LLM Agent Proof-of-Concept (POC): Browser-Based Multi-Tool Reasoning

Modern LLM-powered agents aren't limited to text—they can combine LLM output with external tools like web search, pipelined APIs, and even live code execution!
This proof-of-concept walks you through building a browser-based agent that can use several tools, looping as needed to accomplish a goal.

Overview: POC Requirements

Goal:
Build a minimal JavaScript-based LLM agent that can:

- Take user input in the browser.
- Query an LLM for output.
- Dynamically trigger tool calls (e.g., search, AI workflow, code execution) based on LLM-chosen actions.
- Loop until the task is complete, integrating results at each step.

Core Agent Logic

The core logic is provided by the Python loop below - but it needs to be in JavaScript.

```python
def loop(llm):
    msg = [user_input()]  # App begins by taking user input
    while True:
        output, tool_calls = llm(msg, tools)  # ... and sends the conversation + tools to the LLM
        print("Agent: ", output)  # Always stream LLM output, if any
        if tool_calls:  # Continue executing tool calls until LLM decides it needs no more
            msg += [ handle_tool_call(tc) for tc in tool_calls ]  # Allow multiple tool calls (may be parallel)
        else:
            msg.append(user_input())  # Add the user input message and continue
```

Supported Tool Calls

Your agent should call these tools as needed:

- Google Search API: Return snippet results for user queries.
- AI Pipe API: Use the aipipe proxy for flexible dataflows.
- JavaScript Code Execution: Securely run and display results of user- or agent-provided JS code within the browser.

UI/Code Requirements

- Model Picker: Use bootstrap-llm-provider so users choose the LLM provider/model.
- LLM-Agent API: Use OpenAI-style tool/function calls so the LLM can ask for tool actions and receive their results.
- Alert/Error UI: Show errors gracefully with bootstrap-alert.
- Code Simplicity: Keep all JavaScript and HTML as simple and small as possible—maximal hackability is the goal!
- Implementation Reference: Use apiagent as a starting design, but trim non-essential code.

Example Agent Conversation

Here's a sample "reasoning loop" in action:

```
User: Interview me to create a blog post.
Agent: output = Sure! What's the post about?, tool_calls = []

User: About IBM
Agent: output = Let me search for IBM, tool_calls = [search("IBM")]

Agent: output = OK, IBM is a big company founded in ..., tool_calls = []

User: Next step, please.
...
```

Deliverable

A browser JS app with LLM conversation window and three working tool integrations:

- Google Search Snippets
- AI Pipe proxy API
- JS code execution (sandboxed)

Use OpenAI's tool-calling interface for all tool invocations. Show errors with bootstrap-alert. Keep your code minimal and easy to extend.

Evaluation Criteria (2 Marks)

Criteria	**Marks**
Output functionality	1.0
Code quality & clarity	0.5
UI/UX polish & extras	0.5