify requirements** with stakeholders

2. **Set up development environment** and tooling
3. **Begin Phase 1 implementation** with core chat functionality
4. **Establish regular review cycles** for feedback and iteration

Support & Resources

- **Backend API:** Fully functional and documented
- **Design System:** Tailwind CSS + Radix UI components available
- **WebSocket Integration:** Real-time infrastructure ready
- **Deployment Pipeline:** Render deployment configured

Contact: Available for questions and clarification during development process.

Document Version: 1.0
Last Updated: June 2025
Total Pages: 24