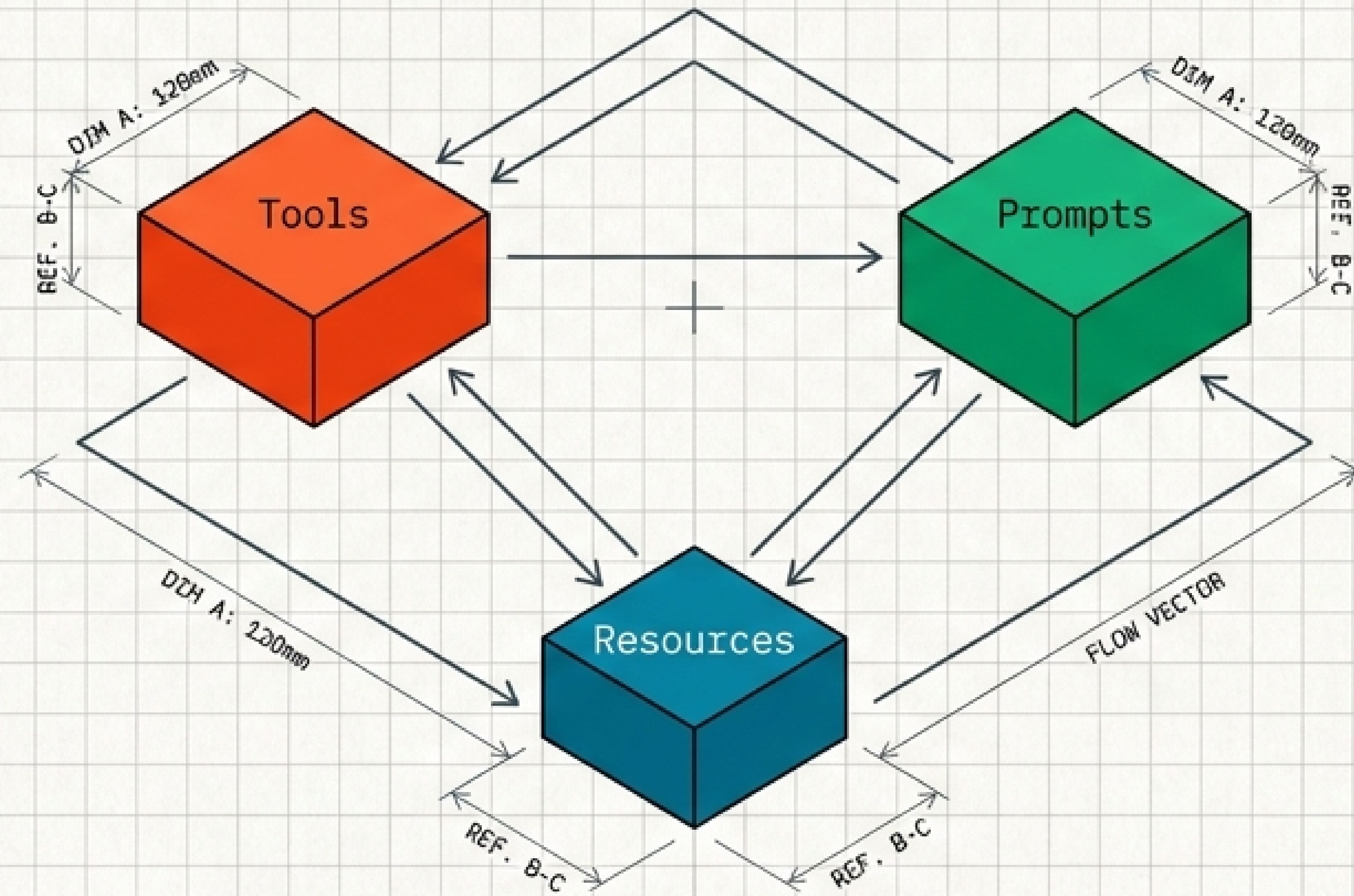
Architecting Model Context Protocol Servers

The Definitive Guide to Tools, Resources, and Prompts



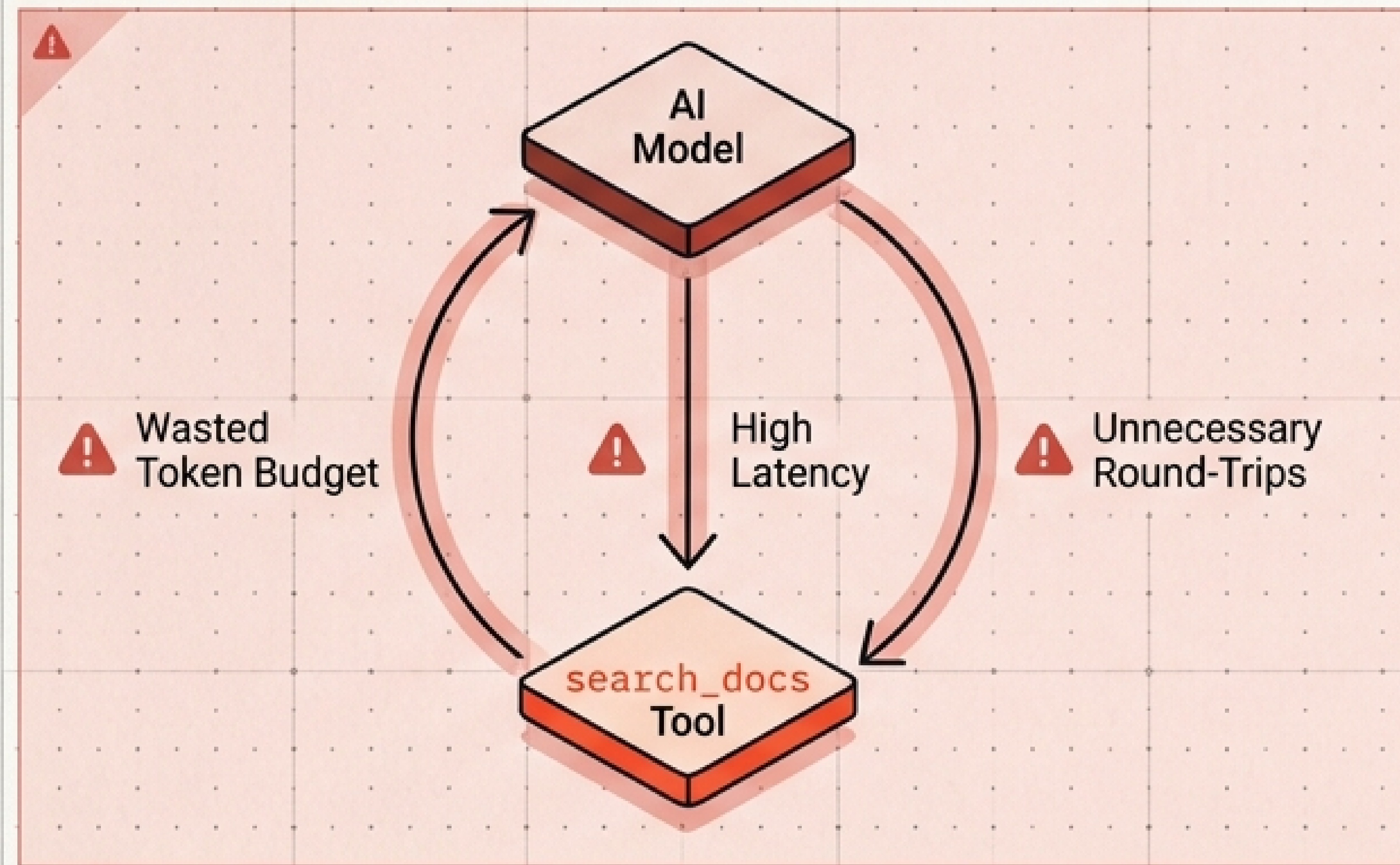
NotebookLM

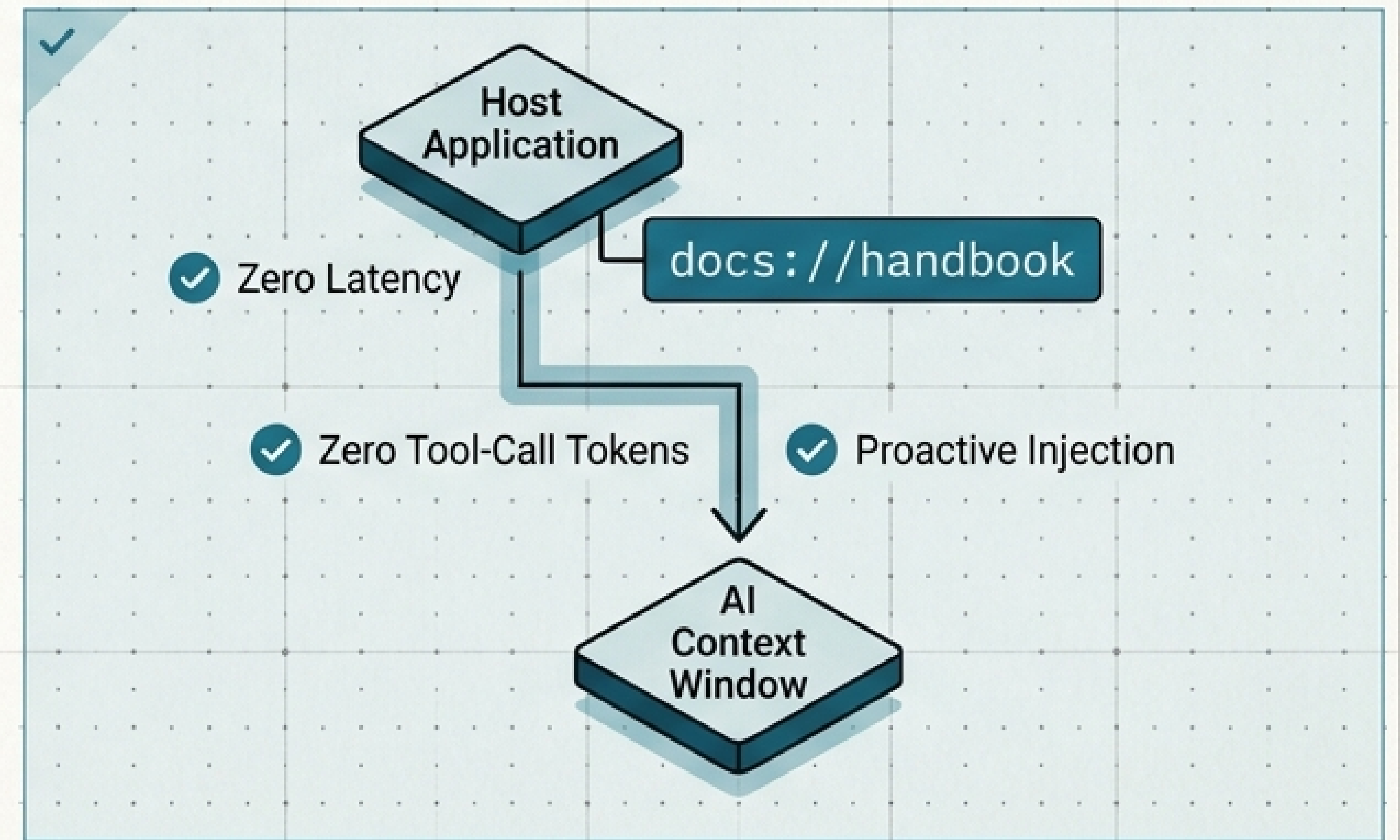
The Mistake Almost Every Developer Makes

Ask an engineer to integrate a read-only knowledge base into an MCP server, and nine out of ten will build a `search_docs` Tool. Fast to write. Reasonable API. Completely wrong primitive.

Conflating Tools and Resources burns your token budget on unnecessary tool-call round-trips and completely removes the host application's ability to pre-load context.

The Token Burner vs. Context Injection

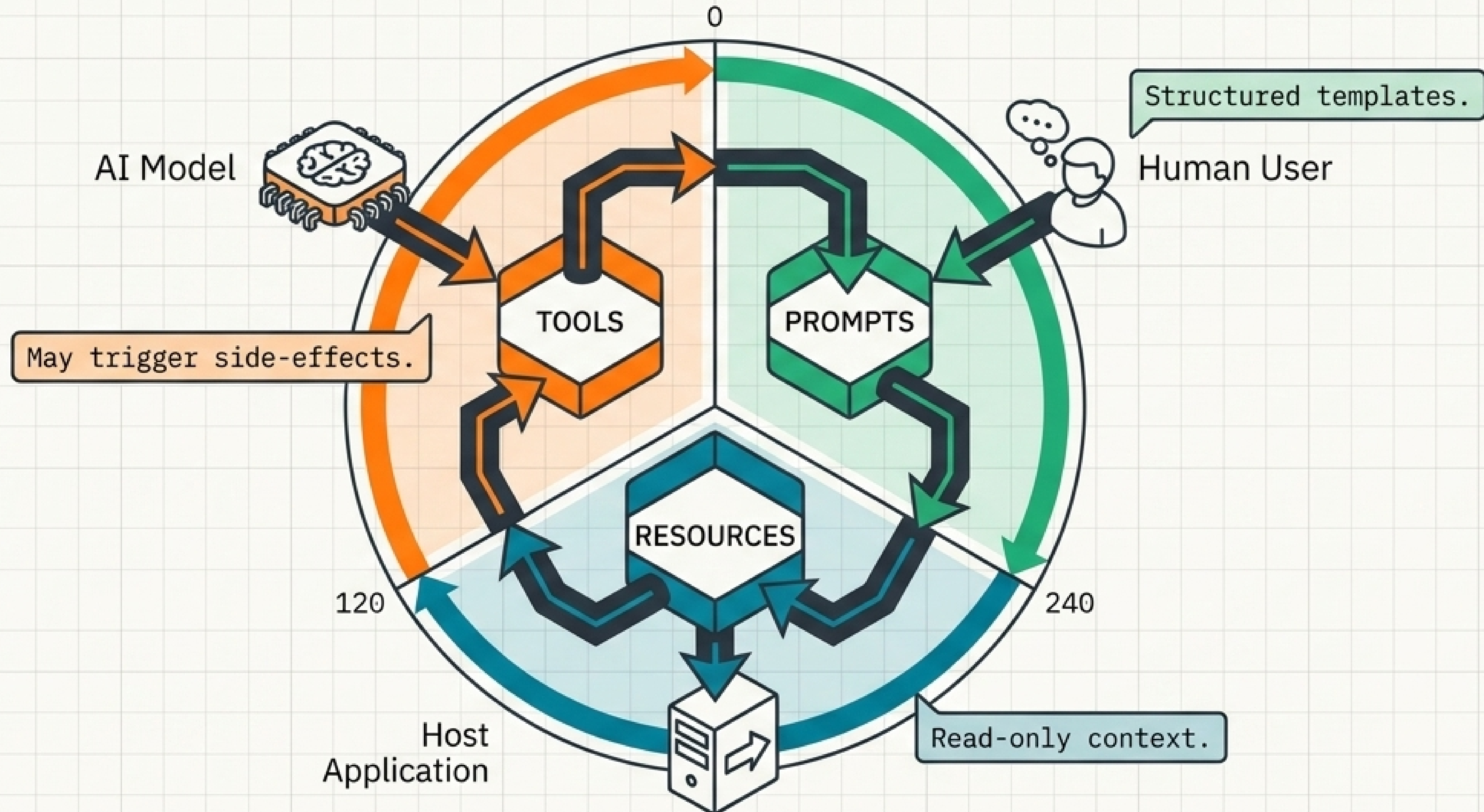




NotebookLM

The Triad Topology

MCP defines exactly three primitive types. Choosing the correct primitive is the primary design decision in server development. The initiation model—who takes the first action—defines the architecture.



NotebookLM

Tools: What the Model Executes

A Tool is an operation the model initiates that may have side effects and returns a result.

Tools are the only MCP primitive where mutation is expected and permitted. The model autonomously decides when to call a tool, whether to call it at all, and what arguments to pass via the `inputSchema`.

NotebookLM

Engineering the Input Schema

A well-designed `inputSchema` is a prompt. It directly shapes the quality of model-generated tool calls.

Three Principles for Quality:

1. The description is a prompt. Vague descriptions produce vague calls.
2. Separate the minimum viable set into `required`. Provide defaults for optionals.
3. Use `enum` constraints to define fixed sets of valid values.

Code Diagnostic Panel

Flawed Code

```
{
  "name": "send_slack",
  "description": "Send a message",
  "properties": {
    "channel": { "type": "string" },
    "urgent": { "type": "boolean" }
  }
}
```

Vague description gives no trigger signal

Missing required fields array

No format hints

Optimized Code

```
{
  "name": "send_slack",
  "description": "Trigger to send a Slack msg
  to a specific C-level channel ID.",
  "required": ["channel", "message"],
  "properties": {
    "channel": { "type": "string",
      "enum": ["C123", "C456"] },
    "urgent": {
      "type": "boolean",
      "default": false }
  }
}
```

✓ *Clear usage trigger in description*

✓ *Strict required array*

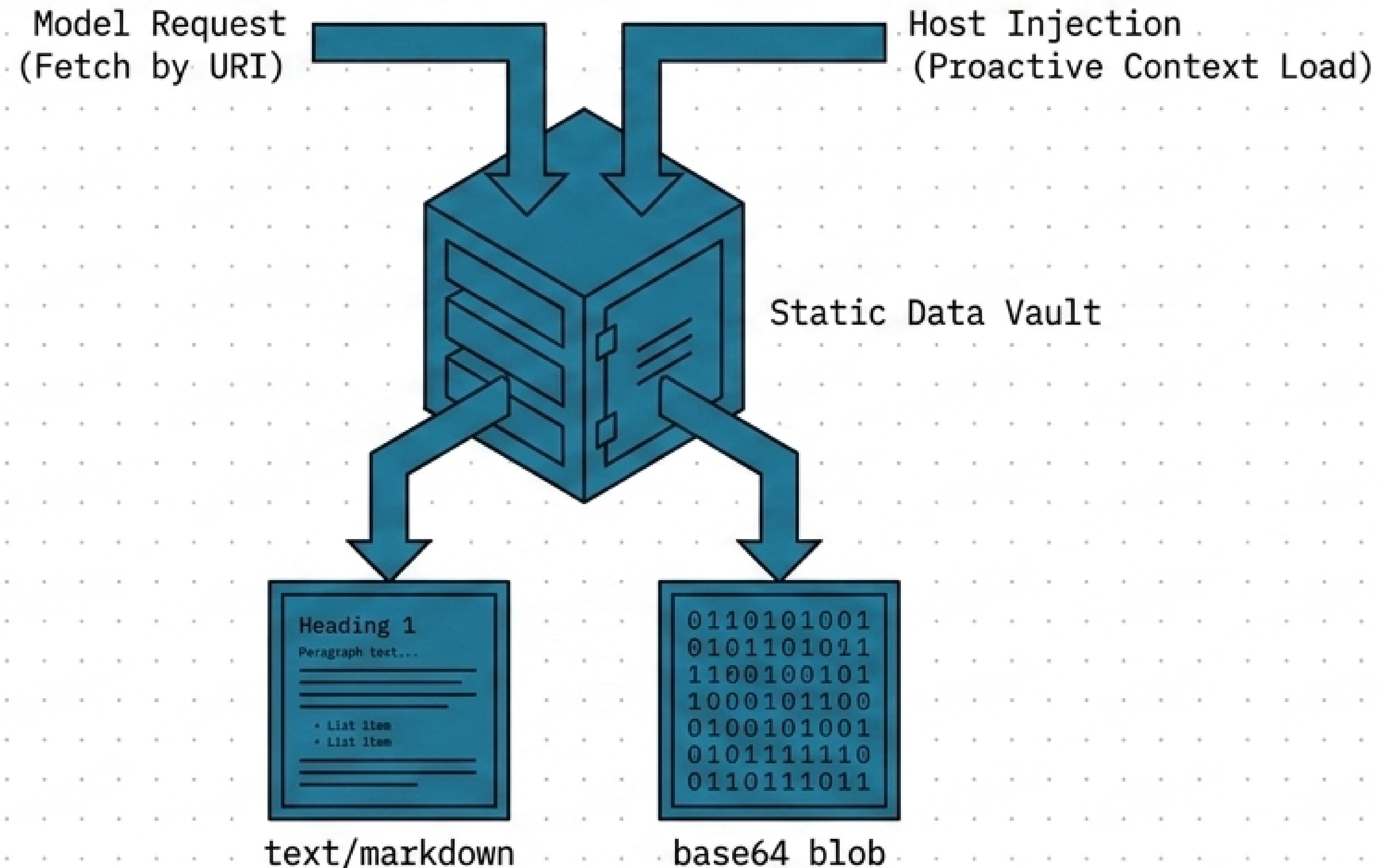
✓ *Enum constraints reduce hallucination*

NotebookLM

Resources: What the Model Reads

A Resource is read-only data identified by a URI that either the model or the application can inject into context.

The host application can make an editorial decision to load content—like `file:///Users/alice/README.md`—before the model even asks.

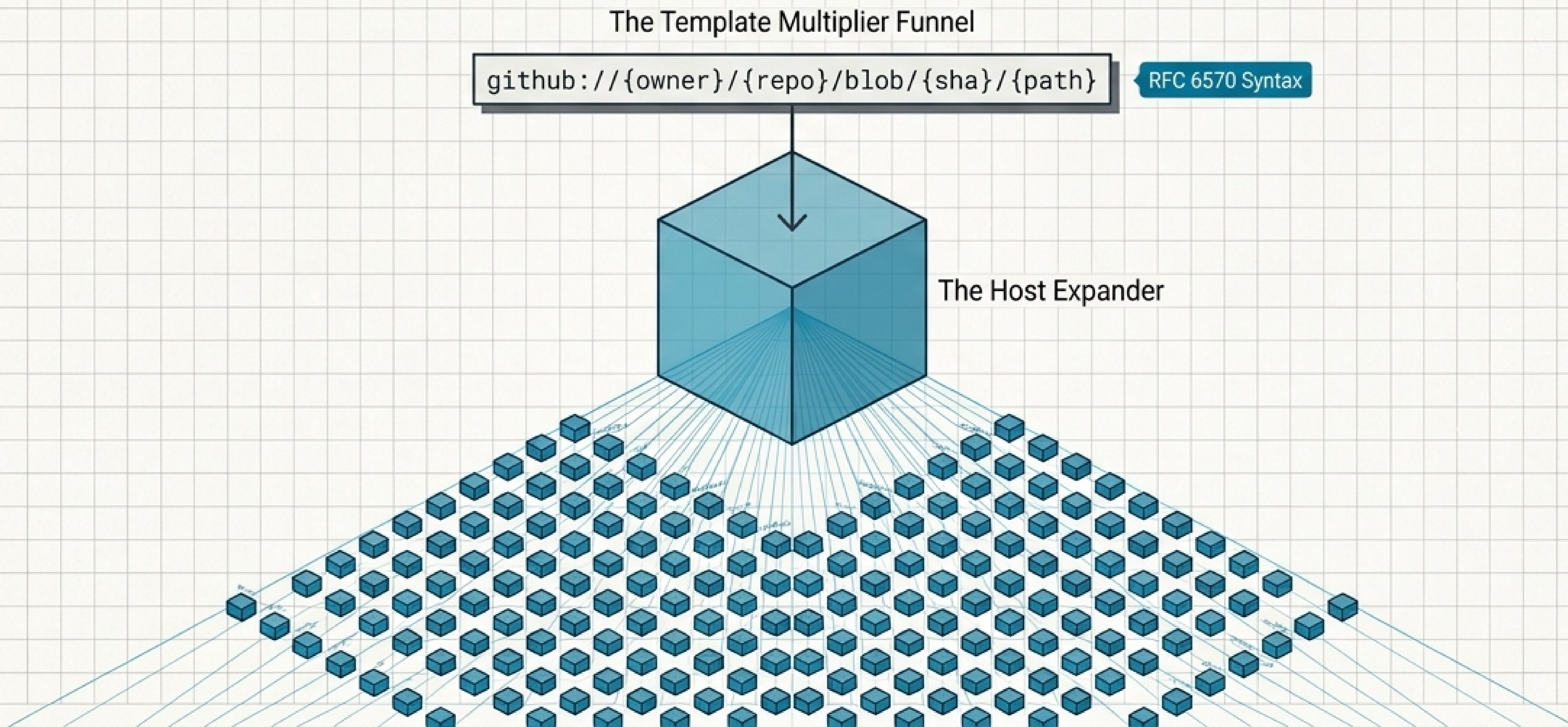


NotebookLM

URI Templating at Scale

Static URIs fail for dynamic systems like S3 buckets or repositories. MCP supports resource templates using RFC 6570 syntax.

A single resource template can represent billions of concrete resources—every file, in every repo, at every branch—expanding dynamically when requested.

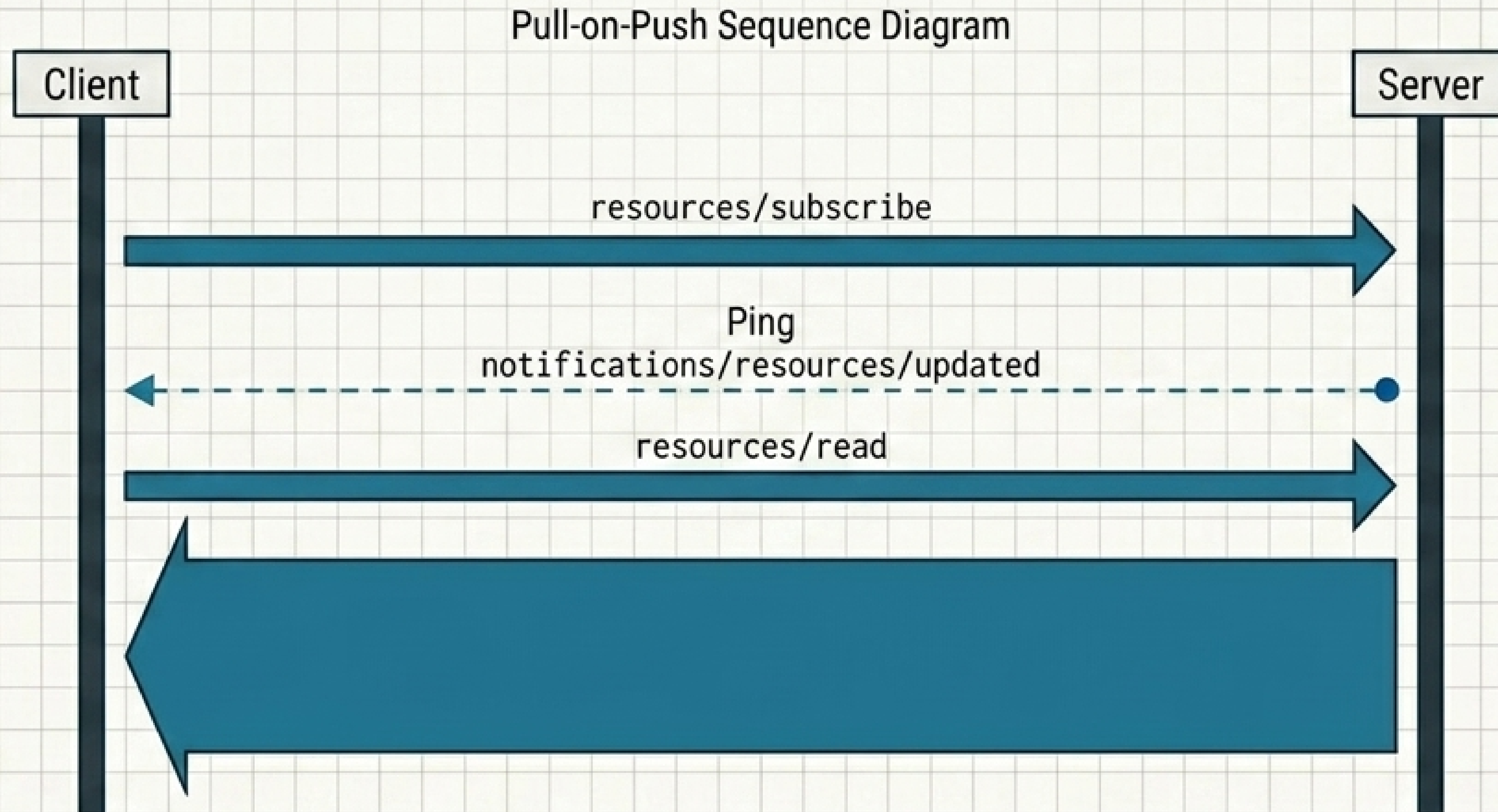


NotebookLM

Resource Subscriptions

For live configurations or real-time metrics, resources support active subscriptions.

The Push-on-Pull Pattern: Notifications tell the client something changed; the client fetches the new content itself. The server never pushes large payloads proactively.

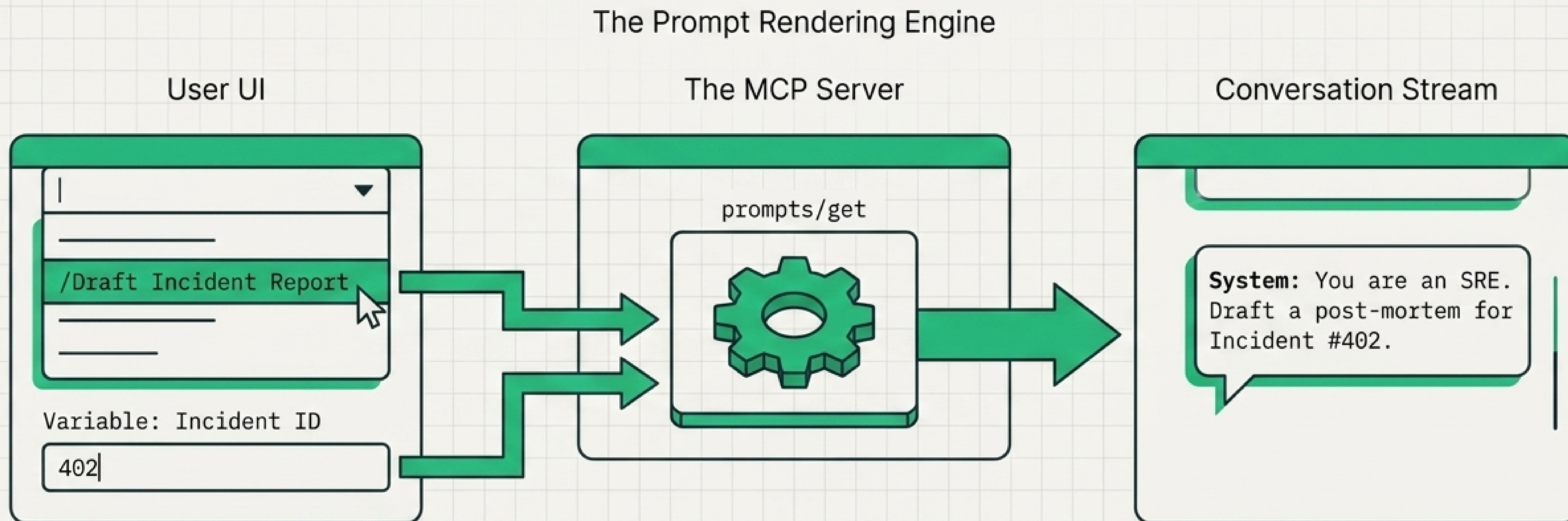


NotebookLM

Prompts: What the User Selects

A Prompt is a parameterised message template that the host exposes as a selectable option in its UI.

Often skipped by developers, Prompts have a structural advantage over static system instructions: they are discoverable, named, and parameterised at runtime by the user.



NotebookLM

The Diagnostic Matrix

Understanding the three primitives at the level of their design intent—not just their API shape—is the difference between an MCP server that works and one that works well.

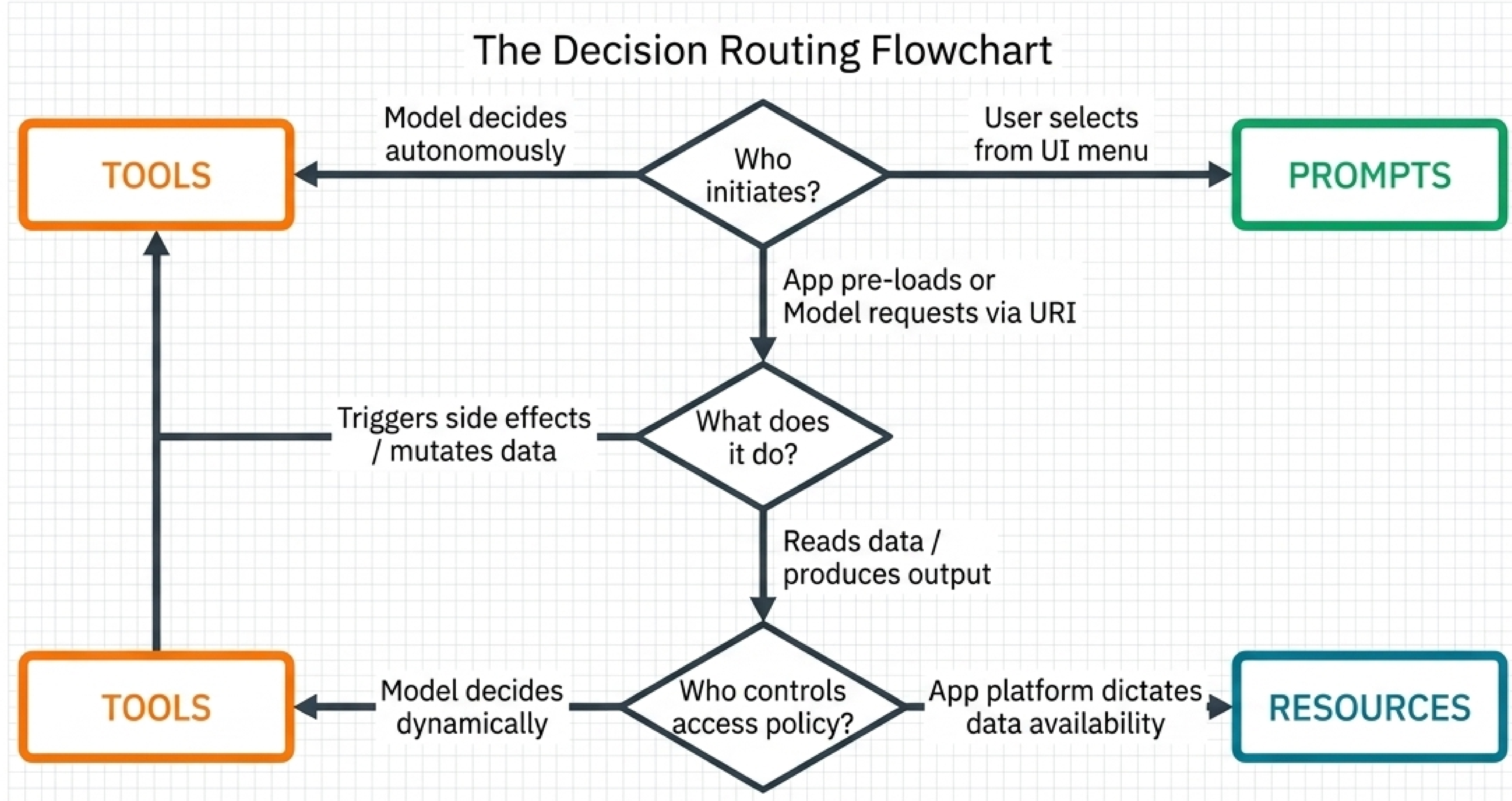
The Definitive MCP Primitive Matrix

	TOOLS	RESOURCES	PROMPTS
Initiated By	Model	App & Model	User
Side Effects Permitted	Yes	No	No
Wire Protocol	`tools/list & call`	`resources/list & read`	`prompts/list & get`
The Trap (Misuse)	Using a 0-argument Tool instead of a Resource.	Trying to push data instead of Pull-on-Push.	Hiding repeatable user tasks in static system prompts.

NotebookLM

The Decision Rule

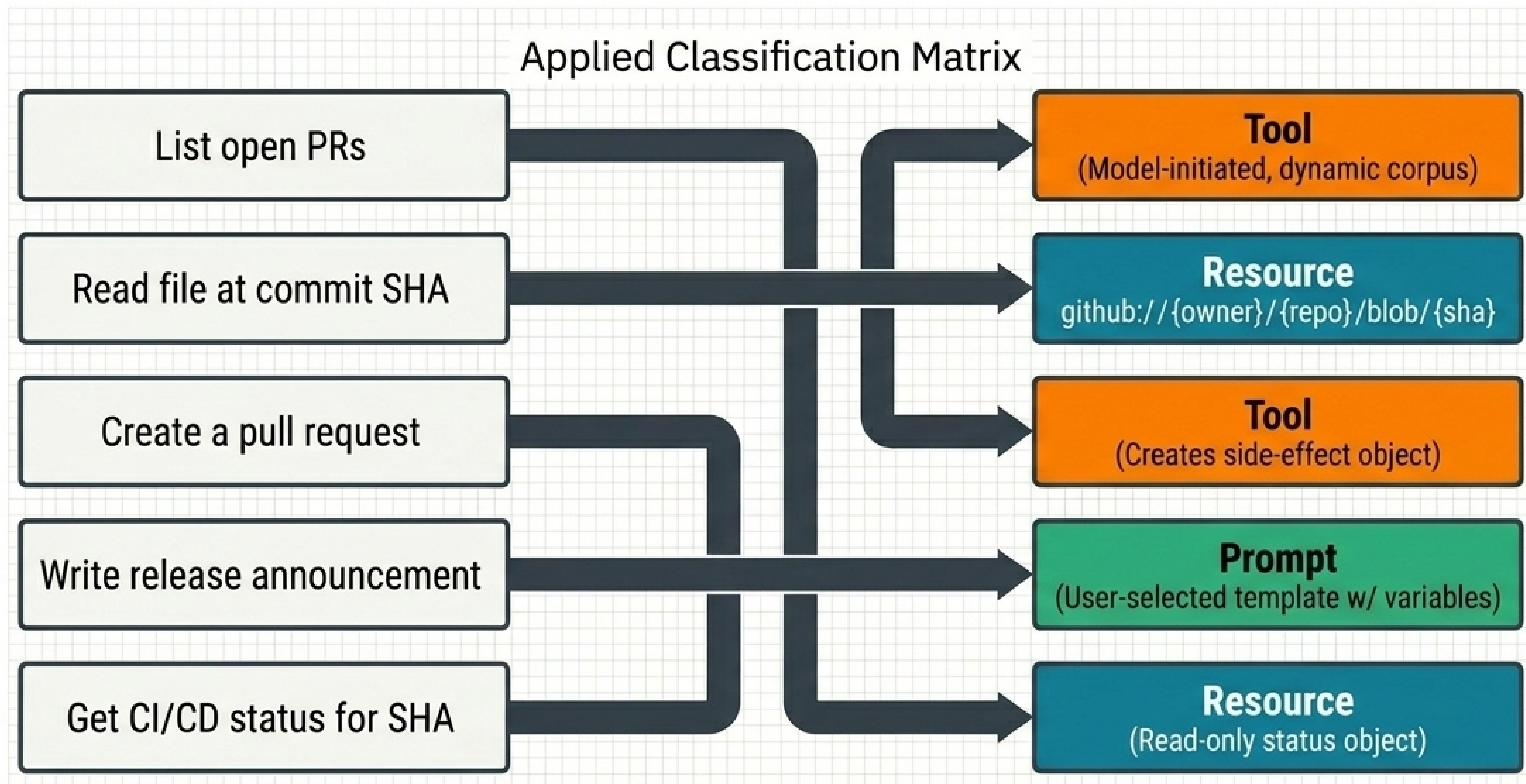
Every integration requirement can be classified using a strict three-question test. Follow the routing logic to arrive at the architecturally correct primitive.



NotebookLM

Applied Architecture: The GitHub Integration

Evaluating raw requirements through the decision rule matrix prevents systemic design flaws before a single line of code is written.

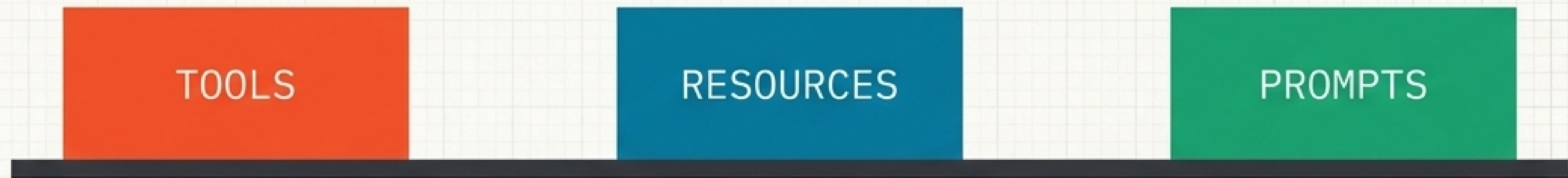


NotebookLM

Design Intent Over API Shape

A zero-argument **Tool** is just a **Resource** in disguise. An overloaded system prompt is just a missing **Prompt** template.

When you align your integration requirements with the true design intent of the Model Context Protocol primitives, you unlock lower latency, optimized token usage, and scalable architecture.



Auth: DPoP

Bearer OAuth 2.1

NotebookLM