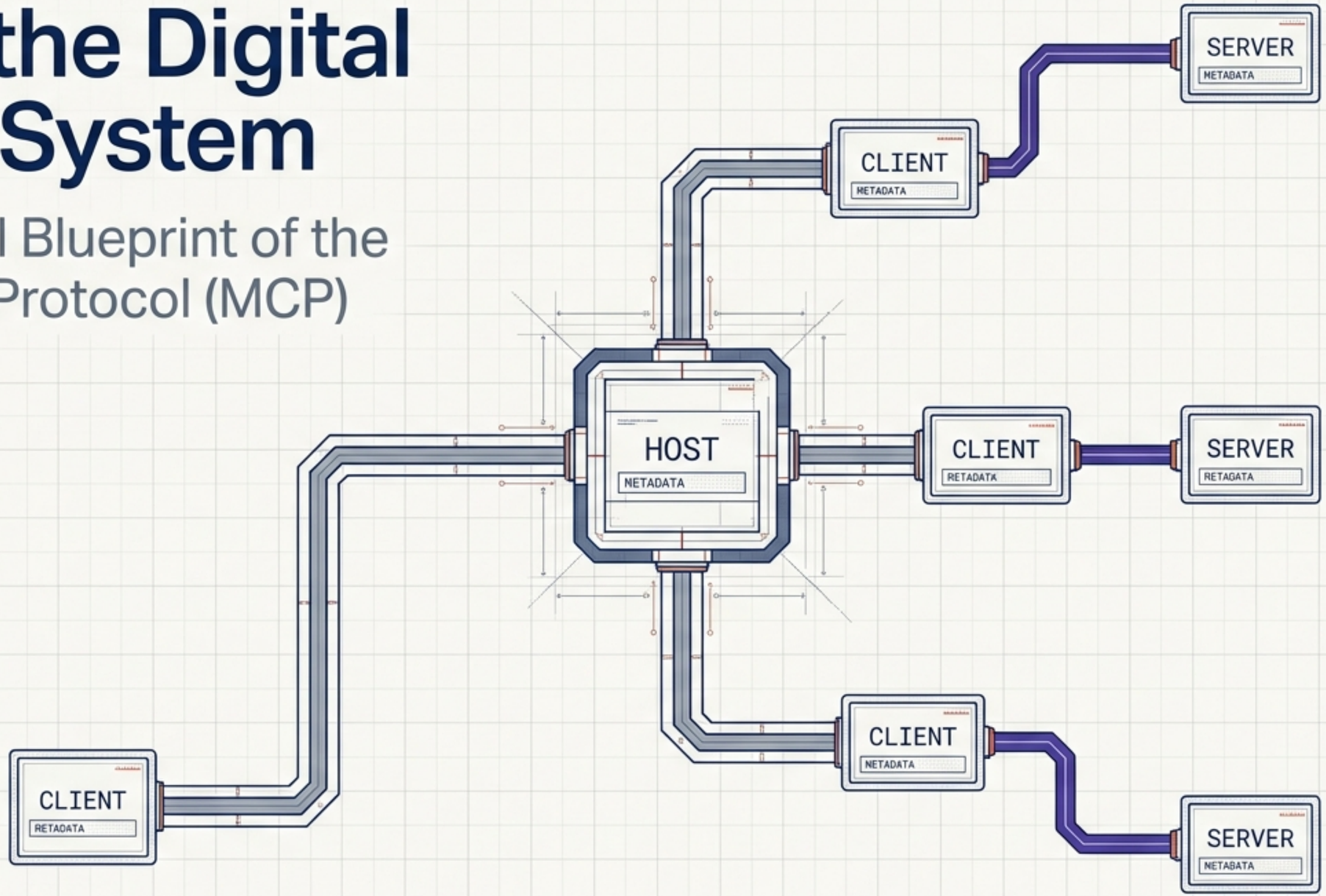
Building the Digital Nervous System

An Architectural Blueprint of the Model Context Protocol (MCP)

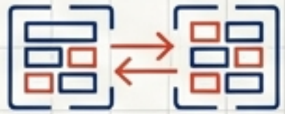


Gemini Notebook

Standardizing Context, Not Orchestration

Scope Boundary

Inside MCP: Context Exchange



JSON-RPC 2.0 communication



Protocol specification & SDKs



Context delivery mechanisms

Outside MCP: Orchestration

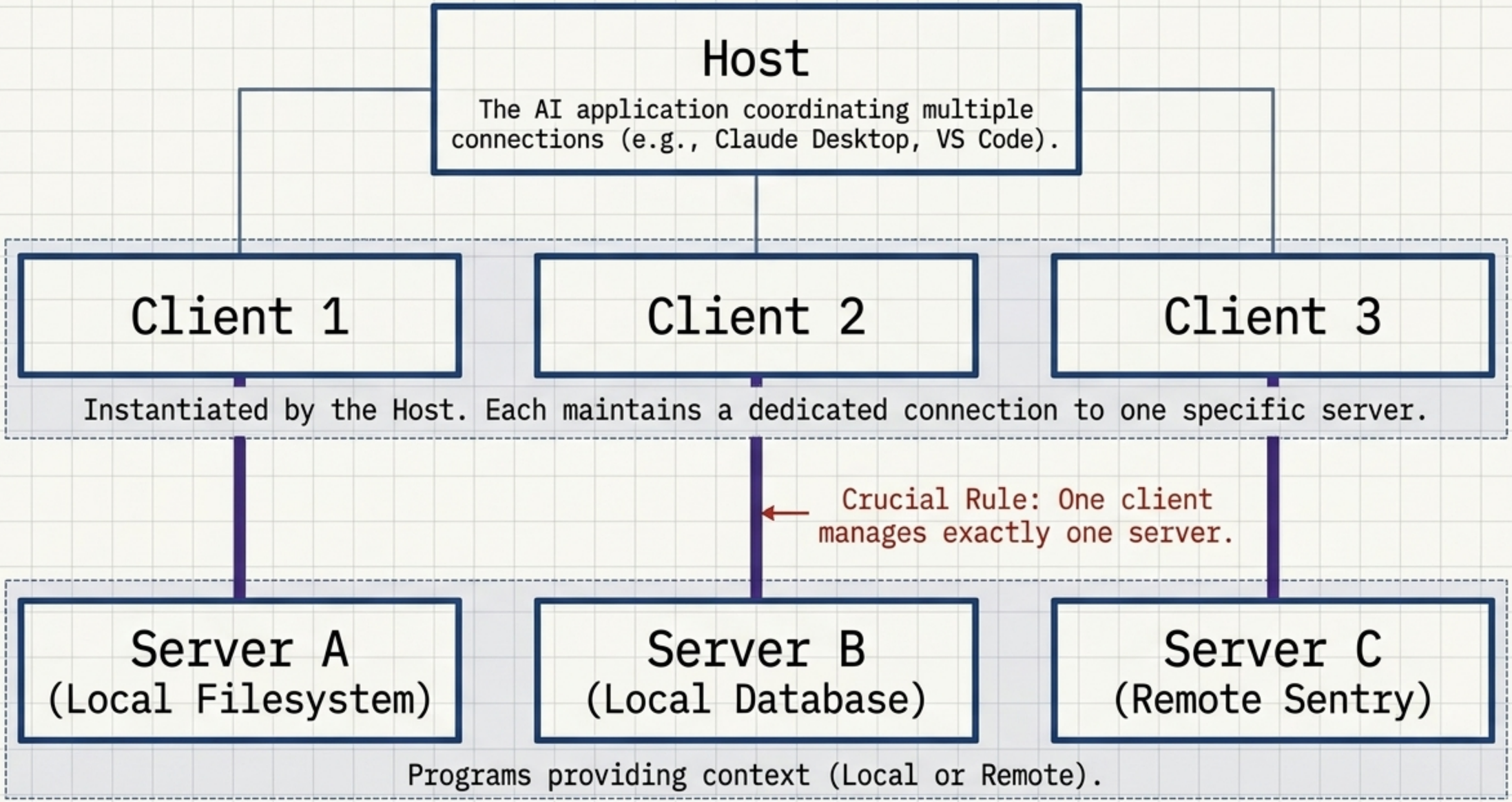
How AI applications use LLMs

How applications manage or store provided context

Key Takeaway: MCP focuses solely on the protocol for context exchange between AI applications and external data sources.

Gemini Notebook

A Strict 1:1 Mapping Between Clients and Servers



Gemini Notebook

The Transport Layer Architecture

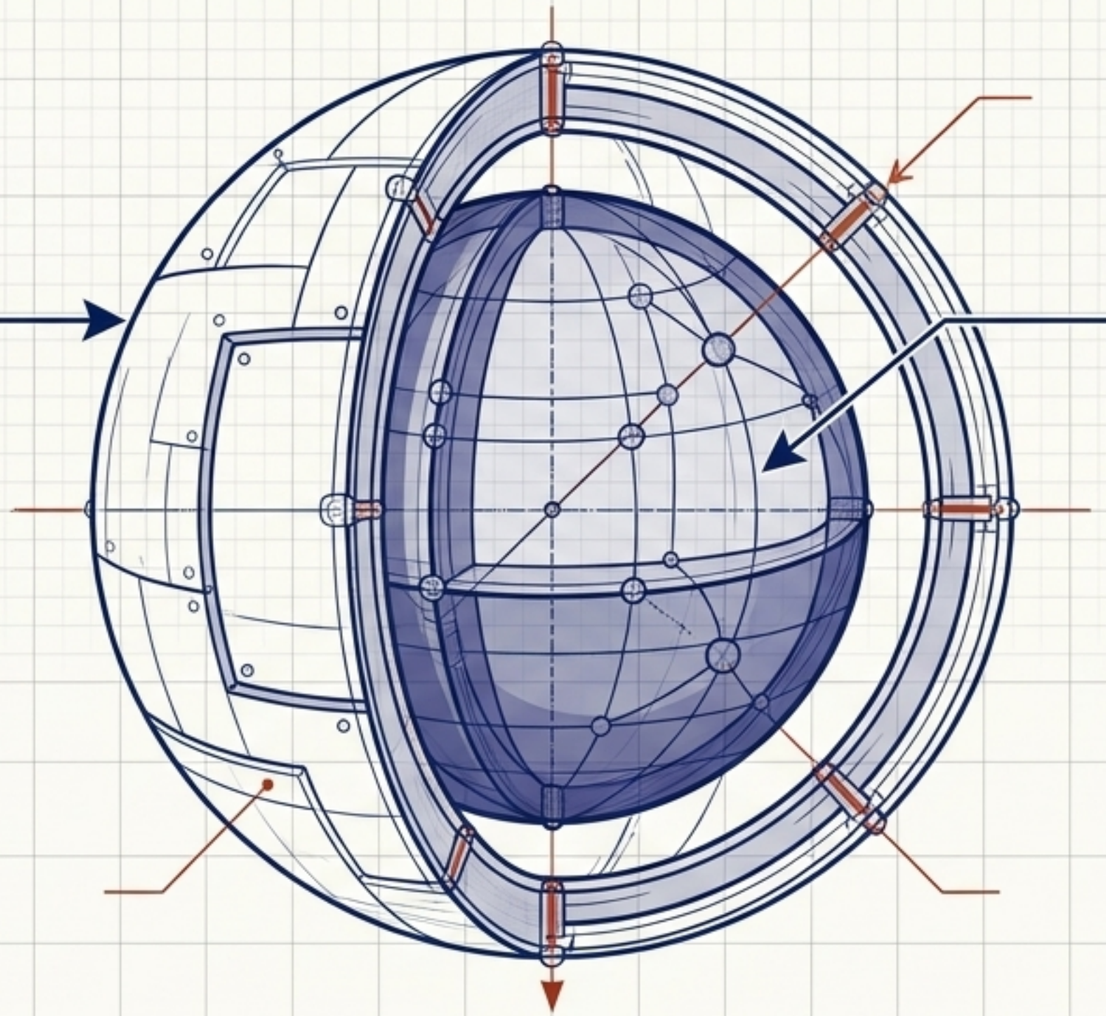
	Stdio Transport	Streamable HTTP Transport
Use Case	Local MCP servers running on the same machine.	Remote MCP servers serving multiple clients.
Communication	Direct process communication via standard input/output streams.	HTTP POST for client-to-server; Server-Sent Events (SSE) for streams.
Authentication	None required (relies on local OS permissions).	Bearer tokens, API keys, custom headers (OAuth recommended).
Performance	Optimal; zero network overhead.	Subject to network latency, but highly scalable.

Gemini Notebook

Abstracting Communication from Exchange

Transport Layer

Manages connection establishment, message framing, and security. Abstracts all networking complexity.



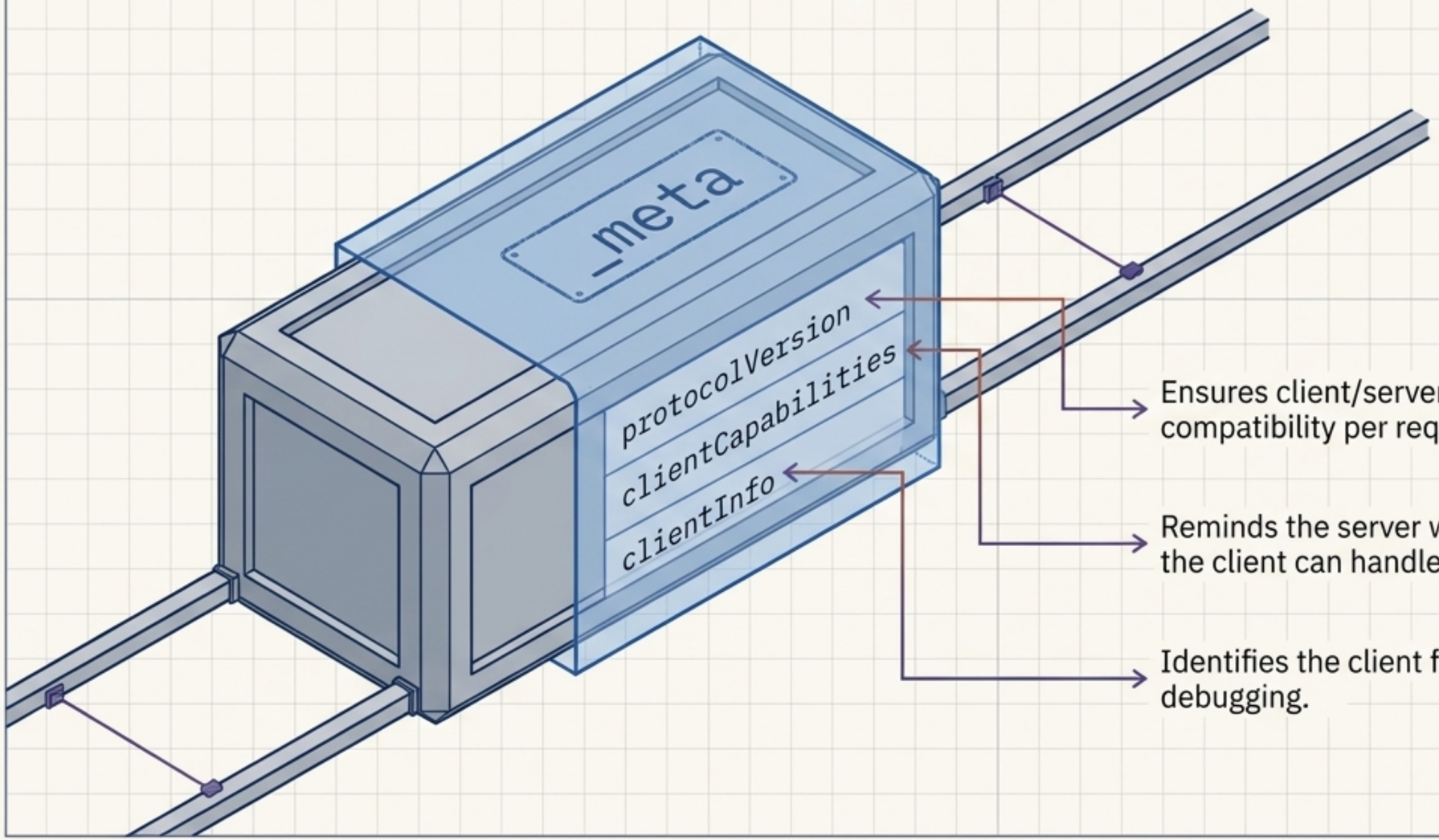
Data Layer

Implements the JSON-RPC 2.0 exchange protocol. Defines message structure, capability discovery, and primitives.

Because the Transport Layer handles the routing, the JSON-RPC 2.0 message format remains identical across all transport mechanisms.

Gemini Notebook

Stateless by Design: The _meta Envelope



Ensures client/server compatibility per request.

Reminds the server what the client can handle.

Identifies the client for debugging.

Sidebar Note

MCP connections do not rely on stored session state. By attaching this metadata envelope to every request, servers can process each JSON-RPC call completely independently.

Gemini Notebook

The Primitives Matrix: Defining Capabilities

Server Primitives (Context Providers)

Tools

Executable functions for AI actions (e.g., API calls, DB queries).

`Discovery: */list,` `Execution: */call.`

Resources

Data sources for contextual information (e.g., file contents).

`Discovery: */list,` `Retrieval: */read.`

Prompts

Reusable templates structuring LLM interactions.

`Discovery: */list,` `Retrieval: */get.`

Client Primitives (Interaction Enablers)

Elicitation

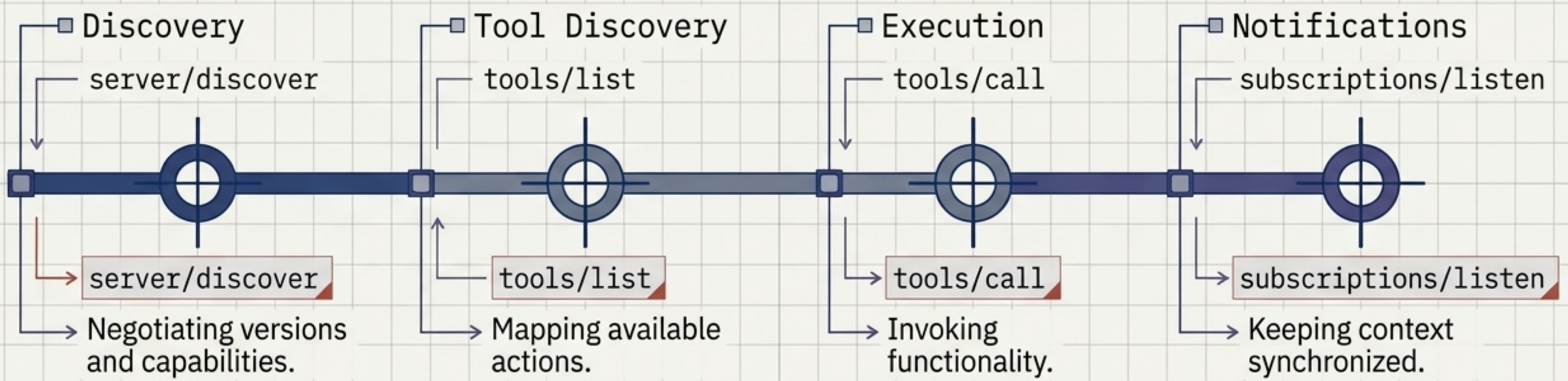
Allows servers to request user input/confirmation via `elicitation/create` (Multi Round-Trip).

Note on Deprecations (v2026-07-28):

Sampling (requesting LLM completions) and Logging to clients are deprecated. Use direct LLM APIs and OpenTelemetry instead.

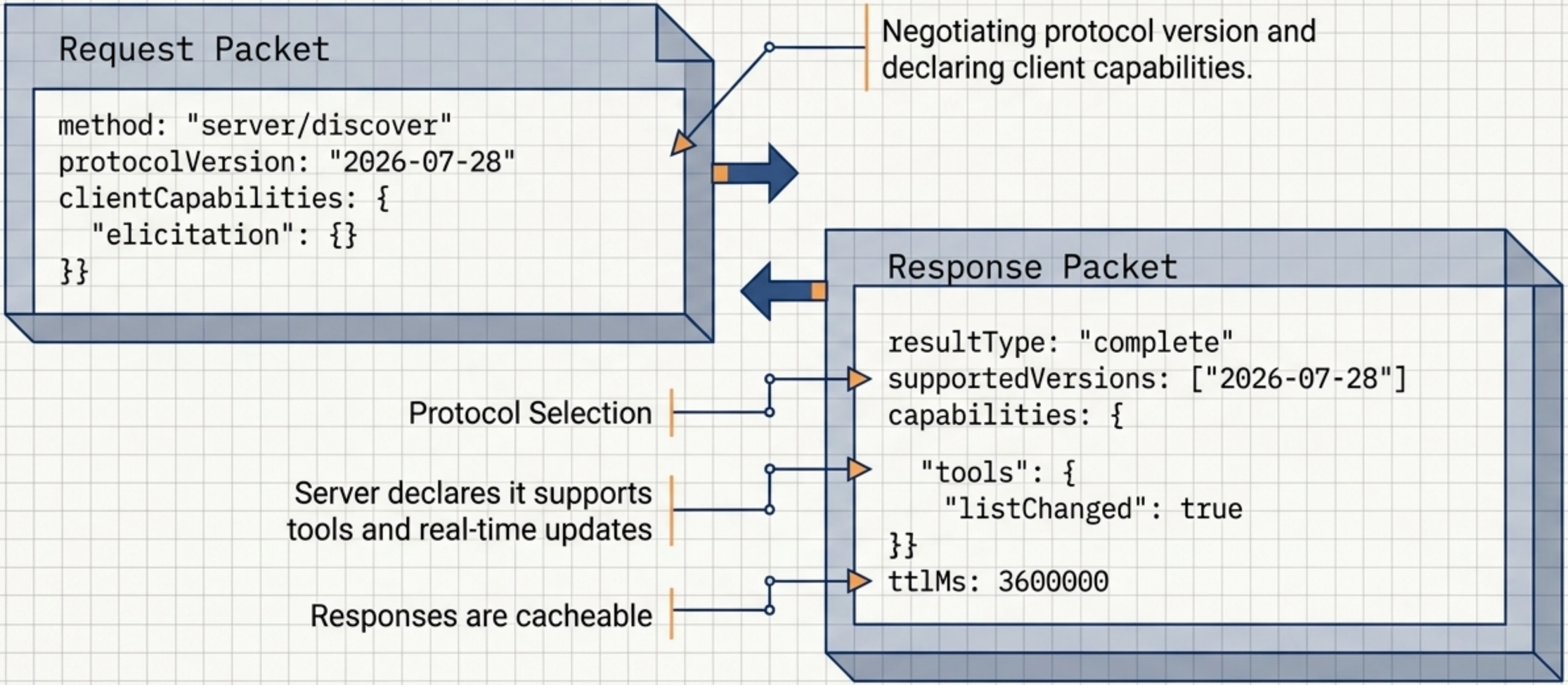
Gemini Notebook

The Lifecycle Arc: From Boot to Real-Time Updates



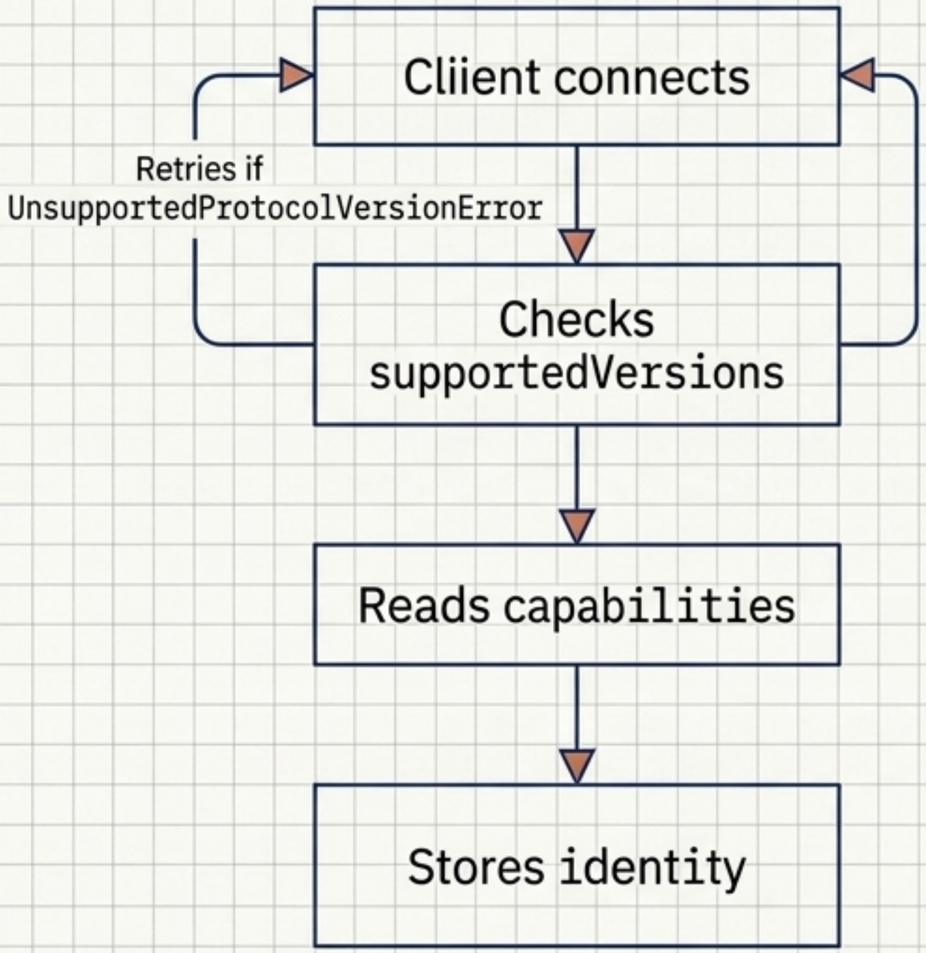
Gemini Notebook

Phase 1: The Discovery Exchange



Gemini Notebook

Connecting the Host to the Protocol



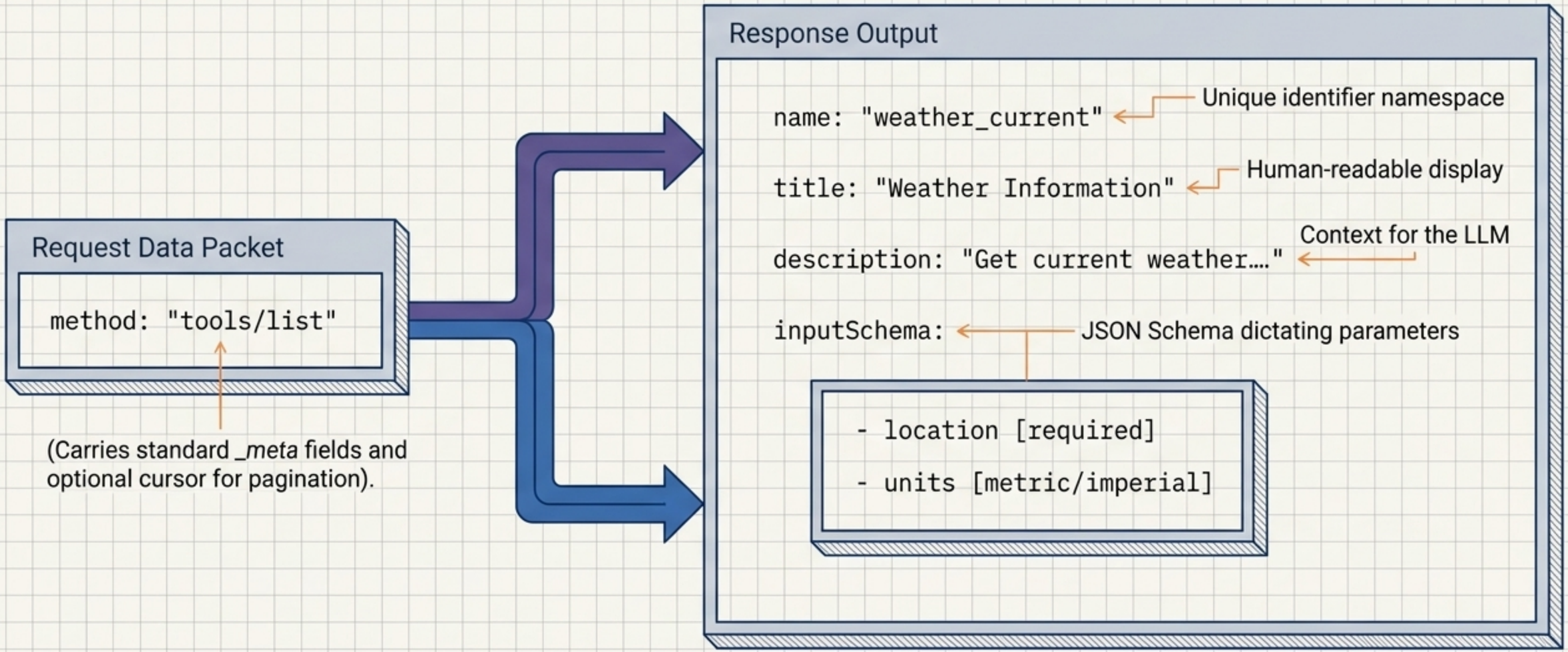
In the AI Application

```
async with Client(stdio_client(config)) as client:
   if client.server_capabilities.tools:
      app.register_mcp_server(client,
         supports_tools=True)
   app.set_server_ready(client)
```

> The Host manager stores discovered capabilities during connection to determine exactly what this specific server can provide.

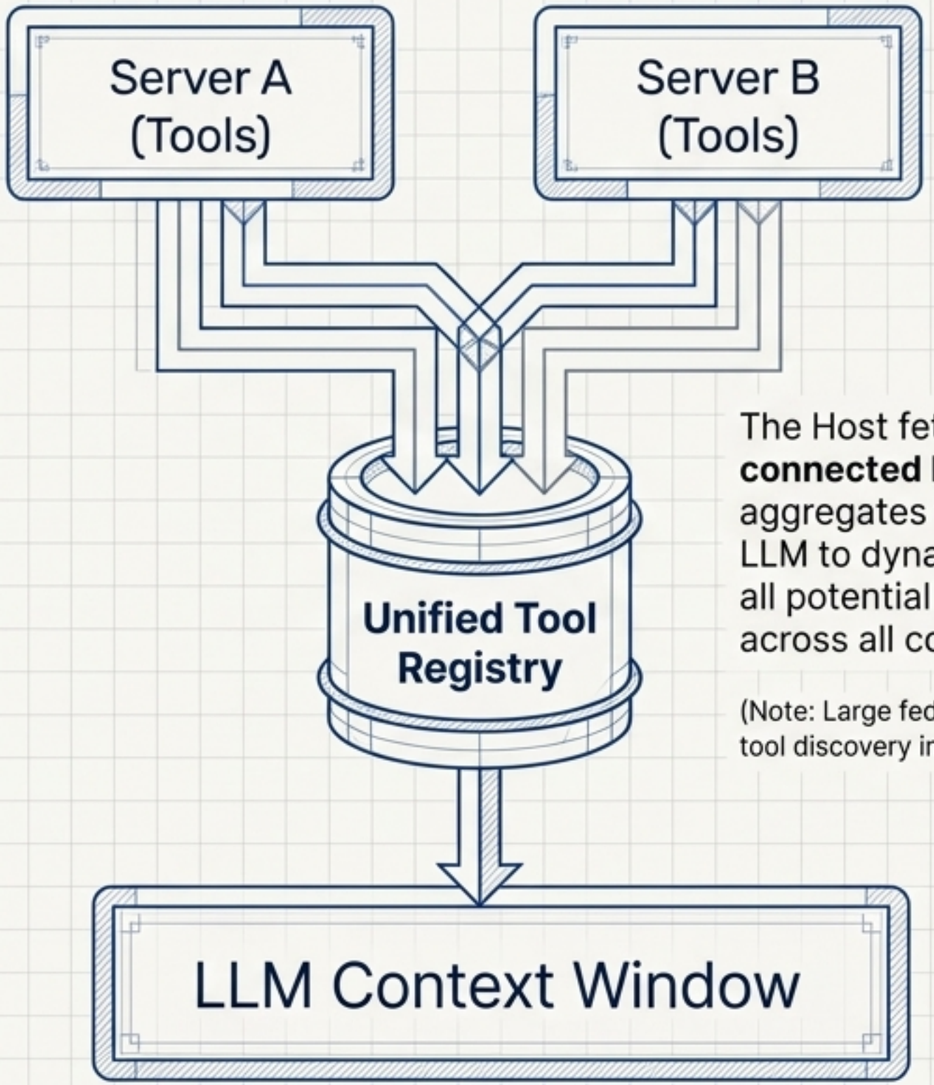
Gemini Notebook

Phase 2: Mapping Available Primitives



Gemini Notebook

Building the Unified Tool Registry



The Host fetches tools from all **connected MCP servers** and aggregates them. This allows the LLM to dynamically understand all potential actions it can take across all connected systems.

(Note: Large federations can use progressive tool discovery instead of upfront loading).

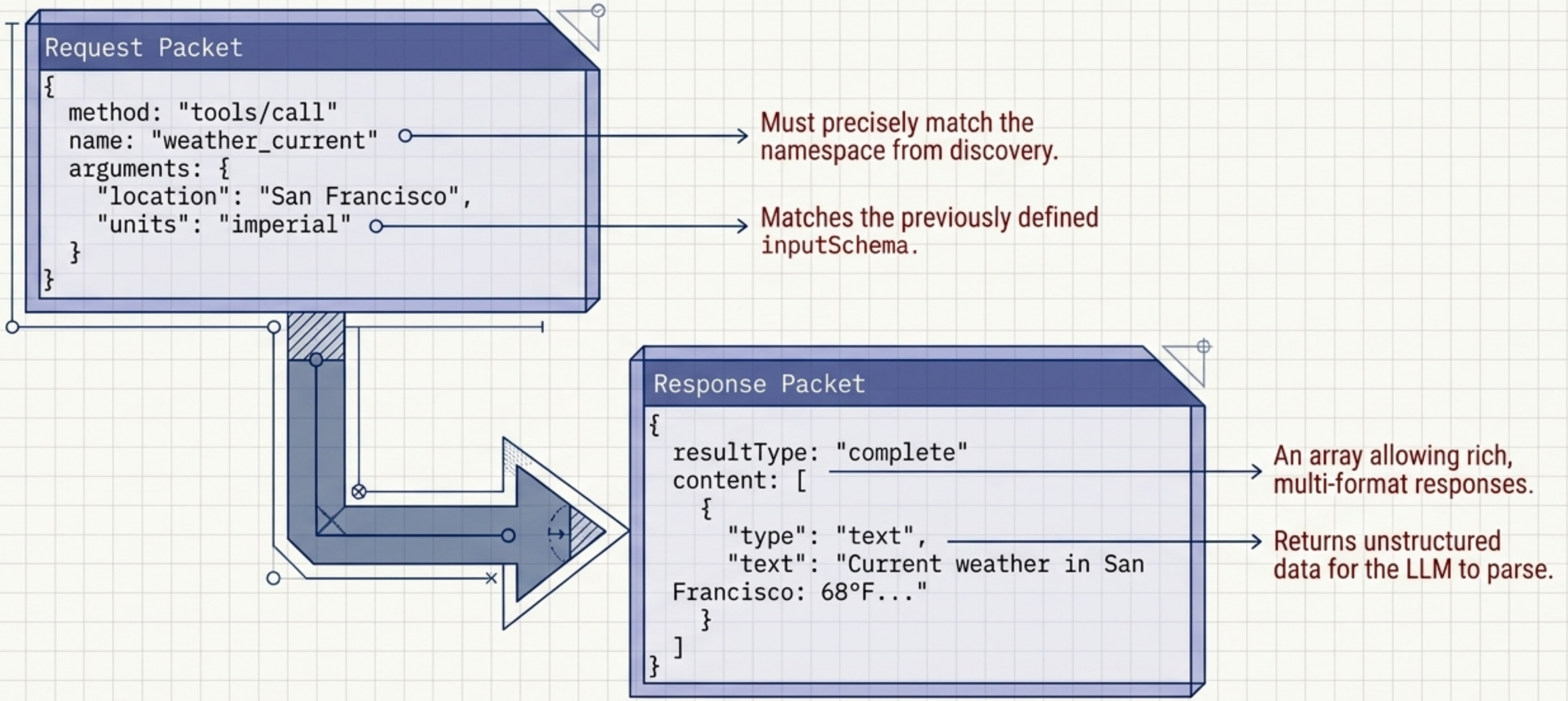
In the AI Application

{Code snippet}

```
available_tools = []
for client in app.mcp_clients():
    response = await client.list_tools()
    available_tools.extend(response.tools)
conversation.register_available_tools(avail
able_tools)
```

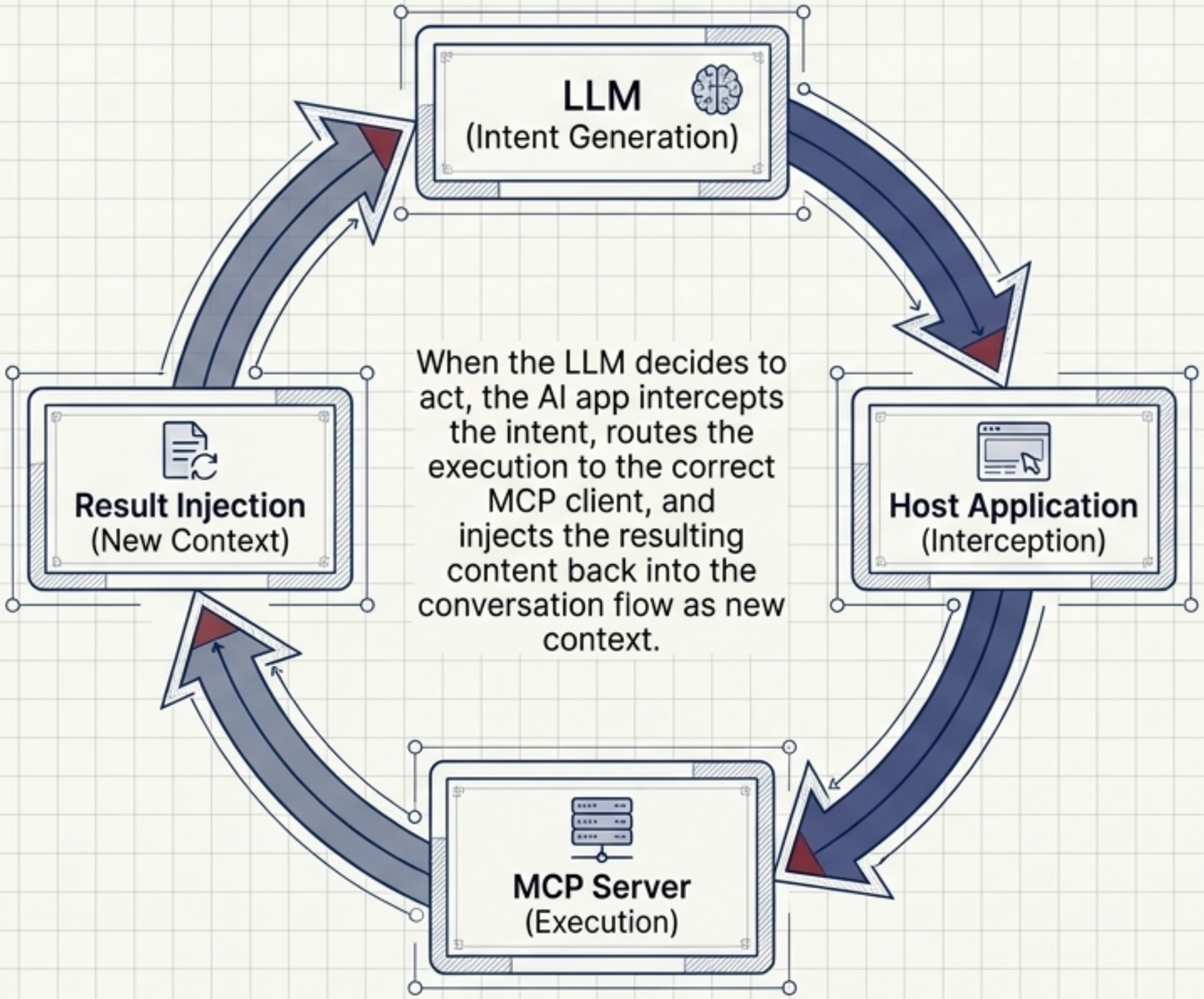
Gemini Notebook

Phase 3: Tool Execution & Type Safety



Gemini Notebook

Intercepting and Injecting Context



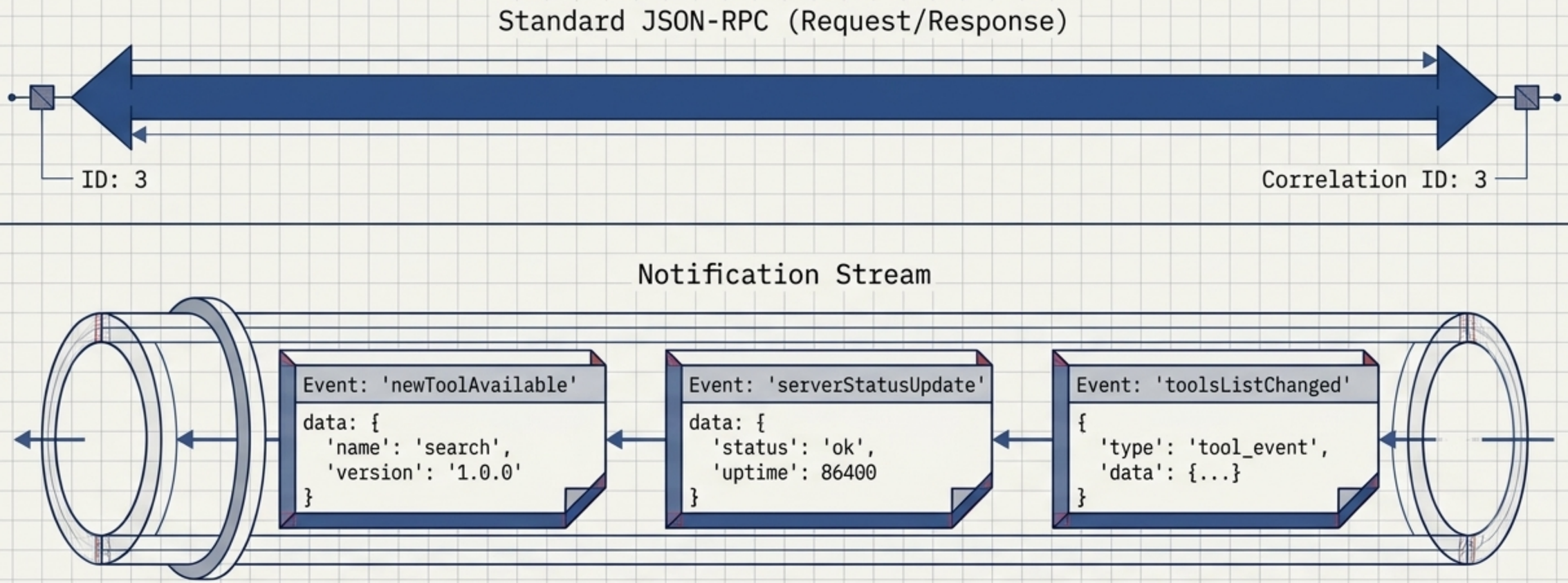
In the AI Application

{Code snippet}

```
async def handle_tool_call(conversation,
name, args):
  client = app.find_mcp_client(name)
  result = await client.call_tool(name,
args)
conversation.add_tool_result(result.content)
```

Gemini Notebook

Phase 4: The Notification Stream Pipe

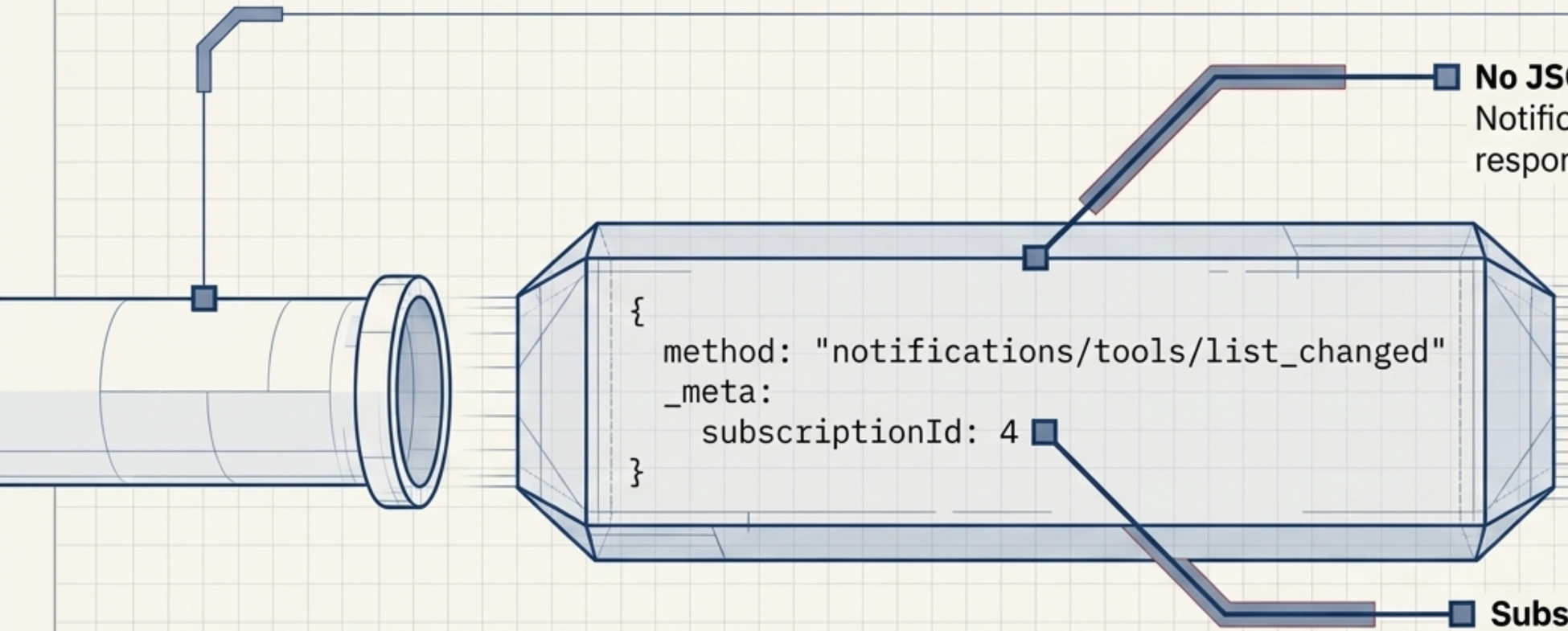


Real-time updates are opt-in. The client opens a long-lived stream via subscriptions/listen, asking for specific events (e.g., 'toolsListChanged': true).

Server Acknowledgment: Returns notifications/subscriptions/acknowledged carrying a subscriptionId. This ID tags all future events flowing down the pipe.

Gemini Notebook

Event-Driven State Changes



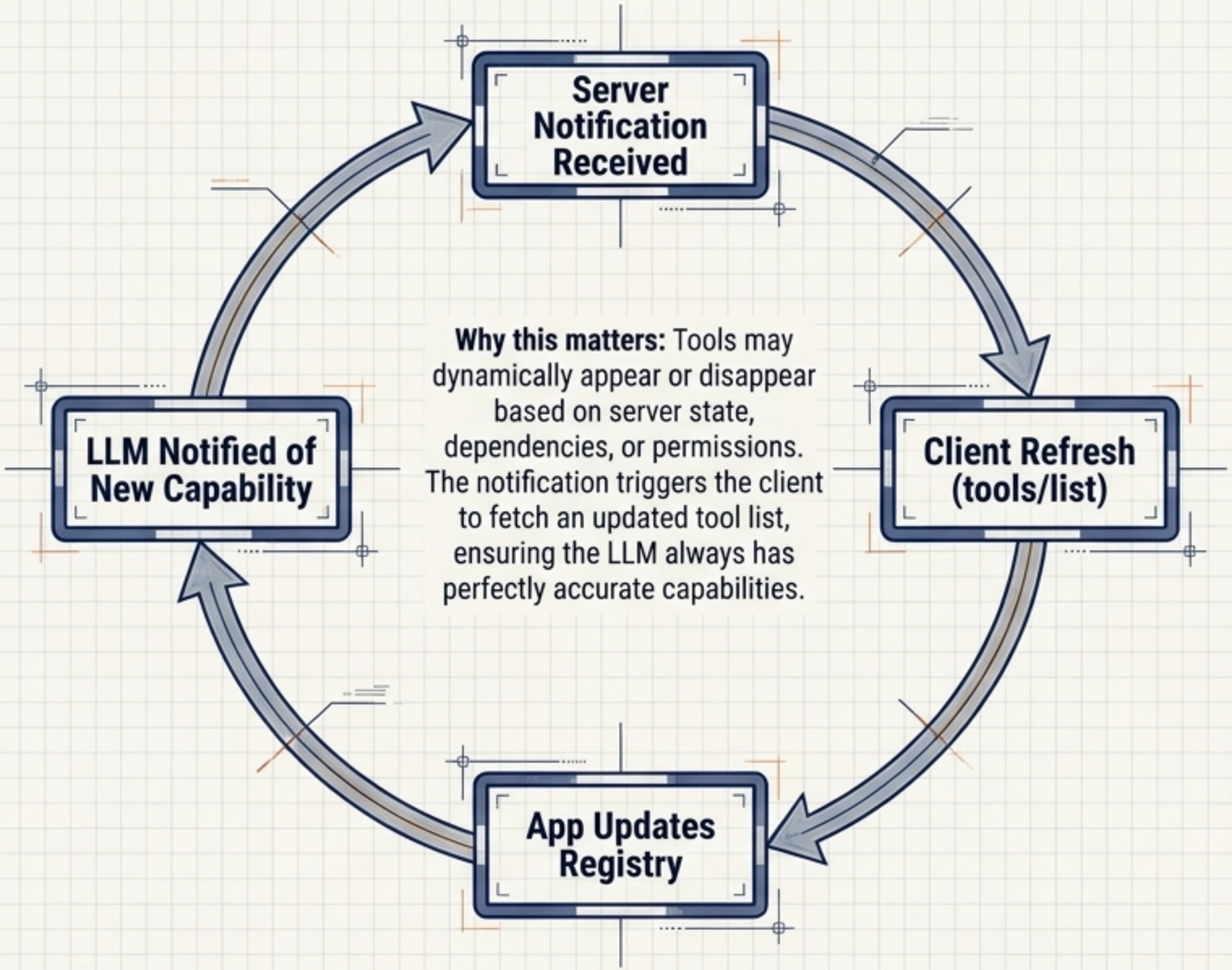
No JSON-RPC ID.
Notification semantics dictate no response is expected or sent.

Subscription ID Tagging
`_meta` correlates the event to the original listen request.

Notifications are event-driven and best effort. There are no guarantees of delivery across transport reconnects, meaning clients should still occasionally rely on polling to preserve freshness.

Gemini Notebook

The Automatic Refresh Cycle



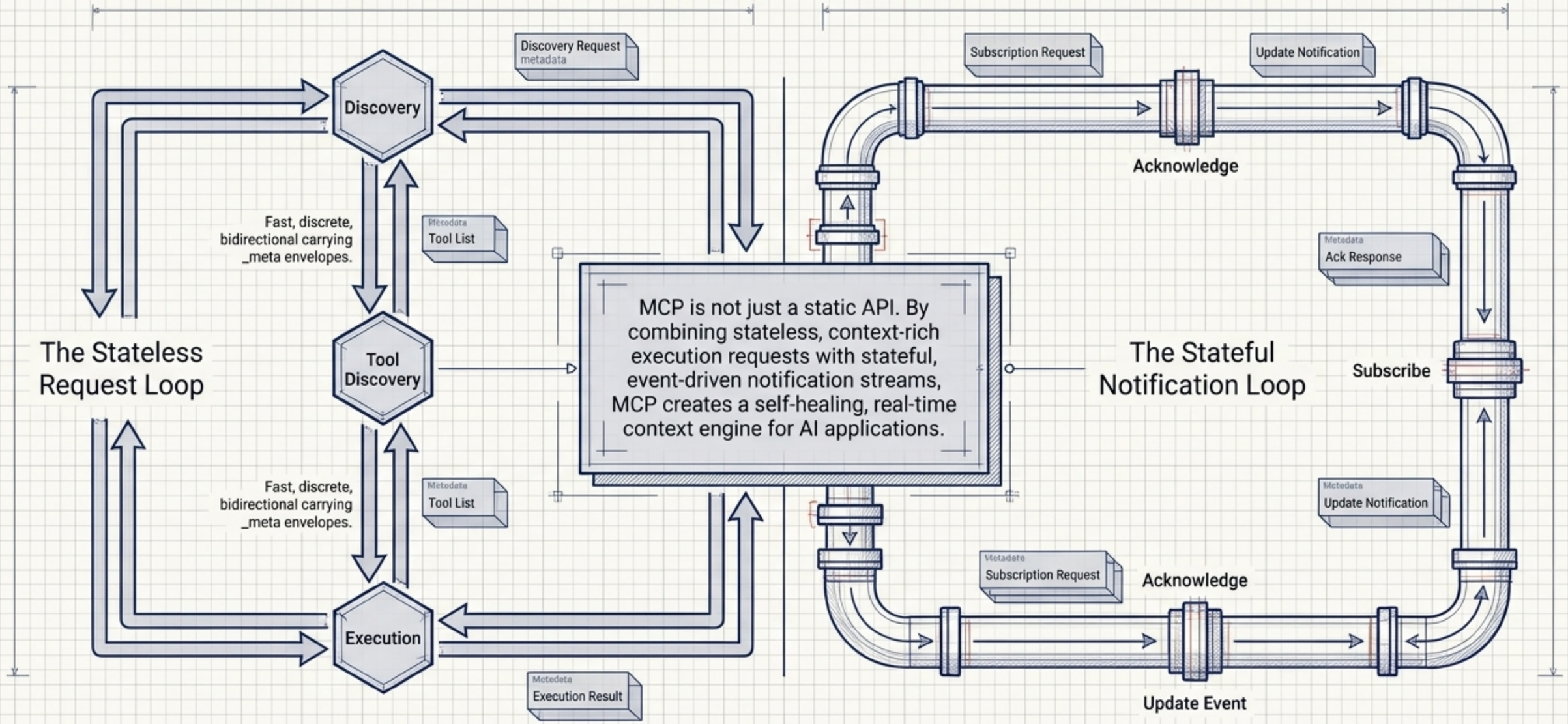
Why this matters: Tools may dynamically appear or disappear based on server state, dependencies, or permissions. The notification triggers the client to fetch an updated tool list, ensuring the LLM always has perfectly accurate capabilities.

In the AI Application

```python
async def follow_tool_changes(client):
    async with client.listen(tools_list_changed=True) as sub:
        async for _event in sub:
            response = await client.list_tools()
            app.update_available_tools(client, response.tools)
            app.conversation.notify_llm()
```

Gemini Notebook

The Complete MCP Interaction Loop



Gemini Notebook