Building Interactive Tools with MCP Elicitation

Ashwin Hariharan

Developer Advocate, **Redis**

About the speaker

@booleanhunter

github.com/booleanhunter

linkedin.com/in/iyerashwin
hariharan

- Ashwin Hariharan

DEVELOPER ADVOCATE

Redis

Educating and enabling developers in building fast, reliable AI systems and user experiences at scale.

What we'll cover

1

The default

Where one-shot tool calls fall short, and what we do instead today.

2

Elicitation + Demo

- What MCP elicitation actually is
- One tool, three workflow categories, in action

3

Building it

Architecture, schemas, and how to start.

If your MCP tool is just a wrapper around your API, you're doing it wrong

"MCP is just an API wrapper" - only if you use it that way.

The Problem

If your MCP server simply wraps thin REST endpoints, it often becomes redundant overhead.

The Nature of APIs

APIs are **transactional** (one-shot), not conversational. Request in, response out.

The MCP Advantage: Elicitation

MCP adds the loop back, making tools conversational with a human-in-the-loop.

A few scenarios

The one-shot pattern works when the prompt has all the info. Often, it doesn't.

schedule_meeting

"design review Friday at 2pm."

Which calendar? Who's invited?

book_flight

"trip to Japan."

Tokyo? Kyoto? Osaka?

send_email

"send Sarah an update."

Sarah who? Which address?

What happens without human-in-the-loop?

Current limitations of agent behavior when context is incomplete

OPTION A

Guess

The agent hallucinates a value based on vague input.

"Book the flight to Tokyo because the user said Japan."

OPTION B

Fail

Tool returns an error. Agent surfaces technical failure as text.

User is forced to retype the entire prompt with more info.

OPTION C

Bounce and Loop

Agent bounces question back in prose. User answers. Agent re-extracts.

New tool call. Loops for each missing slot individually.

Token-wasteful, context-resetting.

None of these are great.

Move past the single shot, API-wrapper model

The goal is broader: **interactive tools** - tools that collaborate with users mid-execution instead of failing or guessing.

MCP Elicitation

The Mechanism of MCP Elicitation

Server pauses mid-execution, asks a typed question, resumes on the response

Benefits vs. Looping/Failing:

Typed Schemas
Server asks with a schema, not text. No re-extraction needed.

Session Continuity
Resumes in the same session. No context reset or token waste.

The Schema defines the response *shape*.

The Client decides the UI (form, chips, dropdown, voice).

Three workflow categories where elicitation pays off

One tool, three different shapes of friction

Ambiguous input

The user said something, but the tool can't pin it down

Missing context

The user just didn't say enough

Auth flows

The tool needs permission.

We'll watch one tool handle all three without the user ever filling out a form upfront.

An example of Elicitation

A shell-command tool that confirms before destructive operations

```
import type { McpServer } from '@modelcontextprotocol/sdk/server/mcp.js';
import { tool } from '@langchain/core/tools';
import { z } from 'zod';

const DANGEROUS = ['rm', 'mv', 'dd', 'shutdown', 'kill', 'chmod'];

export const makeRunCommandTool = (server: McpServer) => tool(
  async ({ command, args }) => {

      if (DANGEROUS.includes(command.trim())) {

          const result = await server.elicitInput({
              message: `Confirm running '${command}'?`,
              requestedSchema: {
                  type: 'object',
                  properties: { confirm: { type: 'boolean', title: 'Confirm' } },
                  required: ['confirm'],
              },
          });

          if (result.action !== 'accept' || !result.content?.confirm) {
              return { error: 'Command rejected by user' };
          }
      }
      return await execShell(command, args);
  },
  { name: 'runCommand', schema: z.object({ command: z.string() }) },
);
```

`server.elicitInput({ message, requestedSchema })` pauses, surfaces the schema, resolves with `{ action, content }`.

One tool, one schema, one boolean response.

[Demo]

Let’s see one working with Agentic AI

Wait. We can't ask for a password in a form.

Form mode is the wrong shape for credentials

SCENARIO

User wants to save the trip to Google Calendar. The tool needs **OAuth**.

Obvious move? Pop up an input box asking for Google username and password. *Right?*

LIMITATIONS

Form mode is **not the right fit** for:

- Credentials and secrets
- OAuth tokens
- Payment info

Use URL Mode Instead.

Architecture: how this was built

Tech stack

Two clients, one backend - AG-UI for streaming, MCP for connecting

INTERNAL APP / WEB UI

`React`

UI framework – sidebar chat, canvas

`@ag-ui/client`

Streams agent events; renders elicit cards + state snapshots.

EXTERNAL MCP HOSTS

`Claude, VS Code, etc.`

Any MCP-compatible client invokes the same tool and renders elicits natively.

AG-UI / SSE

MCP

BACKEND SERVICES

`@langchain/langgraph`

3-node workflow + ReAct loop – the agent engine.

`@ag-ui/langgraph`

Event types + LangGraph bridge for the web UI adapter.

`OpenAI`

LLM provider for chat completions + text embeddings.

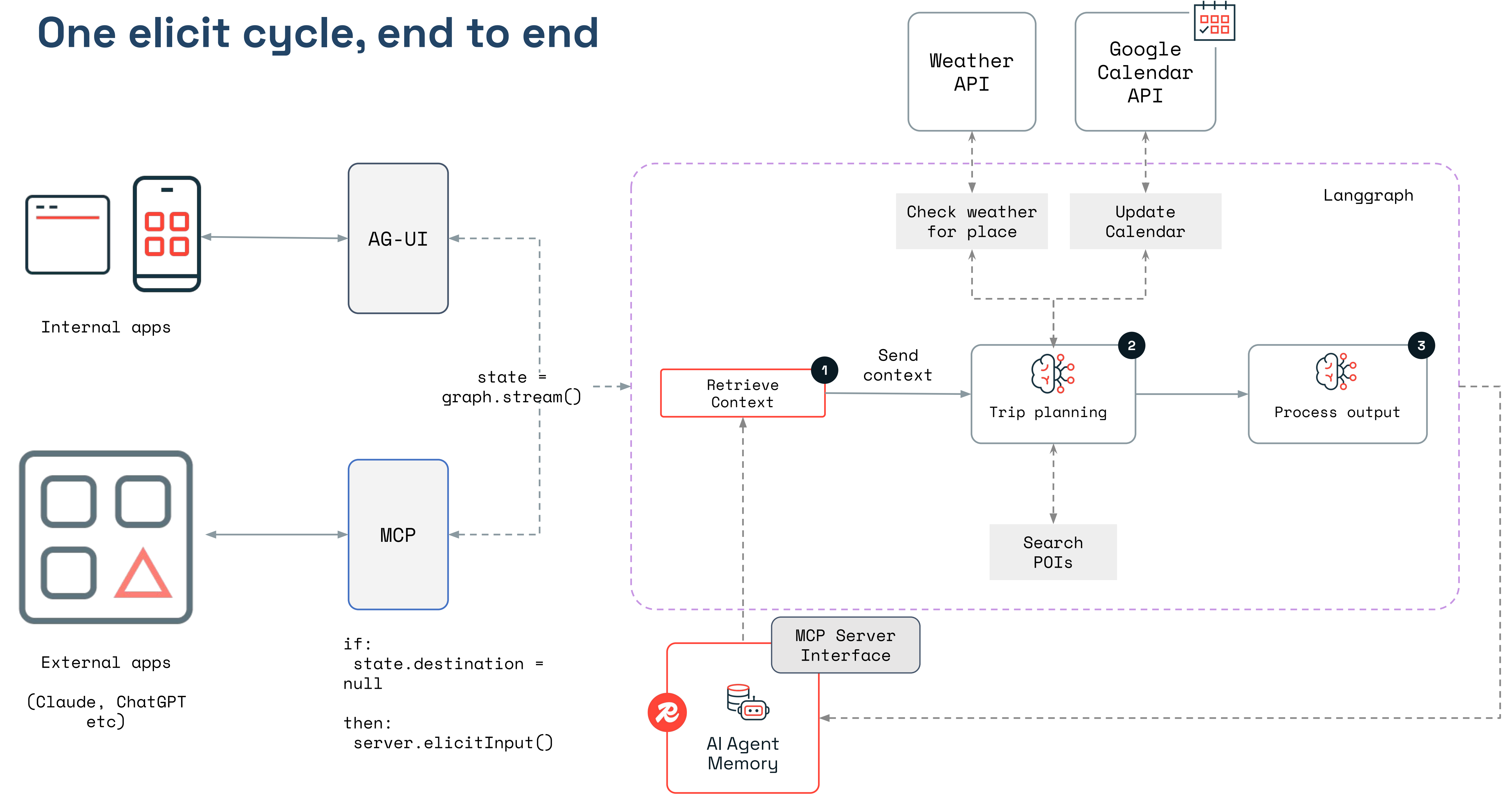
`@modelcontextprotocol/sdk`

MCP server adapter – exposes tools and drives elicitInput.

`Redis`

Agent memory store – working memory + long-term memory, exposed as **MCP**

One elicit cycle, end to end



Schemas are the contract

The server commits to the data shape, allowing clients to render the native UI.

```
{
    "type": "object",
    "properties": {
        "destination": {
            "type": "string", "title": "Where to?",
            "oneOf": [
                { "const": "tokyo", "title": "Tokyo" },
                { "const": "kyoto", "title": "Kyoto" },
                { "const": "osaka", "title": "Osaka" }
            ]
        },
        "startDate": { "type": "string", "title": "Start date", "format": "date" },
        "endDate":   { "type": "string", "title": "End date",   "format": "date" },
        "interests": {
            "type": "array", "title": "What are you in the mood for?",
            "items": {
                "anyOf": [
                    { "const": "food", "title": "Food" },
                    { "const": "culture", "title": "Culture" }
                ]
            }
        }
    },
    "required": ["destination"]
  }
```

The schema language is kept deliberately minimal

Less expressive power per schema. More clients that actually implement it.

Allowed Primitives

- `string (format, enum, pattern)`
- `number / integer`
- `boolean`
- `array of primitives`
- **`Flat object`**

Restrictions

❌ No nested objects

❌ No conditionals

❌ No embedded HTML

Full JSON Schema support might make elicitation more powerful, but **less interoperable**. Most client teams would skip it entirely.

18

[Demo] Same app, different host

Same server, replayed in a different MCP client

Travel Agent Panel

"Plan a 5-day trip."

→ Same flat-object, rendered in the new host's native panel UI.

Calendar Integration

"Save to Google Calendar."

→ Same URL-mode auth, triggering the standard OAuth flow.

Zero changes to the server. Authored once; rendered by any MCP client.

Whether it's today, tomorrow, or in whatever custom UI someone builds next year, the protocol ensures that your implementation remains consistent.

How to get started

Identify what to ask

The schema isn't enough without the right context

A tool can have the perfect schema and still be a bad tool if it doesn't know **what's worth asking.**

Conversation history

What's been said this turn / this session.

User state

Preferences, past sessions, environmental signals.

The prompt itself

Slots the user already filled in their natural-language ask.

The elicit set ≈ **required − known**.

Get the subtraction right and elicitation feels collaborative. Get it wrong - and you either interrogate or guess on the user's behalf.

Context comes from memory

Two types of memory feed context

Session Scoped / Working Memory

- Currently planning a trip to Barcelona.
- Dates picked: Nov 12 - 19.
- Tracks what's in flight, this turn.

Cross session / Long-term memory

- Prefers window seats.
- Always picks museums and food tours.
- Compounds over time.

Memory needs to be fast, fresh, and compounding

Fast

Elicit decisions happen in real-time. Sub-millisecond reads keep the flow snappy.

High latency or lag breaks the collaborative feel.

Fresh

Reflects what the user said two seconds ago.

Eventually-consistent memory makes the tool re-ask things the user just answered.

Compounding

Every session makes the next elicit smarter.

The more the tool knows, the less it has to ask.

Redis Agent Memory provides session + long-term memory in real time, for a smarter elicitation / human-in-the-loop experience.

Debug with the MCP Inspector

The tool to reach for while developing

Inspect Raw JSON

Shows raw `elicitation/create` and `elicitation/response` JSON for every cycle.

Schema Validation

Verifies your server is emitting the right schema and handling responses correctly.

24

Tools that ask clarifying questions aren’t new.

Same pattern, three eras

What's new is the protocol that standardizes it

CLI prompts

`git rebase -i`

Opens an editor mid-command, asking the user to clarify the rebase plan before continuing.

IDE wizards

`npm init`

Walks the user through fields one at a time. Each value typed, validated, and surfaced as the tool runs.

OS permission dialogs

"Allow camera access?"

Every platform does this. App is mid-execution. System pauses. User decides. App resumes. Same negotiation.

26

Takeaways

1

Three workflow categories

- Ambiguous input
- Missing context
- Auth flows

2

Shape vs. UI

Server commits to the response shape.

Client picks the UI.

That's what makes elicitation portable.

3

Two modes by spec

- **Form**: non-sensitive input
- **URL**: credentials, OAuth, payments

4

Context makes it useful

Elicit set = `required - known`

Only ask for what's not already known.

Does this app feel like it's *collaborating* with me?

Or does it feel like a black box that either works or doesn't?

Improve your AI UX with elicitation:

1. Pick

Choose one tool in your MCP server to focus on.

2. Find

Identify moments of guessing, failing, or unnecessary forms.

3. Ship

Replace that moment with an elicit and ship it.

Thank you.

Any questions? Elicit me on these channels

@booleanhunter

github.com/booleanhunter

linkedin.com/in/iyerashwin hariharan

- Ashwin Hariharan

DEVELOPER ADVOCATE