Intent-Driven OS: AI Agent Control Architecture Extension

Architecture Extension Specification v1.0
Generated: February 07, 2026

Executive Summary

This document extends the Intent-Driven OS MVP architecture to enable **AI-controlled application interaction** through an agentic tool-use system. The AI will not only decide which applications to open but actively control them through a standardized tool interface, enabling autonomous task execution based on natural language commands.

Key Capabilities:

- AI discovers and invokes application-specific tools dynamically
- Applications expose controllable actions through a unified Tool Registry
- Streaming agent execution with real-time UI feedback
- Tool result integration back into workspace state
- Conversation-based control with memory of previous actions

Table of Contents

---

1. Architecture Overview

1.1 Conceptual Model

```
User Intent → AI Agent → Tool Discovery → Tool Execution → App Control → UI Update
                ↑                                                          ↓
                └──────────────── Tool Results & Feedback ─────────────────┘
```

1.2 Core Components

Component	Responsibility	Location
Agent Controller	Orchestrates AI tool execution loop	Client-side React Hook
Tool Registry	Maps app actions to executable functions	Global Zustand store
Tool Executor	Invokes tools and handles results	Utility service
Gemini Function Calling	AI decides which tools to use	Google AI SDK
Feedback Stream	Real-time UI updates during execution	React state + SSE

1.3 Integration with Existing Architecture

This extension **augments** the existing workspace architecture without breaking changes:

- **Workspace Manager**: Now receives tool execution results
- **App Registry**: Extended with tool definitions per app
- **State Management**: New `agentState` domain added to Zustand
- **API Routes**: New `/api/agent-execute` endpoint for streaming

---

2. Tool Registry System

2.1 Tool Definition Standard

Each application exposes tools through a standardized schema compatible with Google Generative AI's function calling:

```
interface AppTool {
  name: string;                    // Unique identifier: "notes_create_note"
  description: string;             // Natural language description for AI
  appId: string;                   // Associated app: "notes"
  parameters: {                    // JSON Schema for parameters
    type: "object";
    properties: Record<string, {
      type: string;
      description: string;
      enum?: string[];
    }>;
    required: string[];
  };
  execute: (params: any) => Promise<ToolResult>;  // Actual implementation
}

interface ToolResult {
  success: boolean;
  data?: any;
  error?: string;
  uiUpdate?: {                     // Optional UI feedback
    type: "highlight" | "scroll" | "flash";
    targetId: string;
  };
}
```

2.2 Central Tool Registry

```
// stores/toolRegistry.ts
interface ToolRegistryState {
  tools: Map<string, AppTool>;

  // Actions
  registerTool: (tool: AppTool) => void;
  unregisterTool: (toolName: string) => void;
  getToolsForApp: (appId: string) => AppTool[];
  getAllTools: () => AppTool[];
  getToolDefinitionsForAI: () => FunctionDeclaration[];  // For Gemini
}

export const useToolRegistry = create<ToolRegistryState>((set, get) => ({
  tools: new Map(),

  registerTool: (tool) => set((state) => {
    const newTools = new Map(state.tools);
    newTools.set(tool.name, tool);
    return { tools: newTools };
  }),

  getToolDefinitionsForAI: () => {
    return Array.from(get().tools.values()).map(tool => ({
      name: tool.name,
      description: tool.description,
      parameters: tool.parameters
    }));
  }
}));
```

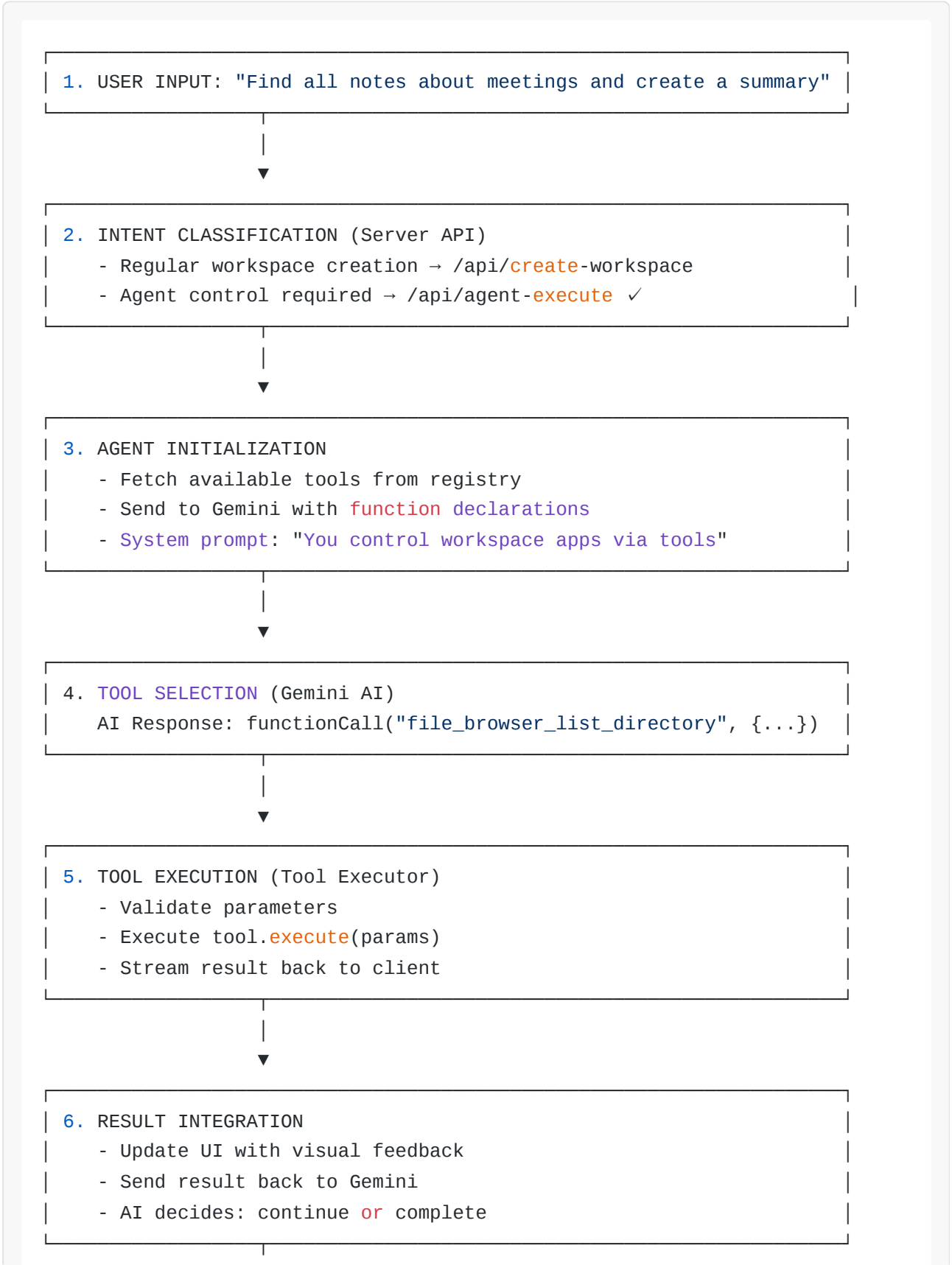
2.3 Tool Discovery Pattern

Applications **self-register** their tools when mounted:

```tsx
// apps/NotesApp.tsx
import { useToolRegistry } from '@/stores/toolRegistry';

export default function NotesApp({ id, position }: AppProps) {
  const registerTool = useToolRegistry(state => state.registerTool);
  const unregisterTool = useToolRegistry(state => state.unregisterTool);

  useEffect(() => {
    // Register tools when app mounts
    registerTool({
      name: "notes_create_note",
      description: "Create a new note with specified content and save to
file system",
      appId: "notes",
      parameters: {
        type: "object",
        properties: {
          filename: { type: "string", description: "Note filename (e.g.
'meeting.txt')" },
          content: { type: "string", description: "Note content" }
        },
        required: ["filename", "content"]
      },
      execute: async (params) => {
        const path = `/notes/${params.filename}`;
        await fileSystem.writeFile(path, params.content);
        return {
          success: true,
          data: { path },
          uiUpdate: { type: "flash", targetId: id }
        };
      }
    });

    // Cleanup on unmount
    return () => unregisterTool("notes_create_note");
  }, []);

  return (/* App UI */);
}
```

3. AI Agent Control Flow

3.1 Execution Lifecycle



```
                    |
                    ▼
┌──────────────────────────────────────────────────────────────────────┐
| 7. ITERATION (if needed)                                             |
|    - AI calls next tool based on previous results                    |
|    - Loop back to step 4 until task complete                         |
└──────────────────────────────────────────────────────────────────────┘
```

3.2 Decision Logic: Workspace Creation vs Agent Control

```
// pages/api/intent-router.ts
export default async function handler(req: NextApiRequest, res:
NextApiResponse) {
  const { intent } = req.body;

  // Use Gemini to classify intent type
  const classificationPrompt = `
Classify this user intent:
Intent: "${intent}"

Does this require:
A) Just opening apps (workspace creation)
B) Controlling apps to accomplish a task (agent mode)

Examples:
- "Open notes and timer" → A
- "Find all meeting notes and summarize them" → B
- "Create a study workspace" → A
- "Schedule all my tasks in the calendar" → B

Respond with just A or B.
  `;

  const result = await model.generateContent(classificationPrompt);
  const classification = result.response.text().trim();

  if (classification === 'B') {
    // Route to agent execution
    return res.status(200).json({
      mode: 'agent',
      redirectTo: '/api/agent-execute'
    });
  } else {
    // Regular workspace creation
    return createWorkspace(intent);
  }
}
```

---

4. Application Tool Interface

4.1 Tool Categories by Application

Application	Sample Tools	Use Cases
Notes	`create_note`, `search_notes`, `append_to_note`	Content creation, search
Todo	`add_task`, `complete_task`, `list_tasks`	Task management
File Browser	`list_directory`, `move_file`, `delete_file`	File operations
Calendar	`create_event`, `list_events`, `delete_event`	Scheduling
Timer	`start_timer`, `pause_timer`, `reset_timer`	Time tracking
Code Editor	`open_file`, `edit_line`, `run_code`	Code manipulation
Email Composer	`draft_email`, `send_email`, `save_draft`	Communication
Whiteboard	`add_shape`, `add_text`, `clear_board`	Visual creation

4.2 Example: Complete Todo App Tools

```
// apps/TodoApp/tools.ts
import { fileSystem } from '@/lib/fileSystem';

export const todoTools: AppTool[] = [
  {
    name: "todo_add_task",
    description: "Add a new task to the todo list",
    appId: "todo",
    parameters: {
      type: "object",
      properties: {
        title: {
          type: "string",
          description: "Task title"
        },
        priority: {
          type: "string",
          description: "Priority level",
          enum: ["low", "medium", "high"]
        },
        dueDate: {
          type: "string",
          description: "Due date in ISO format (optional)"
        }
      },
      required: ["title"]
    },
    execute: async (params) => {
      const tasks = await fileSystem.readFile('/todos/tasks.json');
      const taskList = JSON.parse(tasks || '[]');

      const newTask = {
        id: Date.now().toString(),
        title: params.title,
        priority: params.priority || 'medium',
        completed: false,
        dueDate: params.dueDate,
        createdAt: new Date().toISOString()
      };

      taskList.push(newTask);
      await fileSystem.writeFile('/todos/tasks.json',
JSON.stringify(taskList));
```

```
      return {
        success: true,
        data: newTask,
        uiUpdate: {
          type: "scroll",
          targetId: "todo-list-bottom"
        }
      };
    }
  },

  {
    name: "todo_complete_task",
    description: "Mark a task as completed",
    appId: "todo",
    parameters: {
      type: "object",
      properties: {
        taskId: {
          type: "string",
          description: "ID of task to complete"
        }
      },
      required: ["taskId"]
    },
    execute: async (params) => {
      const tasks = await fileSystem.readFile('/todos/tasks.json');
      const taskList = JSON.parse(tasks || '[]');

      const task = taskList.find((t: any) => t.id === params.taskId);
      if (!task) {
        return { success: false, error: "Task not found" };
      }

      task.completed = true;
      task.completedAt = new Date().toISOString();

      await fileSystem.writeFile('/todos/tasks.json',
JSON.stringify(taskList));

      return {
        success: true,
        data: task,
        uiUpdate: {
          type: "highlight",
```

```
          targetId: `task-${params.taskId}`
        }
      };
    }
  },

  {
    name: "todo_list_tasks",
    description: "Get all tasks, optionally filtered by completion status",
    appId: "todo",
    parameters: {
      type: "object",
      properties: {
        completed: {
          type: "boolean",
          description: "Filter by completion status (optional)"
        }
      },
      required: []
    },
    execute: async (params) => {
      const tasks = await fileSystem.readFile('/todos/tasks.json');
      const taskList = JSON.parse(tasks || '[]');

      const filtered = params.completed !== undefined
        ? taskList.filter((t: any) => t.completed === params.completed)
        : taskList;

      return {
        success: true,
        data: { tasks: filtered, count: filtered.length }
      };
    }
  }
];
```

4.3 Tool Registration Pattern

```
// apps/TodoApp/index.tsx
import { todoTools } from './tools';

export default function TodoApp({ id }: AppProps) {
  const registerTool = useToolRegistry(state => state.registerTool);
  const unregisterTool = useToolRegistry(state => state.unregisterTool);

  useEffect(() => {
    // Register all tools for this app
    todoTools.forEach(tool => registerTool(tool));

    // Cleanup
    return () => {
      todoTools.forEach(tool => unregisterTool(tool.name));
    };
  }, []);

  // ... rest of component
}
```

---

5. Google Generative AI Integration

5.1 Function Calling Setup

```
// lib/gemini/agentClient.ts
import { GoogleGenerativeAI } from "@google/generative-ai";

const genAI = new GoogleGenerativeAI(process.env.GEMINI_API_KEY!);

export async function createAgentSession(
  tools: AppTool[],
  systemPrompt: string
) {
  // Convert tools to Gemini function declarations
  const functionDeclarations = tools.map(tool => ({
    name: tool.name,
    description: tool.description,
    parameters: tool.parameters
  }));

  const model = genAI.getGenerativeModel({
    model: "gemini-2.0-flash-exp",
    tools: [{ functionDeclarations }],
    systemInstruction: systemPrompt
  });

  return model;
}
```

5.2 Agent Execution Loop

```
// pages/api/agent-execute.ts
import { createAgentSession } from '@/lib/gemini/agentClient';
import { getToolRegistry } from '@/lib/toolExecutor';

export default async function handler(req: NextApiRequest, res:
NextApiResponse) {
  const { intent, availableTools } = req.body;

  // Setup SSE for streaming
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  const sendEvent = (event: string, data: any) => {
    res.write(`event: ${event}\ndata: ${JSON.stringify(data)}\n\n`);
  };

  try {
    // Initialize AI model with available tools
    const model = await createAgentSession(
      availableTools,
      `You are an AI agent controlling workspace applications.

You have access to the following apps and their tools:
${availableTools.map(t => `- ${t.name}: ${t.description}`).join('\n')}

Your goal: ${intent}

Think step-by-step:
1. Determine which tools you need
2. Call tools in the right order
3. Use results from previous tools to inform next steps
4. Continue until the task is complete

Always explain what you're doing before calling a tool.`
    );

    const chat = model.startChat({
      history: []
    });

    let iteration = 0;
    const MAX_ITERATIONS = 10;
```

```
sendEvent('agent-start', { intent });

while (iteration < MAX_ITERATIONS) {
  iteration++;

  // AI generates response
  const result = await chat.sendMessage(
    iteration === 1 ? intent : "Continue with the next step"
  );

  const response = result.response;

  // Check if AI wants to call a function
  const functionCall = response.functionCalls()?.[0];

  if (!functionCall) {
    // AI is done - send final message
    sendEvent('agent-complete', {
      message: response.text(),
      iterations: iteration
    });
    break;
  }

  // AI wants to call a tool
  const { name: toolName, args } = functionCall;

  sendEvent('tool-call', {
    toolName,
    args,
    thinking: response.text() || `Calling ${toolName}...`
  });

  // Execute the tool
  const tool = availableTools.find(t => t.name === toolName);
  if (!tool) {
    sendEvent('error', { message: `Tool ${toolName} not found` });
    break;
  }

  try {
    const toolResult = await tool.execute(args);

    sendEvent('tool-result', {
      toolName,
```

```
          result: toolResult,
          uiUpdate: toolResult.uiUpdate
        });

        // Send result back to AI
        const functionResponse = {
          functionResponse: {
            name: toolName,
            response: toolResult
          }
        };

        // Continue chat with tool result
        await chat.sendMessage([functionResponse]);

      } catch (error) {
        sendEvent('tool-error', {
          toolName,
          error: error.message
        });
      }
    }

    if (iteration >= MAX_ITERATIONS) {
      sendEvent('agent-timeout', {
        message: 'Maximum iterations reached'
      });
    }

  } catch (error) {
    sendEvent('error', { message: error.message });
  } finally {
    res.end();
  }
}
```

5.3 Client-Side Agent Hook

```
// hooks/useAgentExecution.ts
import { useState, useCallback } from 'react';
import { useToolRegistry } from '@/stores/toolRegistry';

interface AgentEvent {
  type: 'agent-start' | 'tool-call' | 'tool-result' | 'agent-complete' |
'error';
  data: any;
}

export function useAgentExecution() {
  const [isExecuting, setIsExecuting] = useState(false);
  const [events, setEvents] = useState<AgentEvent[]>([]);
  const getAllTools = useToolRegistry(state => state.getAllTools);

  const executeIntent = useCallback(async (intent: string) => {
    setIsExecuting(true);
    setEvents([]);

    const availableTools = getAllTools();

    const response = await fetch('/api/agent-execute', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        intent,
        availableTools: availableTools.map(t => ({
          name: t.name,
          description: t.description,
          parameters: t.parameters,
          appId: t.appId
        }))
      })
    });

    const reader = response.body!.getReader();
    const decoder = new TextDecoder();

    while (true) {
      const { done, value } = await reader.read();
      if (done) break;

      const chunk = decoder.decode(value);
```

```
      const lines = chunk.split('\n\n');

      for (const line of lines) {
        if (line.startsWith('event: ')) {
          const eventType = line.split('event: ')[1].split('\n')[0];
          const dataLine = line.split('data: ')[1];

          if (dataLine) {
            const eventData = JSON.parse(dataLine);

            setEvents(prev => [...prev, {
              type: eventType as any,
              data: eventData
            }]);

            // Handle UI updates
            if (eventData.uiUpdate) {
              handleUIUpdate(eventData.uiUpdate);
            }
          }
        }
      }
    }

    setIsExecuting(false);
  }, [getAllTools]);

  return {
    isExecuting,
    events,
    executeIntent
  };
}

function handleUIUpdate(update: any) {
  const element = document.getElementById(update.targetId);
  if (!element) return;

  switch (update.type) {
    case 'highlight':
      element.classList.add('agent-highlight');
      setTimeout(() => element.classList.remove('agent-highlight'), 2000);
      break;
    case 'scroll':
      element.scrollIntoView({ behavior: 'smooth' });
      break;
```

```
    case 'flash':
      element.classList.add('agent-flash');
      setTimeout(() => element.classList.remove('agent-flash'), 500);
      break;
  }
}
```

---

6. State Management Extension

6.1 New Agent State Domain

```
// stores/agentStore.ts
interface AgentState {
  // Current execution
  isExecuting: boolean;
  currentIntent: string | null;
  executionHistory: AgentEvent[];

  // Tool management
  lastToolCall: {
    toolName: string;
    args: any;
    result: any;
    timestamp: number;
  } | null;

  // UI state
  agentPanelOpen: boolean;
  streamingThinking: string;

  // Actions
  startExecution: (intent: string) => void;
  addEvent: (event: AgentEvent) => void;
  completeExecution: () => void;
  toggleAgentPanel: () => void;
  setThinking: (thinking: string) => void;
}

export const useAgentStore = create<AgentState>((set) => ({
  isExecuting: false,
  currentIntent: null,
  executionHistory: [],
  lastToolCall: null,
  agentPanelOpen: false,
  streamingThinking: '',

  startExecution: (intent) => set({
    isExecuting: true,
    currentIntent: intent,
    executionHistory: [],
    agentPanelOpen: true
```

```
  }),

  addEvent: (event) => set((state) => ({
    executionHistory: [...state.executionHistory, event],
    lastToolCall: event.type === 'tool-result' ? {
      toolName: event.data.toolName,
      args: event.data.args,
      result: event.data.result,
      timestamp: Date.now()
    } : state.lastToolCall
  })),

  completeExecution: () => set({
    isExecuting: false,
    currentIntent: null,
    streamingThinking: ''
  }),

  toggleAgentPanel: () => set((state) => ({
    agentPanelOpen: !state.agentPanelOpen
  })),

  setThinking: (thinking) => set({ streamingThinking: thinking })
}));
```

6.2 Integration with Workspace State

```typescript
// stores/workspaceStore.ts (extended)
interface WorkspaceState {
  // ... existing fields

  // New agent-related fields
  agentControlEnabled: boolean;
  lastAgentAction: {
    intent: string;
    toolsUsed: string[];
    timestamp: number;
  } | null;

  // New actions
  enableAgentControl: () => void;
  disableAgentControl: () => void;
  recordAgentAction: (intent: string, tools: string[]) => void;
}
```

---

7. Real-time Feedback System

7.1 Agent Activity Panel Component

```
// components/AgentPanel.tsx
import { useAgentStore } from '@/stores/agentStore';

export default function AgentPanel() {
  const {
    isExecuting,
    currentIntent,
    executionHistory,
    streamingThinking,
    agentPanelOpen,
    toggleAgentPanel
  } = useAgentStore();

  if (!agentPanelOpen) {
    return (
      <button
        onClick={toggleAgentPanel}
        className="fixed bottom-4 right-4 bg-blue-600 text-white px-4 py-2
rounded-full shadow-lg"
      >
        {isExecuting ? '🤖 Agent Working...' : '🤖 Agent'}
      </button>
    );
  }

  return (
    <div className="fixed bottom-4 right-4 w-96 bg-white rounded-lg shadow-
2xl border border-gray-200 max-h-[600px] flex flex-col">
      <div className="p-4 border-b border-gray-200 flex justify-between
items-center">
        <h3 className="font-semibold text-gray-900">AI Agent</h3>
        <button onClick={toggleAgentPanel} className="text-gray-500
hover:text-gray-700">
          ×
        </button>
      </div>

      <div className="flex-1 overflow-y-auto p-4 space-y-3">
        {currentIntent && (
          <div className="bg-blue-50 border border-blue-200 rounded p-3">
```

```
            <div className="text-xs text-blue-600 font-medium mb-
1">Intent</div>
            <div className="text-sm text-gray-900">{currentIntent}</div>
          </div>
        )}

        {executionHistory.map((event, idx) => (
          <AgentEventCard key={idx} event={event} />
        ))}

        {isExecuting && streamingThinking && (
          <div className="bg-gray-50 border border-gray-200 rounded p-3
animate-pulse">
            <div className="text-xs text-gray-500 mb-1">Thinking...</div>
            <div className="text-sm text-gray-700">{streamingThinking}</div>
          </div>
        )}
      </div>
    </div>
  );
}

function AgentEventCard({ event }: { event: AgentEvent }) {
  if (event.type === 'tool-call') {
    return (
      <div className="bg-purple-50 border border-purple-200 rounded p-3">
        <div className="text-xs text-purple-600 font-medium mb-1">
          🔧 Calling Tool
        </div>
        <div className="text-sm font-mono text-gray-900">
          {event.data.toolName}
        </div>
        {event.data.thinking && (
          <div className="text-xs text-gray-600 mt-1">
            {event.data.thinking}
          </div>
        )}
      </div>
    );
  }

  if (event.type === 'tool-result') {
    const success = event.data.result.success;
    return (
      <div className={`border rounded p-3 ${success ? 'bg-green-50 border-
green-200' : 'bg-red-50 border-red-200'}`}>
```

```
        <div className={`text-xs font-medium mb-1 ${success ? 'text-green-
600' : 'text-red-600'}`}>
          {success ? '✓ Success' : '✗ Failed'}
        </div>
        <div className="text-sm text-gray-900">
          {success ? 'Tool executed successfully' : event.data.result.error}
        </div>
      </div>
    );
  }

  if (event.type === 'agent-complete') {
    return (
      <div className="bg-blue-50 border border-blue-200 rounded p-3">
        <div className="text-xs text-blue-600 font-medium mb-1">
          ✓ Task Complete
        </div>
        <div className="text-sm text-gray-900">
          {event.data.message}
        </div>
      </div>
    );
  }

  return null;
}
```

7.2 Visual Feedback Animations

```css
/* styles/agent-feedback.css */

@keyframes agent-highlight {
  0%, 100% {
    background-color: transparent;
  }
  50% {
    background-color: rgba(59, 130, 246, 0.2);
    box-shadow: 0 0 0 3px rgba(59, 130, 246, 0.3);
  }
}

.agent-highlight {
  animation: agent-highlight 2s ease-in-out;
}

@keyframes agent-flash {
  0%, 100% { opacity: 1; }
  50% { opacity: 0.5; }
}

.agent-flash {
  animation: agent-flash 0.5s ease-in-out;
}

.agent-controlled-app {
  position: relative;
}

.agent-controlled-app::after {
  content: '🤖';
  position: absolute;
  top: 8px;
  right: 8px;
  font-size: 12px;
  background: rgba(59, 130, 246, 0.9);
  padding: 2px 6px;
  border-radius: 4px;
  animation: pulse 2s infinite;
}

@keyframes pulse {
  0%, 100% { opacity: 1; }
```

```
  50% { opacity: 0.6; }
}
```

---

8. Implementation Guide

8.1 Phase 1: Foundation (Week 1)

Tasks:

1. ✅ Create Tool Registry Zustand store
2. ✅ Create Agent State Zustand store
3. ✅ Build `/api/agent-execute` endpoint with SSE
4. ✅ Implement `useAgentExecution` hook
5. ✅ Create basic AgentPanel component

Success Criteria:

- Single tool can be called from AI
- Results stream back to client
- UI shows execution progress

8.2 Phase 2: Tool Implementation (Week 2)

Tasks:

1. ✅ Implement tools for Notes app (3 tools)
2. ✅ Implement tools for Todo app (5 tools)
3. ✅ Implement tools for File Browser (4 tools)
4. ✅ Implement tools for Calendar (3 tools)
5. ✅ Add tool registration to all apps

Success Criteria:

- Each app exposes minimum 3 tools

- Tools execute successfully
- Results update app state

8.3 Phase 3: Agent Intelligence (Week 3)

Tasks:

1. ✅ Improve Gemini system prompts for multi-step tasks
2. ✅ Add tool result validation
3. ✅ Implement error recovery (retry logic)
4. ✅ Add conversation memory (use chat history)
5. ✅ Optimize tool descriptions for better AI understanding

Success Criteria:

- Agent completes 3+ step tasks
- Handles errors gracefully
- Uses previous results in next steps

8.4 Phase 4: UX Polish (Week 4)

Tasks:

1. ✅ Add visual feedback animations
2. ✅ Improve AgentPanel UI
3. ✅ Add keyboard shortcuts (Cmd+K to invoke agent)
4. ✅ Implement agent thinking/reasoning display
5. ✅ Add execution history persistence

Success Criteria:

- Smooth animations
- Clear user feedback
- Intuitive controls

---

9. Example Use Cases

Use Case 1: Multi-App Task Orchestration

User Intent: *"Find all notes from last week about the project and create a summary, then add it as a task due Friday"*

Agent Execution:

```
1. tool_call: file_browser_list_directory({ path: "/notes" })
   → Returns: ["meeting1.txt", "standup.txt", "brainstorm.txt"]

2. tool_call: notes_search_notes({
     query: "project",
     after: "2026-01-30"
   })
   → Returns: 3 matching notes with content

3. [AI synthesizes summary from results]

4. tool_call: notes_create_note({
     filename: "project-summary.txt",
     content: "<AI-generated summary>"
   })
   → Success

5. tool_call: todo_add_task({
     title: "Review project summary",
     dueDate: "2026-02-14",
     priority: "high"
   })
   → Success

6. Agent completes: "Summary created and task added!"
```

Use Case 2: Workspace Setup with Data

User Intent: *"Set up a study session with my CS notes open, timer for 25 minutes, and todo list showing only incomplete tasks"*

Agent Execution:

```
1. [Workspace creation happens first - standard flow]
   → Notes app, Timer app, Todo app all open

2. tool_call: notes_open_note({ filename: "cs-fundamentals.txt" })
   → Opens specific note

3. tool_call: timer_start({ duration: 1500 })
   → Starts 25-minute timer

4. tool_call: todo_list_tasks({ completed: false })
   → Filters todo list to show only incomplete

5. Agent completes: "Study workspace ready!"
```

Use Case 3: Continuous Monitoring

User Intent: *"Keep track of my pomodoro sessions and create a note with my progress every hour"*

Agent Execution:

```
[This would require a scheduled task system - future enhancement]

Every hour:
1. tool_call: timer_get_statistics()
   → Returns completed pomodoros

2. tool_call: notes_append_to_note({
     filename: "daily-log.txt",
     content: "Hour X: Completed Y pomodoros"
   })
   → Appends to daily log

3. Agent continues monitoring...
```

---

10. Advanced Features (Future Extensions)

10.1 Tool Chaining

Allow tools to explicitly declare dependencies:

```
interface AppTool {
  // ... existing fields
  dependencies?: string[];  // Tools that should run first
  chainable?: boolean;      // Can be auto-chained by AI
}
```

10.2 Parallel Tool Execution

For independent operations:

```
// Execute multiple tools concurrently
const results = await Promise.all([
  executeTool('notes_search_notes', { query: 'project' }),
  executeTool('todo_list_tasks', {}),
  executeTool('calendar_list_events', { date: 'today' })
]);
```

10.3 Scheduled Agent Tasks

```
interface ScheduledTask {
  id: string;
  intent: string;
  schedule: CronExpression;
  enabled: boolean;
}

// Store in workspaceState
scheduledTasks: ScheduledTask[];
```

10.4 Agent Learning

Track successful tool sequences:

```
interface ToolSequencePattern {
  intent: string;
  tools: string[];
  successRate: number;
  timesExecuted: number;
}

// Suggest patterns when similar intents detected
```

---

11. Testing Strategy

11.1 Tool Testing

```
// __tests__/tools/notes.test.ts
import { todoTools } from '@/apps/TodoApp/tools';

describe('Todo Tools', () => {
  test('todo_add_task creates task', async () => {
    const addTool = todoTools.find(t => t.name === 'todo_add_task');
    const result = await addTool.execute({
      title: 'Test task',
      priority: 'high'
    });

    expect(result.success).toBe(true);
    expect(result.data.title).toBe('Test task');
  });

  test('todo_list_tasks returns all tasks', async () => {
    const listTool = todoTools.find(t => t.name === 'todo_list_tasks');
    const result = await listTool.execute({});

    expect(result.success).toBe(true);
    expect(Array.isArray(result.data.tasks)).toBe(true);
  });
});
```

11.2 Agent Execution Testing

```
// __tests__/agent/execution.test.ts
import { createAgentSession } from '@/lib/gemini/agentClient';

describe('Agent Execution', () => {
  test('completes multi-step task', async () => {
    const mockTools = [/* mock tool definitions */];
    const model = await createAgentSession(mockTools, 'Test prompt');

    // Simulate execution
    const result = await executeWithModel(model, 'Create a note and add a
task');

    expect(result.toolCallsMade).toContain('notes_create_note');
    expect(result.toolCallsMade).toContain('todo_add_task');
    expect(result.completed).toBe(true);
  });
});
```

---

12. Security Considerations

12.1 Tool Authorization

```
interface AppTool {
  // ... existing fields
  requiresAuth?: boolean;
  permissions?: string[];  // e.g., ['file:write', 'calendar:edit']
}

// Check before execution
function canExecuteTool(tool: AppTool, userPermissions: string[]): boolean {
  if (!tool.permissions) return true;
  return tool.permissions.every(p => userPermissions.includes(p));
}
```

12.2 Rate Limiting

```
// Prevent infinite loops
const MAX_TOOL_CALLS_PER_SESSION = 20;
const MAX_SAME_TOOL_REPEATS = 3;

// Track in agent state
toolCallCounts: Record<string, number>;
```

12.3 Sensitive Data Handling

```
// Redact sensitive data in tool results sent to AI
function sanitizeToolResult(result: any): any {
  // Remove API keys, passwords, etc.
  return {
    ...result,
    data: redactSensitiveFields(result.data)
  };
}
```

---

13. Performance Optimization

13.1 Tool Result Caching

```
const toolResultCache = new Map<string, {
  result: any;
  timestamp: number;
  ttl: number;
}>();

function getCachedResult(toolName: string, params: any): any | null {
  const key = `${toolName}:${JSON.stringify(params)}`;
  const cached = toolResultCache.get(key);

  if (cached && Date.now() - cached.timestamp < cached.ttl) {
    return cached.result;
  }

  return null;
}
```

13.2 Lazy Tool Registration

```
// Only register tools when app is actually mounted
// Unregister when unmounted
// This keeps the tool registry lean
```

13.3 Streaming Optimization

```
// Batch UI updates to reduce re-renders
const uiUpdateQueue: UIUpdate[] = [];

function flushUIUpdates() {
  const updates = [...uiUpdateQueue];
  uiUpdateQueue.length = 0;

  // Apply all updates in single batch
  updates.forEach(update => handleUIUpdate(update));
}

// Flush every 100ms during execution
setInterval(flushUIUpdates, 100);
```

---

14. Conclusion

This architecture extension enables your Intent-Driven OS to evolve from a workspace creation tool into a **true AI agent system** that actively controls and orchestrates applications.

Key Achievements:

- ✅ Standardized tool interface across all apps
- ✅ Dynamic tool discovery by AI
- ✅ Multi-step agentic task execution
- ✅ Real-time feedback and streaming
- ✅ Seamless integration with existing architecture

Next Steps:

1. Start with Phase 1 implementation
2. Build tools for 2-3 core apps first
3. Test end-to-end agent execution
4. Gradually add tools for remaining apps

5. Polish UX and add advanced features

The system maintains your core architectural principles (separation of concerns, unidirectional data flow) while adding powerful AI-driven automation capabilities.