🎯 AGENT HQ - TECHNICAL IMPLEMENTATION PROMPT

CORE CONCEPT

Build a platform where users pay $1, get a 24h chat link, talk to "ONE AI AGENT", click EXECUTE, and watch live browser automation happen. Behind the scenes: multiple specialist agents coordinated via MCP, but user never knows.

Complete User Flow:

> Landing page interaction → $1 payment → Real agent interface → Chat with Agent PHOENIX-7742 → Say task → Click EXECUTE → Watch browser automation live

Technical Reality:

> Landing page hooks user → Payment → Chat → MCP routes to specialist agents → Browser automation streams live → User sees magic

LANDING PAGE INTEGRATION

Reference File: `ai_control_center_white (8).html`

The provided landing page serves as the marketing funnel entry point with these key features:

- **Interactive terminal commands** - Users can type commands and see responses before paying
- **Brain activation animation** - Clicking the AI brain shows immediate visual feedback
- **$1/24H pricing prominently displayed** - Clear impulse purchase positioning
- **"Perfect for deadlines" messaging** - Creates urgency psychology
- **Contact section with multiple channels** - Builds trust and legitimacy

Landing Page Flow:

> 1. User arrives at landing page (white, professional theme)
> 2. Interacts with terminal commands (preview of capabilities)
> 3. Clicks brain or main action button
> 4. Redirected to Stripe $1 payment
> 5. Payment success → Generate 24h mission link
> 6. Redirect to actual agent interface (dark terminal theme)

Design Philosophy:

- **Landing page (white theme)**: Accessible, professional, builds trust

- **Agent interface (dark terminal)**: Powerful, hacker-style, actual product

---

🤖 FREE AI AGENT FRAMEWORKS

Browser Automation Agents

1. Browser-Use ⭐ PRIMARY CHOICE

- **GitHub**: https://github.com/browser-use/browser-use
- **Install**: `pip install browser-use`
- **Features**: MCP integration, parallel tasks, cloud-ready

```python
import asyncio
from browser_use import Agent, ChatOpenAI

async def main():
    agent = Agent(
        task="Execute user mission",
        llm=ChatOpenAI(model="gpt-4.1-mini"),  # Replace with Ollama
    )
    await agent.run()
```

2. Nanobrowser

- **GitHub**: https://github.com/nanobrowser/nanobrowser
- **Type**: Chrome Extension
- **Features**: Multi-agent system, local execution, Ollama support

3. Steel Browser

- **GitHub**: https://github.com/steel-dev/steel-browser
- **Install**: `docker run -p 3000:3000 steeldev/steel-browser`
- **Features**: Puppeteer/CDP control, session management

4. Browserable

- **GitHub**: https://github.com/browserable/browserable
- **Features**: Self-hostable, MongoDB integration

```javascript
```

```
import { Browserable } from 'browserable-js';
const browserable = new Browserable({ apiKey: 'your-api-key' });
```

5. TheAgenticBrowser

- **GitHub**: https://github.com/TheAgenticAI/TheAgenticBrowser
- **Features**: PydanticAI-based, API interface

```
bash

POST http://127.0.0.1:8000/execute_task
{"command": "Find price of RTX 3060ti on amazon"}
```

6. BrowserOS

- **GitHub**: https://github.com/browseros-ai/BrowserOS
- **Type**: Chromium fork with native AI agents
- **Features**: Privacy-first, local execution

Multi-Agent Frameworks

7. AutoGPT

- **GitHub**: https://github.com/Significant-Gravitas/AutoGPT
- **Setup**: `git clone && ./setup.sh`
- **Features**: Agent marketplace, Forge framework

8. ChatDev

- **GitHub**: https://github.com/OpenBMB/ChatDev
- **Features**: Multi-agent collaboration, role-playing

```
bash

git clone https://github.com/OpenBMB/ChatDev.git
pip install -r requirements.txt
```

9. AgentGPT

- **GitHub**: https://github.com/reworkd/AgentGPT
- **Live**: https://agentgpt.reworkd.ai/
- **Features**: Browser-based deployment

10. Kortix (Suna)

- **GitHub**: https://github.com/kortix-ai/suna
- **Features**: Docker execution, file management

CN FREE CHINESE AI MODELS

Local Models (Zero Cost)

Ollama Setup:

```bash
# Install Ollama
curl -fsSL https://ollama.com/install.sh | sh

# Chinese Models (ALL FREE)
ollama pull qwen2.5:7b          # Alibaba - general purpose
ollama pull qwen2.5:14b         # Larger version
ollama pull qwen2.5-coder:7b    # Coding specialist
ollama pull deepseek-coder:6.7b # DeepSeek coder
ollama pull yi:6b               # 01.AI model
ollama pull chatglm3:6b         # Zhipu AI
ollama pull baichuan2:7b        # Baichuan domain specialist

# API usage
ollama run qwen2.5:7b
```

LocalAI Setup:

```bash
docker run -p 8080:8080 localai/localai:latest-aio-cpu
local-ai run qwen2.5:7b
```

- **GitHub**: https://github.com/mudler/LocalAI

OpenWebUI Interface:

```bash
docker run -d -p 3000:8080 \
  --add-host=host.docker.internal:host-gateway \
  -v open-webui:/app/backend/data \
  --name open-webui --restart always \
  ghcr.io/open-webui/open-webui:main
```

- **GitHub**: https://github.com/open-webui/open-webui

Cloud APIs (Free Tiers)

DeepSeek API 💰 CHEAPEST

- **Docs**: https://api-docs.deepseek.com/quick_start/pricing
- **Pricing**: $0.07/$0.14 per 1M tokens
- **Models**: DeepSeek-V3, DeepSeek-R1, DeepSeek-Coder

Groq Free Tier

- **Limit**: 30 requests/minute free
- **Models**: Llama 3.1, Mixtral

Together AI

- **Site**: https://www.together.ai/pricing
- **Free**: $5 credit monthly

📐 TECHNICAL ARCHITECTURE

Unified Chat Interface

```javascript
```

```jsx
// UnifiedAgentInterface.jsx
import React, { useState, useRef } from 'react';

const UnifiedAgentInterface = ({ agentId, timeRemaining }) => {
  const [messages, setMessages] = useState([]);
  const [isExecuting, setIsExecuting] = useState(false);
  const [browserView, setBrowserView] = useState(null);

  const sendMessage = async (message) => {
    setMessages(prev => [...prev, { role: 'user', content: message }]);

    const response = await fetch('/api/unified-agent/chat', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        agentId,
        message,
        conversationHistory: messages
      })
    });

    const data = await response.json();

    setMessages(prev => [...prev, {
      role: 'agent',
      content: data.response,
      hasExecutableTask: data.canExecute,
      taskDescription: data.taskDescription
    }]);
  };

  const executeTask = async (taskDescription) => {
    setIsExecuting(true);

    const browserStream = await fetch('/api/unified-agent/execute', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ agentId, task: taskDescription })
    });

    const streamUrl = await browserStream.json();
    setBrowserView(streamUrl.liveStream);
  };

  return (
    <div className="min-h-screen bg-black text-green-400 font-mono">
```

```
<div className="bg-gray-900 p-4 border-b border-green-400">
 <div className="flex justify-between items-center">
  <h1 className="text-2xl font-bold text-cyan-400">
   🤖 AI AGENT {agentId}
  </h1>
  <div className="text-red-500 font-bold">
   ⏱️ {formatTime(timeRemaining)} REMAINING
  </div>
 </div>
</div>

<div className="flex h-full">
 {/* Chat Panel */}
 <div className="w-1/2 p-4">
  <div className="bg-gray-900 rounded-lg p-4 h-96 overflow-y-auto mb-4">
   {messages.map((msg, idx) => (
    <div key={idx} className={`mb-4 ${msg.role === 'user' ? 'text-cyan-400' : 'text-green-400'}`}>
     <div className="font-bold">
      {msg.role === 'user' ? '👤 YOU' : '🤖 AGENT'}
     </div>
     <div className="ml-6">{msg.content}</div>

     {msg.hasExecutableTask && (
      <button
       onClick={() => executeTask(msg.taskDescription)}
       className="ml-6 mt-2 bg-red-600 hover:bg-red-700 px-4 py-2 rounded font-bold"
       disabled={isExecuting}
      >
       {isExecuting ? '⚡ EXECUTING...' : '🚀 EXECUTE TASK'}
      </button>
     )}
    </div>
   ))}
  </div>

  <ChatInput onSend={sendMessage} disabled={isExecuting} />
 </div>

 {/* Live Browser View */}
 <div className="w-1/2 p-4">
  <div className="bg-gray-900 rounded-lg p-4 h-full">
   <h3 className="text-xl font-bold text-yellow-400 mb-4">
    🎬 LIVE AUTOMATION FEED
   </h3>

   {browserView ? (
    <div className="relative">
```

```
            <iframe
              src={browserView}
              className="w-full h-80 border border-green-400 rounded"
              title="Live Browser Automation"
            />
            <div className="absolute top-2 right-2 bg-red-600 text-white px-2 py-1 rounded text-xs">
              🔴 LIVE
            </div>
          </div>
        ) : (
          <div className="flex items-center justify-center h-80 border-2 border-dashed border-gray-600 rounded">
            <div className="text-center text-gray-500">
              <div className="text-4xl mb-2">🎭</div>
              <div>Send a task and click EXECUTE</div>
              <div>to watch the magic happen</div>
            </div>
          </div>
        )}
      </div>
    </div>
  </div>
 );
};
```

MCP Orchestrator

javascript

```
// services/MCPOrchestrator.js
class MCPOrchestrator {
  constructor() {
    this.specialists = {
      webResearch: new WebResearchAgent(),
      socialMedia: new SocialMediaAgent(),
      ecommerce: new EcommerceAgent(),
      coding: new CodingAgent(),
      formFilling: new FormFillingAgent(),
      dataExtraction: new DataExtractionAgent()
    };

    this.modelRouting = {
      webResearch: 'qwen2.5:7b',
      socialMedia: 'qwen2.5:7b',
      ecommerce: 'deepseek-coder:6.7b',
      coding: 'deepseek-coder:6.7b',
      formFilling: 'yi:6b',
      dataExtraction: 'qwen2.5:7b'
    };
  }

  async processUnifiedRequest(agentId, userMessage, conversationHistory) {
    const taskAnalysis = await this.analyzeUserIntent(userMessage);
    const requiredSpecialists = this.determineRequiredSpecialists(taskAnalysis);
    const unifiedResponse = await this.generateUnifiedResponse(taskAnalysis, requiredSpecialists);

    return {
      response: unifiedResponse.message,
      canExecute: unifiedResponse.isExecutable,
      taskDescription: taskAnalysis.task,
      internalPlan: requiredSpecialists
    };
  }

  async executeUnifiedTask(agentId, taskDescription) {
    const executionPlan = await this.createExecutionPlan(taskDescription);
    const browserSession = await this.startLiveBrowserSession(agentId);
    const execution = await this.coordinateSpecialists(executionPlan, browserSession);

    return {
      liveStream: browserSession.streamUrl,
      executionId: execution.id,
      activeSpecialists: execution.specialists
    };
  }
```

```
async analyzeUserIntent(message) {
  const response = await this.callOllamaModel('qwen2.5:7b', `
    Analyze this user request and determine what needs to be done:
    "${message}"

    Respond with JSON:
    {
      "task": "brief description",
      "category": "web_research|social_media|ecommerce|coding|form_filling|data_extraction",
      "complexity": "simple|medium|complex",
      "requiresBrowser": true/false,
      "estimatedTime": "1-5 minutes"
    }
  `);

  return JSON.parse(response);
}

determineRequiredSpecialists(taskAnalysis) {
  const specialists = [];

  switch(taskAnalysis.category) {
    case 'ecommerce':
      specialists.push('webResearch', 'ecommerce', 'formFilling');
      break;
    case 'social_media':
      specialists.push('socialMedia', 'webResearch');
      break;
    case 'coding':
      specialists.push('coding', 'webResearch');
      break;
    case 'data_extraction':
      specialists.push('webResearch', 'dataExtraction');
      break;
    default:
      specialists.push('webResearch');
  }

  return specialists;
}

async generateUnifiedResponse(taskAnalysis, specialists) {
  const response = await this.callOllamaModel('qwen2.5:7b', `
    You are a unified AI agent. The user wants: "${taskAnalysis.task}"

    Respond as ONE agent who can do everything. Be confident and engaging.
```

```
      End with "Click EXECUTE to watch me work!" if it's an executable task.

      Keep it conversational and exciting.
    `);

    return {
      message: response,
      isExecutable: taskAnalysis.requiresBrowser
    };
  }

  async coordinateSpecialists(executionPlan, browserSession) {
    const results = [];

    for (const step of executionPlan.steps) {
      const specialist = this.specialists[step.specialist];
      const model = this.modelRouting[step.specialist];

      const result = await specialist.execute(step.action, model, browserSession);
      results.push(result);

      await this.sleep(2000); // Let user see automation
    }

    return {
      id: this.generateId(),
      specialists: executionPlan.steps.map(s => s.specialist),
      results
    };
  }

  async callOllamaModel(model, prompt) {
    const response = await fetch('http://localhost:11434/api/generate', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        model,
        prompt,
        stream: false
      })
    });

    const data = await response.json();
    return data.response;
  }
}
```

Specialist Agents

javascript

```
// agents/WebResearchAgent.js
class WebResearchAgent {
  async execute(action, model, browserSession) {
    const browser = browserSession.browser;
    const page = await browser.newPage();

    const instructions = await this.getInstructions(action, model);

    // Execute with live browser (USER SEES THIS)
    await page.goto('https://www.google.com');
    await page.type('input[name="q"]', instructions.searchQuery);
    await page.press('input[name="q"]', 'Enter');

    const results = await page.$$eval('.g', elements =>
      elements.slice(0, 5).map(el => ({
        title: el.querySelector('h3')?.textContent,
        url: el.querySelector('a')?.href,
        snippet: el.querySelector('.VwiC3b')?.textContent
      }))
    );

    await page.close();

    return {
      specialist: 'webResearch',
      action,
      results,
      cost: 0.0000
    };
  }

  async getInstructions(action, model) {
    const response = await fetch('http://localhost:11434/api/generate', {
      method: 'POST',
      body: JSON.stringify({
        model,
        prompt: `Convert this action to search query: ${action}`,
        stream: false
      })
    });

    return JSON.parse((await response.json()).response);
  }
}

// agents/EcommerceAgent.js
```

```
class EcommerceAgent {
  async execute(action, model, browserSession) {
    const browser = browserSession.browser;
    const page = await browser.newPage();

    // Get shopping instructions from model
    const instructions = await this.getShoppingInstructions(action, model);

    // Execute e-commerce automation
    await page.goto('https://amazon.com');
    await page.type('#twotabsearchtextbox', instructions.searchTerm);
    await page.press('#twotabsearchtextbox', 'Enter');

    // Extract product info
    const products = await page.$$eval('[data-component-type="s-search-result"]', items =>
      items.slice(0, 3).map(item => ({
        title: item.querySelector('h2 span')?.textContent,
        price: item.querySelector('.a-price-whole')?.textContent,
        rating: item.querySelector('.a-icon-alt')?.textContent
      }))
    );

    await page.close();

    return {
      specialist: 'ecommerce',
      action,
      products,
      cost: 0.0000
    };
  }
}

// agents/SocialMediaAgent.js
class SocialMediaAgent {
  async execute(action, model, browserSession) {
    const browser = browserSession.browser;
    const page = await browser.newPage();

    const content = await this.generateContent(action, model);

    // Social media automation (example: LinkedIn)
    await page.goto('https://linkedin.com/login');
    // Handle authentication flow
    // Create and post content

    return {
```

```
      specialist: 'socialMedia',
      action,
      contentCreated: content,
      cost: 0.0000
    };
  }
}
```

Live Browser Streaming

javascript

```
// services/LiveBrowserStream.js
class LiveBrowserStream {
  async startSession(agentId) {
    const browser = await chromium.launch({
      headless: false, // USER MUST SEE BROWSER
      args: ['--no-sandbox'],
      recordVideo: { dir: `recordings/${agentId}/` }
    });

    const recordingStream = await this.setupScreenRecording(browser);

    return {
      browser,
      streamUrl: `/live-browser/${agentId}`,
      recordingStream
    };
  }

  async setupScreenRecording(browser) {
    // Implementation for streaming browser to user
    // Use WebRTC or Socket.io for real-time streaming
  }
}
```

Payment Integration

javascript

```
// routes/payment.js
const express = require('express');
const stripe = require('stripe')(process.env.STRIPE_SECRET_KEY);
const MCPOrchestrator = require('../services/MCPOrchestrator');

const router = express.Router();
const orchestrator = new MCPOrchestrator();

router.post('/create-payment-intent', async (req, res) => {
  const { agentType } = req.body;

  const paymentIntent = await stripe.paymentIntents.create({
    amount: 100, // $1.00
    currency: 'usd',
    metadata: { agentType, service: 'disposable-agent' }
  });

  res.json({ clientSecret: paymentIntent.client_secret });
});

router.post('/deploy-agent', async (req, res) => {
  const { paymentIntentId, agentType, userId } = req.body;

  const paymentIntent = await stripe.paymentIntents.retrieve(paymentIntentId);

  if (paymentIntent.status !== 'succeeded') {
    return res.status(400).json({ error: 'Payment not completed' });
  }

  const deployment = await orchestrator.deployAgent(userId, agentType);

  res.json({
    success: true,
    agentId: deployment.agentId,
    missionToken: deployment.missionToken,
    missionUrl: `/mission/${deployment.agentId}?token=${deployment.missionToken}`
  });
});
```

🗄️ DATABASE SCHEMA

sql

```sql
-- Agent Sessions
CREATE TABLE agent_sessions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  agent_code VARCHAR(20) UNIQUE NOT NULL,
  user_id UUID,
  expires_at TIMESTAMP NOT NULL,
  status VARCHAR(20) DEFAULT 'ACTIVE',
  total_cost DECIMAL(10,4) DEFAULT 0.0000,
  tasks_completed INTEGER DEFAULT 0,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Conversations
CREATE TABLE conversations (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  agent_id UUID REFERENCES agent_sessions(id),
  role VARCHAR(10) NOT NULL,
  message TEXT NOT NULL,
  has_executable_task BOOLEAN DEFAULT FALSE,
  timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Task Executions
CREATE TABLE task_executions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  agent_id UUID REFERENCES agent_sessions(id),
  task_description TEXT NOT NULL,
  specialists_used JSONB,
  execution_plan JSONB,
  browser_recording_url TEXT,
  status VARCHAR(20) DEFAULT 'PENDING',
  cost DECIMAL(10,4) DEFAULT 0.0000,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Indexes
CREATE INDEX idx_agent_sessions_expires_at ON agent_sessions(expires_at);
CREATE INDEX idx_conversations_agent_id ON conversations(agent_id);
CREATE INDEX idx_task_executions_agent_id ON task_executions(agent_id);
```

⚙️ SETUP INSTRUCTIONS

Install Dependencies

```bash
# Backend
npm install express sqlite3 better-sqlite3 bullmq stripe
npm install playwright puppeteer cors helmet compression

# Frontend
npm install react terminal-in-react tailwindcss
npm install socket.io-client

# AI Models
curl -fsSL https://ollama.com/install.sh | sh
ollama pull qwen2.5:7b deepseek-coder:6.7b yi:6b
```

Environment Variables

```bash
# .env file
STRIPE_SECRET_KEY=sk_test_...
STRIPE_PUBLISHABLE_KEY=pk_test_...
JWT_SECRET=your-secret-key
OLLAMA_URL=http://localhost:11434
DATABASE_URL=sqlite:./agents.db
NODE_ENV=development
```

Start Services

```bash
# Start Ollama
ollama serve

# Start OpenWebUI (optional)
docker run -d -p 3000:8080 ghcr.io/open-webui/open-webui:ollama

# Start application
npm run dev
```

🔧 IMPLEMENTATION NOTES

Model Routing Strategy

```javascript
const modelSelection = {
  'simple_search': 'qwen2.5:7b',       // Fast, general
  'complex_research': 'qwen2.5:14b',   // More capable
  'coding_tasks': 'deepseek-coder:6.7b', // Specialized
  'form_filling': 'yi:6b',             // Lightweight
  'chinese_content': 'chatglm3:6b',    // Language specific
  'ecommerce': 'deepseek-coder:6.7b'   // Automation focus
};
```

Browser Automation Requirements

- Must be visible to user
- Add delays between actions (2-3 seconds)
- Stream screen recording in real-time
- Handle multiple browser sessions
- Auto-cleanup after 24 hours

Agent Codename Generation

```javascript
const generateAgentCode = () => {
  const prefixes = ['PHOENIX', 'CYBER', 'GHOST', 'VIPER', 'SHADOW', 'NEXUS', 'OMEGA'];
  const prefix = prefixes[Math.floor(Math.random() * prefixes.length)];
  const suffix = Math.floor(Math.random() * 9999).toString().padStart(4, '0');
  return `${prefix}-${suffix}`;
};
```

Cost Tracking

```javascript
const trackCosts = {
  aiModel: 0.0000,   // Free Ollama
  browser: 0.0000,   // Free Playwright
  server: 0.005,     // Minimal compute
  total: 0.005       // Target under $0.01
};
```

🎯 KEY IMPLEMENTATION POINTS

1. **User sees ONE agent** - Never expose multiple specialists
2. **Live browser automation** - User must watch automation happen
3. **FREE models only** - Use Ollama for all AI calls
4. **MCP coordination** - Route tasks to appropriate specialists
5. **24-hour expiration** - Auto-terminate agent sessions
6. **Hacker aesthetic** - Terminal theme, green text, codenames
7. **EXECUTE button** - Clear trigger for automation
8. **Real-time streaming** - Browser actions must be visible

Core Files Needed:

- `UnifiedAgentInterface.jsx` - Main user interface
- `MCPOrchestrator.js` - Task routing and coordination
- `SpecialistAgents/` - Individual agent implementations
- `LiveBrowserStream.js` - Browser automation streaming
- `PaymentHandler.js` - Stripe integration
- `database.sql` - Schema setup

External Dependencies:

- Ollama (AI models)
- Playwright (browser automation)
- Stripe (payments)
- Socket.io (real-time streaming)