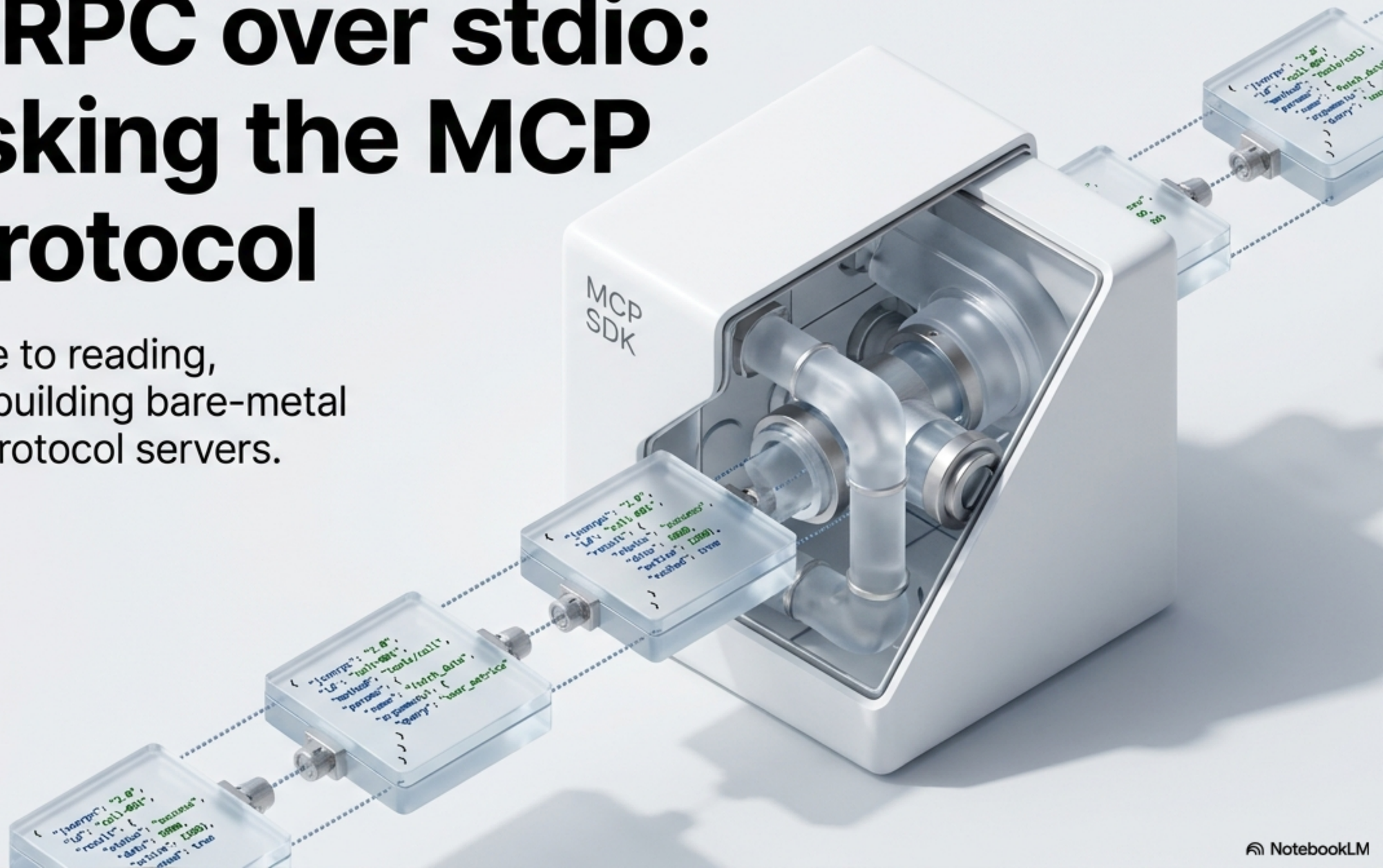
JSON-RPC over stdio: Unmasking the MCP Wire Protocol

A definitive guide to reading, debugging, and building bare-metal Model Context Protocol servers.



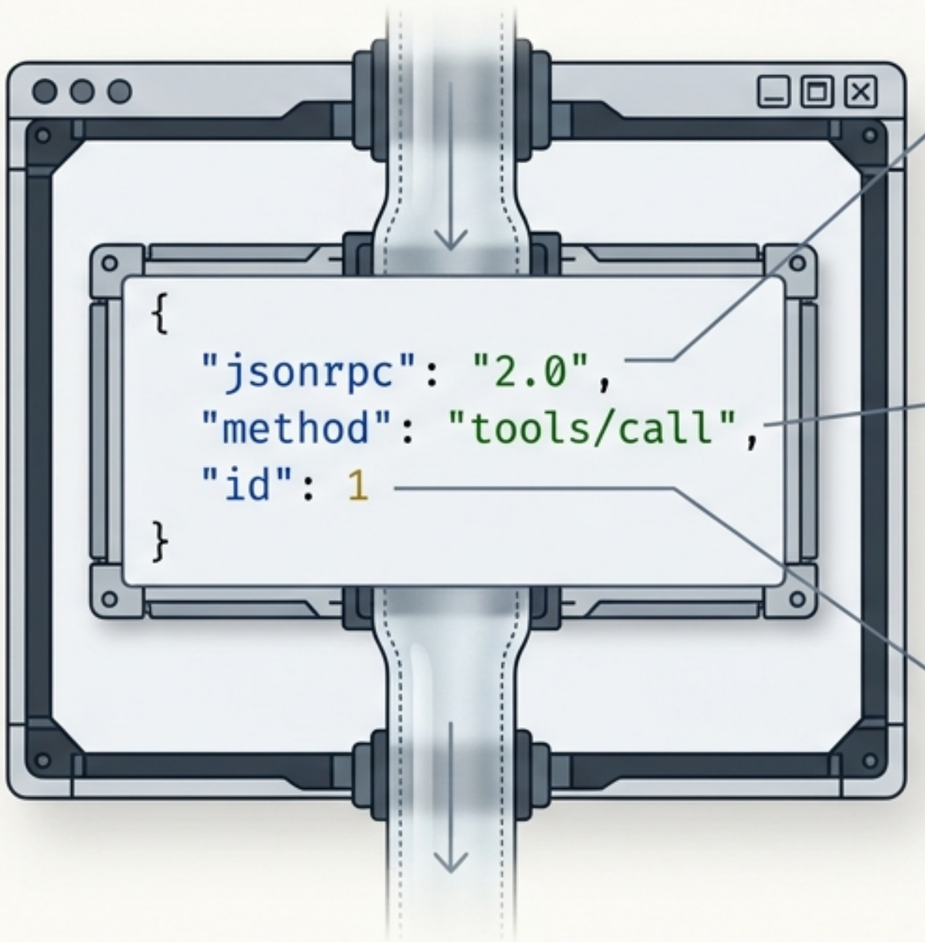
The Abstraction Illusion

Every MCP SDK is an abstraction layer over a raw wire protocol. When production integrations fail, abstractions disappear. Fluency in the raw wire format separates configuration mechanics from true integration engineers.

The Abstraction

```
mcp.run()
```

The Wire Reality



Performance (Round-Trips)

Directly dictates the frequency and latency of network requests, bypassing abstraction overhead. Critical for high-throughput applications.

Error Handling (Structured vs. Malformed)

Pinpoints exact failure points within the method invocation, distinguishing between client-side malformed structures and server-side failures.

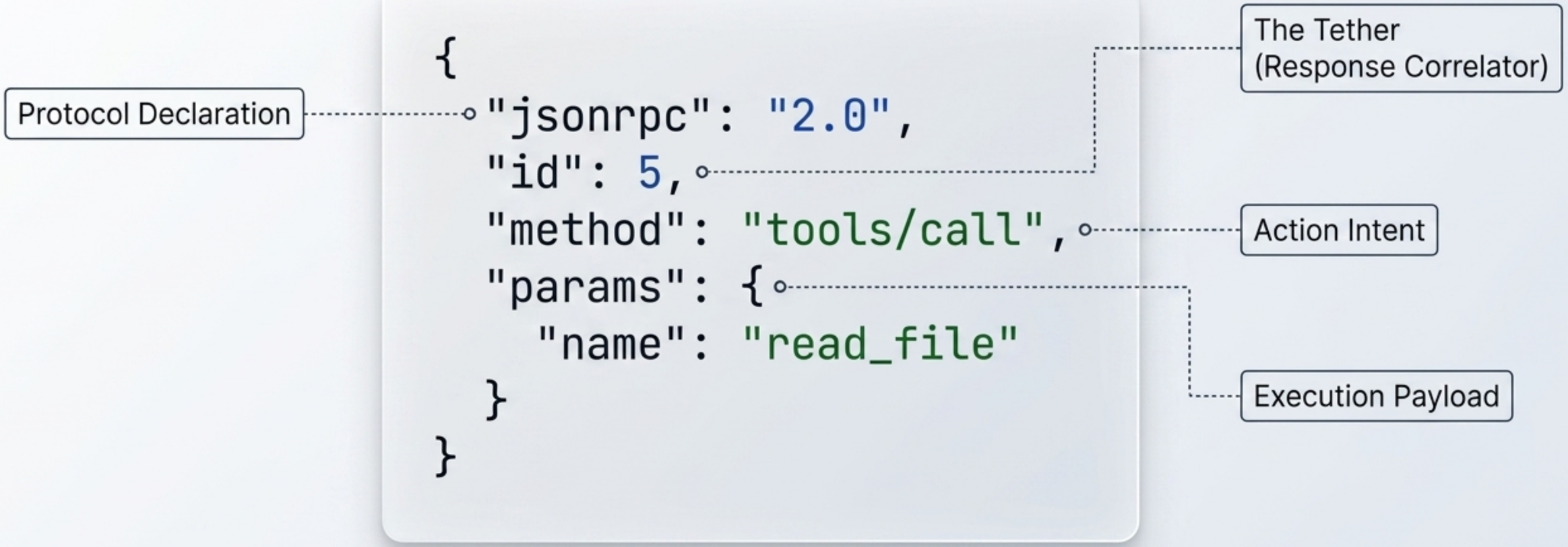
Security (Payload Visibility)

Reveals raw data and identifiers exposed on the wire, highlighting potential vulnerabilities and compliance risks masked by high-level calls.

The Universal Envelope: JSON-RPC 2.0

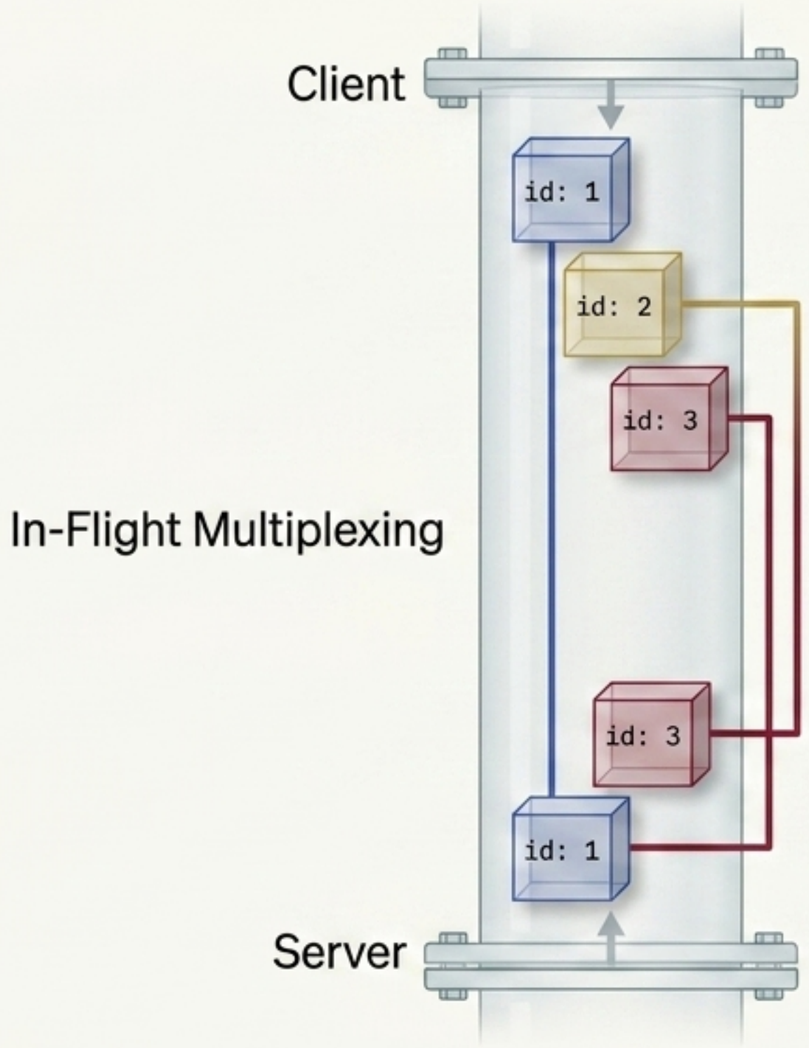
MCP deliberately adopts JSON-RPC 2.0 to wrap its messages. This shape acts as the universal envelope, standardizing how every system communicates before specific tools or prompts are even evaluated.

Anatomy Blueprint



The Tether: Correlating Requests and Responses

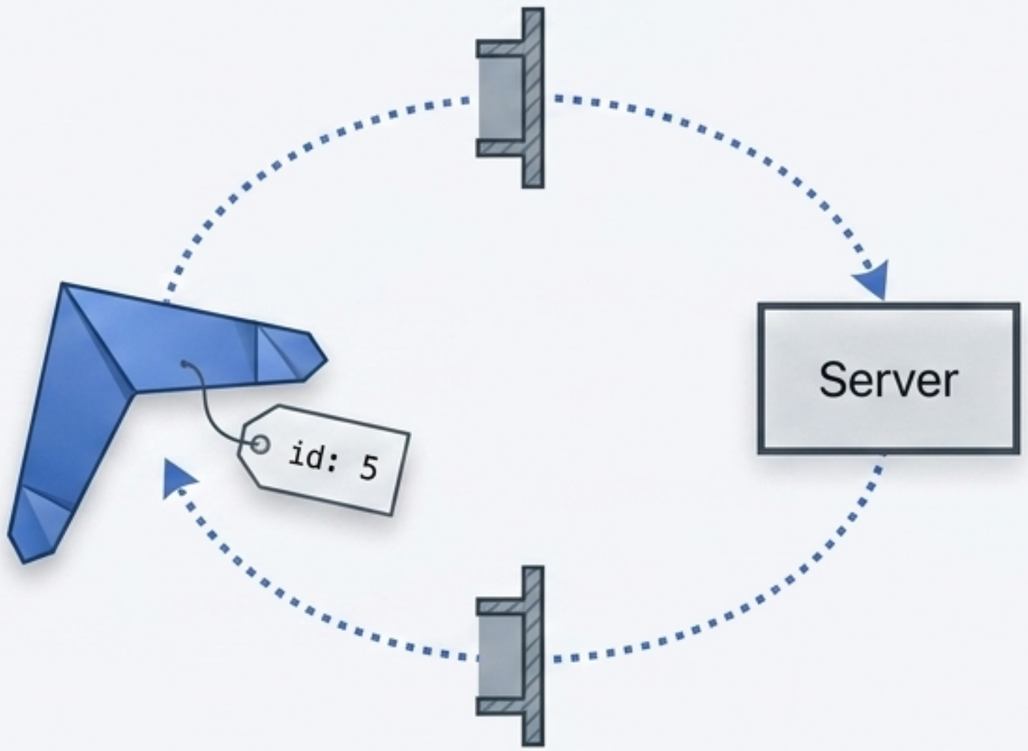
The client does not wait for one response before sending the next. Multiple requests can be in-flight simultaneously. The “id” field is the tether that matches an incoming server response to its original client request.



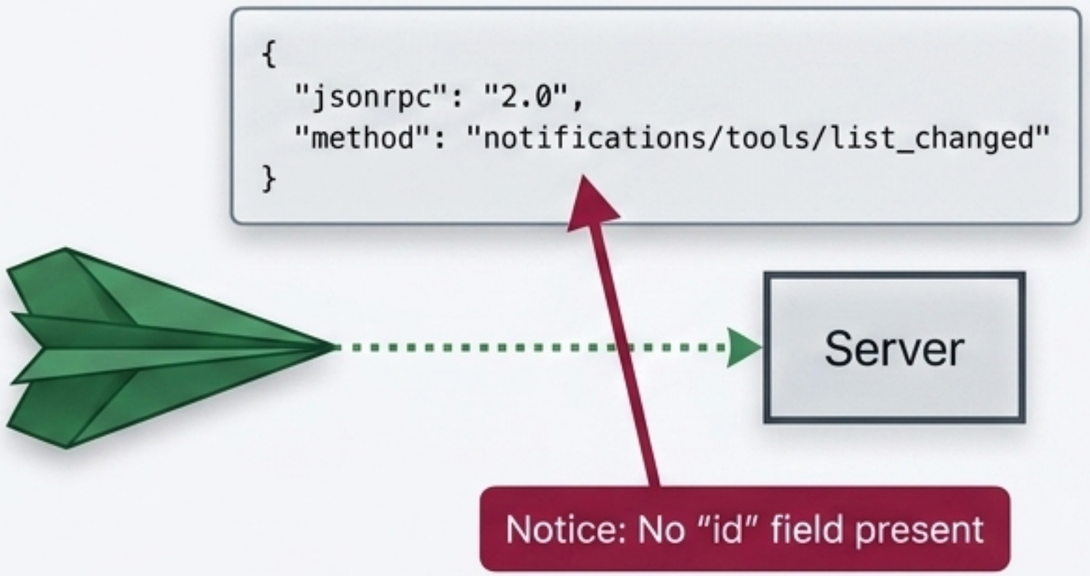
The ‘Fire and Forget’ Primitive

Not all messages expect a reply. Notifications are one-way dispatches used for unsolicited events. By omitting the “id” field, the protocol explicitly signals: do not respond.

Request

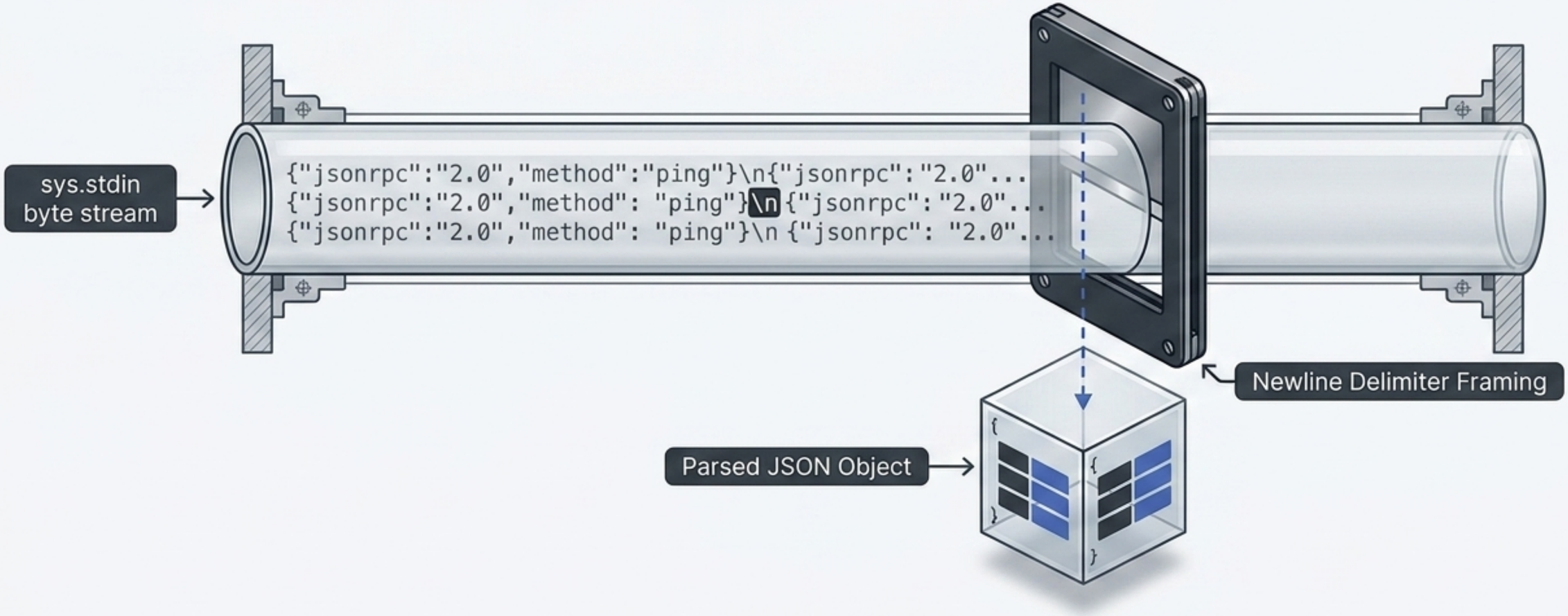


Notification



The Local Carrier: Stdio & NDJSON

For local integration, the host forks the MCP server as a subprocess. Communication flows directly through standard input (**sys.stdin**) and standard output (**sys.stdout**). The framing rule is brutally simple: read bytes until a newline character (**\n**), parse as JSON, process.



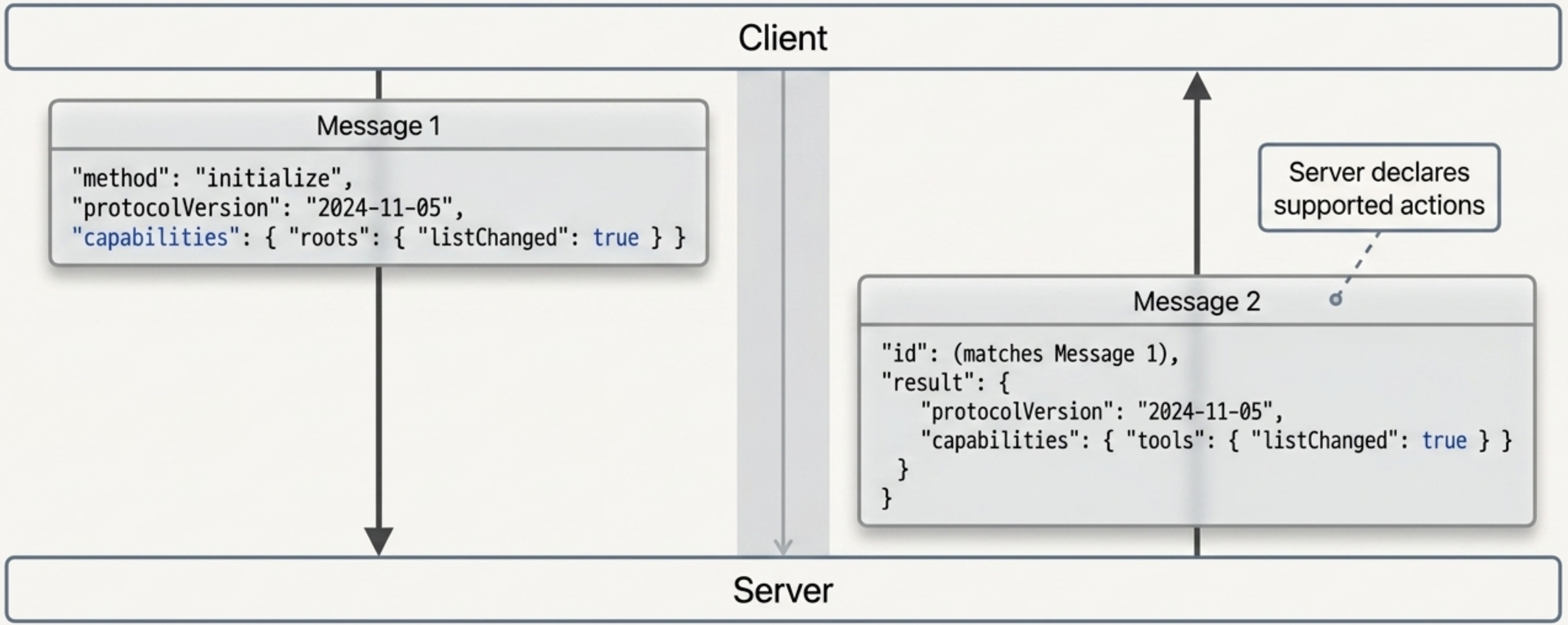
Architectural Justification: Why Stdio Wins Locally

MCP optimizes for the N×M local desktop problem. Evaluating alternative transports reveals the deliberate trade-offs made to minimize operational operational overhead and tooling weight.

Transport	Network Stack Dependency	Lifecycle Complexity	Format Opacity	Bidirectionality
WebSockets	Fails: Requires port & TLS	Fails: Complex host coordination	Passes: Plaintext JSON	Passes: Native full duplex
gRPC	Fails: Requires network port	Fails: Separate daemon needed	Fails: Binary (Protobuf)	Passes: Native streaming
REST	Fails: Requires HTTP server	Fails: Separate daemon needed	Passes: Plaintext JSON	Fails: Requires polling/webhooks
Stdio (MCP Local)	Wins: Zero network dependency	Wins: Implicit subprocess	Wins: Plaintext JSON	Wins: Free duplexing via pipes

The Initialize Handshake (Step 1 & 2)

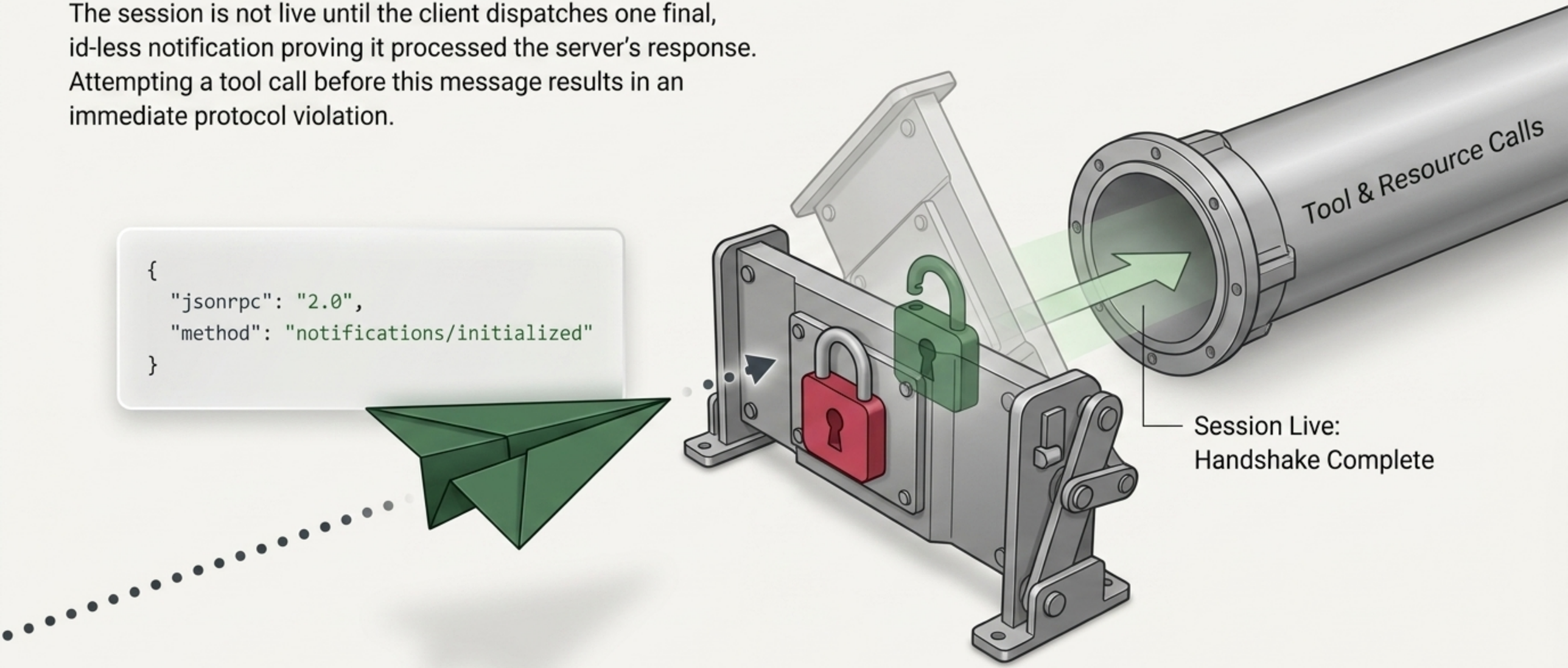
Every session starts with an exchange of capabilities. The client dictates the `protocolVersion`, while both sides declare what they can do. Un-declared capabilities are unsupported.



The Final Lock-In: `notifications/initialized`

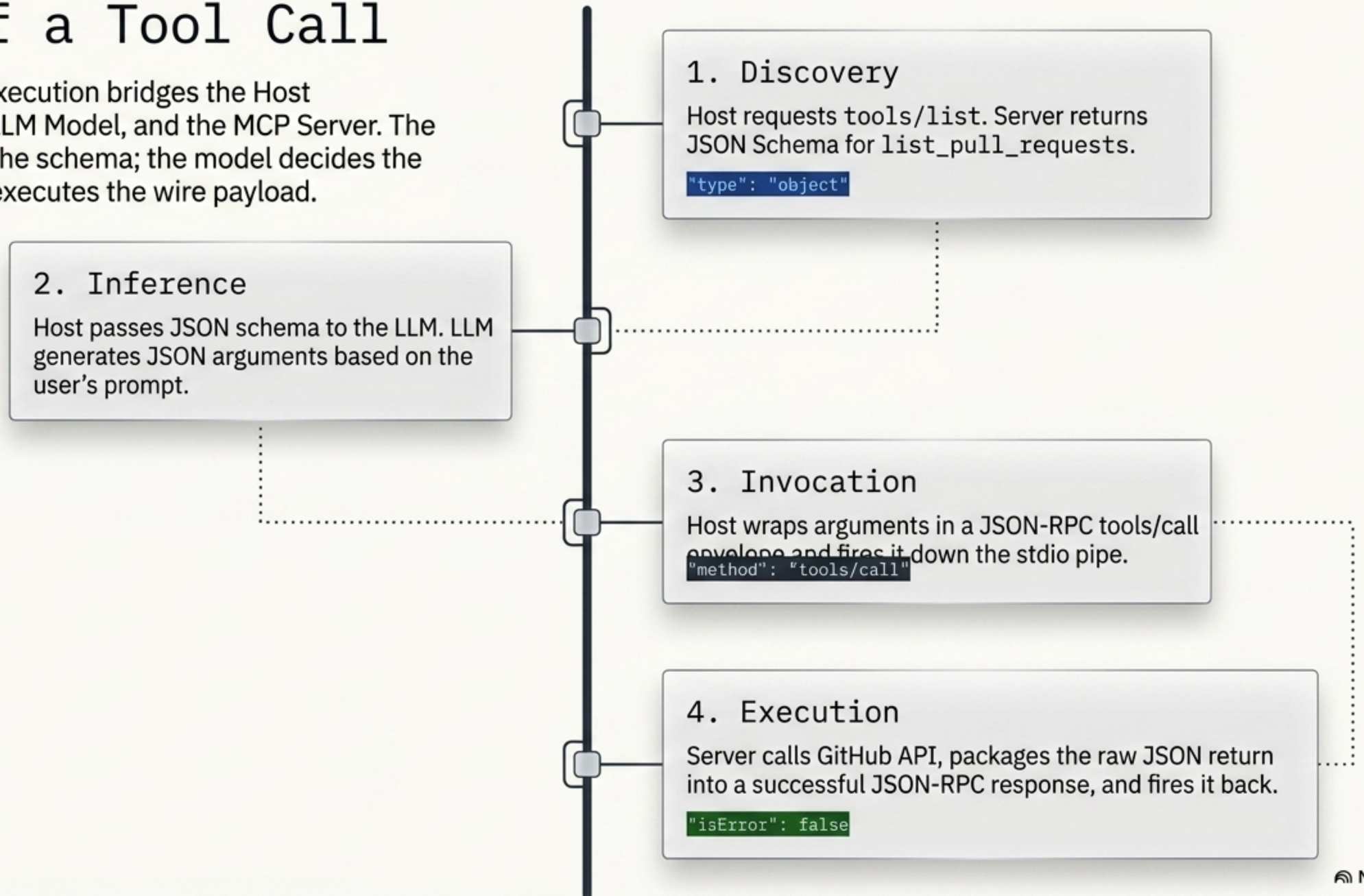
The session is not live until the client dispatches one final, id-less notification proving it processed the server's response. Attempting a tool call before this message results in an immediate protocol violation.

```
{
  "jsonrpc": "2.0",
  "method": "notifications/initialized"
}
```



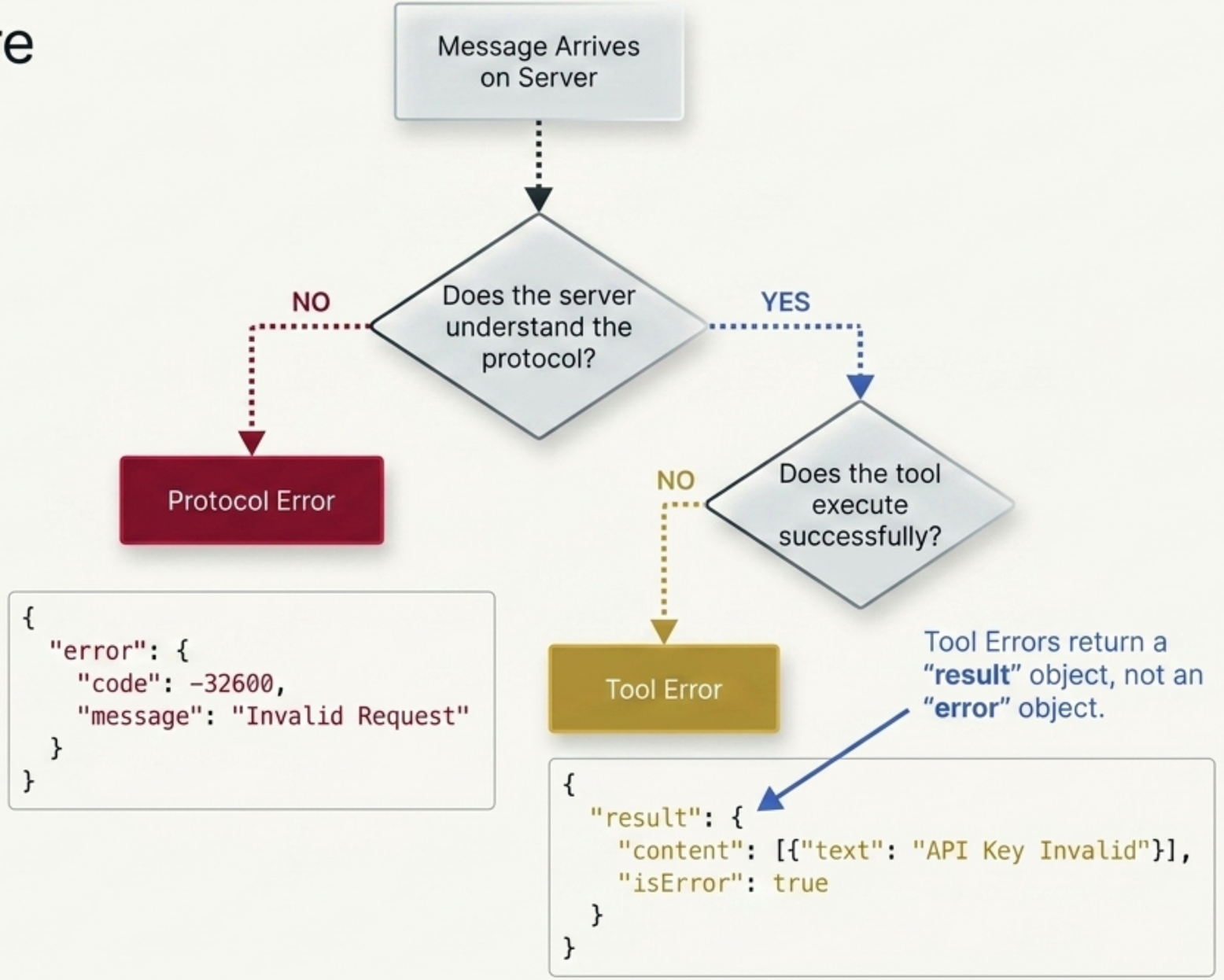
Life of a Tool Call

An end-to-end execution bridges the Host application, the LLM Model, and the MCP Server. The server provides the schema; the model decides the inputs; the host executes the wire payload.



The Two Paths of Failure

MCP separates transport/parsing failures from business logic failures. A protocol error means the server couldn't understand the request. A tool error means the server understood it perfectly, but the execution failed.



Protocol Error Reference Dictionary

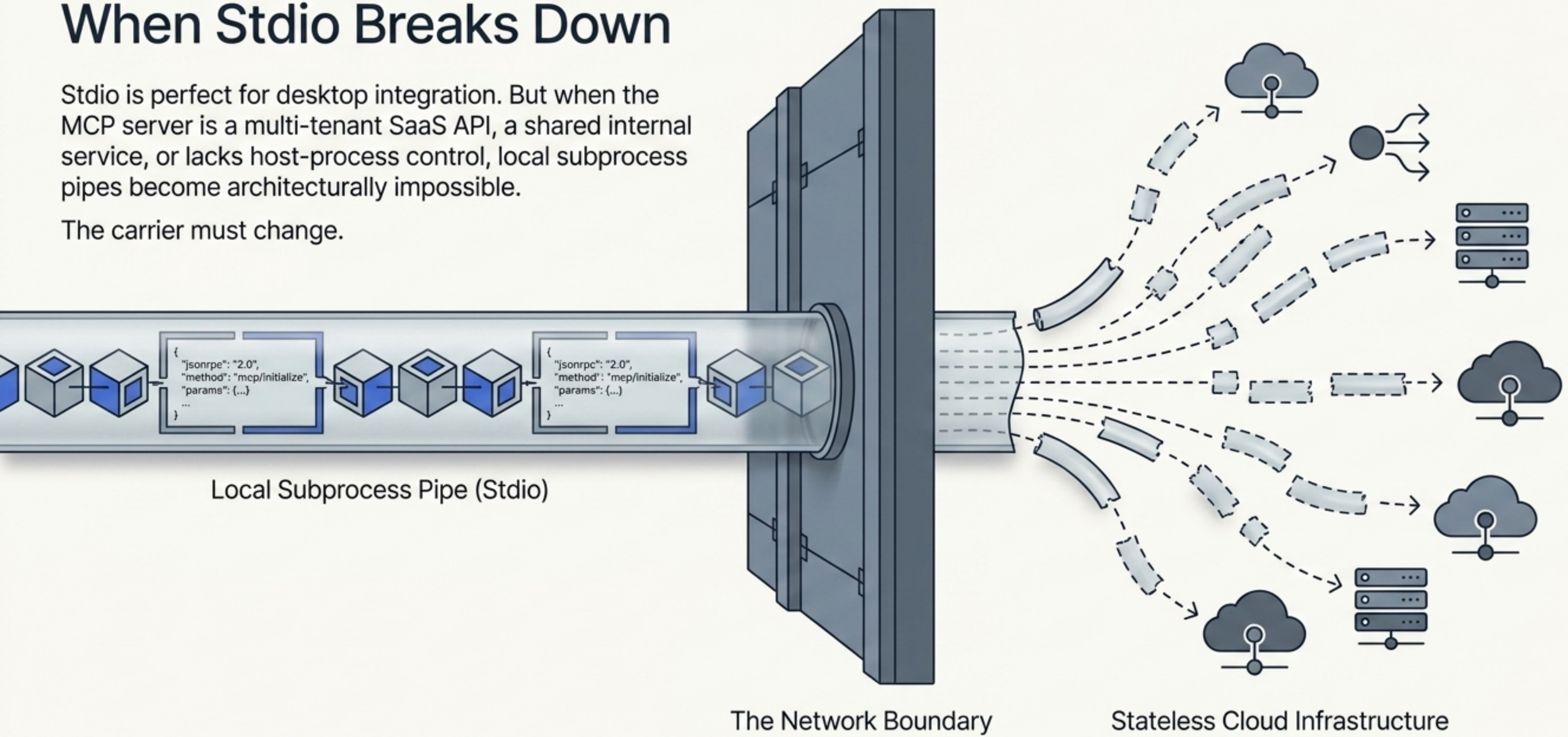
Standard JSON-RPC 2.0 error codes act as the diagnostic foundation for protocol-level failures. MCP-specific errors start at -32000 and go down. Domain/tool errors must never use these codes.

JSON-RPC Code	Diagnostic Meaning
`-32700 Parse error`	Invalid JSON received. Host sent malformed garbage bytes.
`-32600 Invalid Request`	Valid JSON parsed, but it does not match the JSON-RPC envelope structure.
`-32601 Method not found`	Host requested an unknown protocol action or unlisted tool.
`-32602 Invalid params`	Method exists, but parameters violate the defined JSON schema constraints.
`-32603 Internal error`	Uncaught server-side bug during processing (e.g., Python exception).

The Carrier Boundary: When Stdio Breaks Down

Stdio is perfect for desktop integration. But when the MCP server is a multi-tenant SaaS API, a shared internal service, or lacks host-process control, local subprocess pipes become architecturally impossible.

The carrier must change.



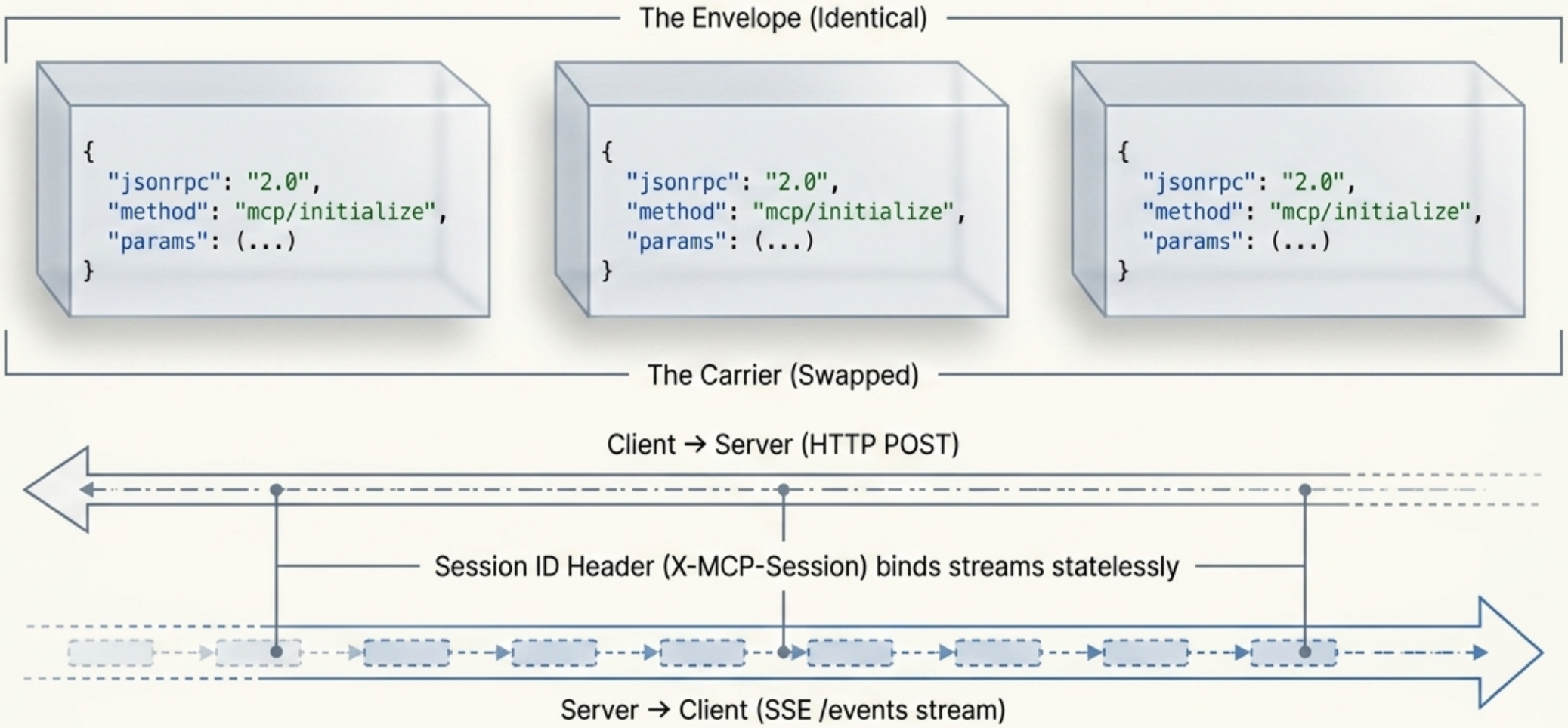
Local Subprocess Pipe (Stdio)

The Network Boundary

Stateless Cloud Infrastructure

HTTP Streaming Transport

For remote servers, the client swaps stdio for HTTP. Requests become POSTs. Server-push notifications require Server-Sent Events (SSE). Crucially, the wire format—the newline-delimited JSON-RPC envelope—remains identical.



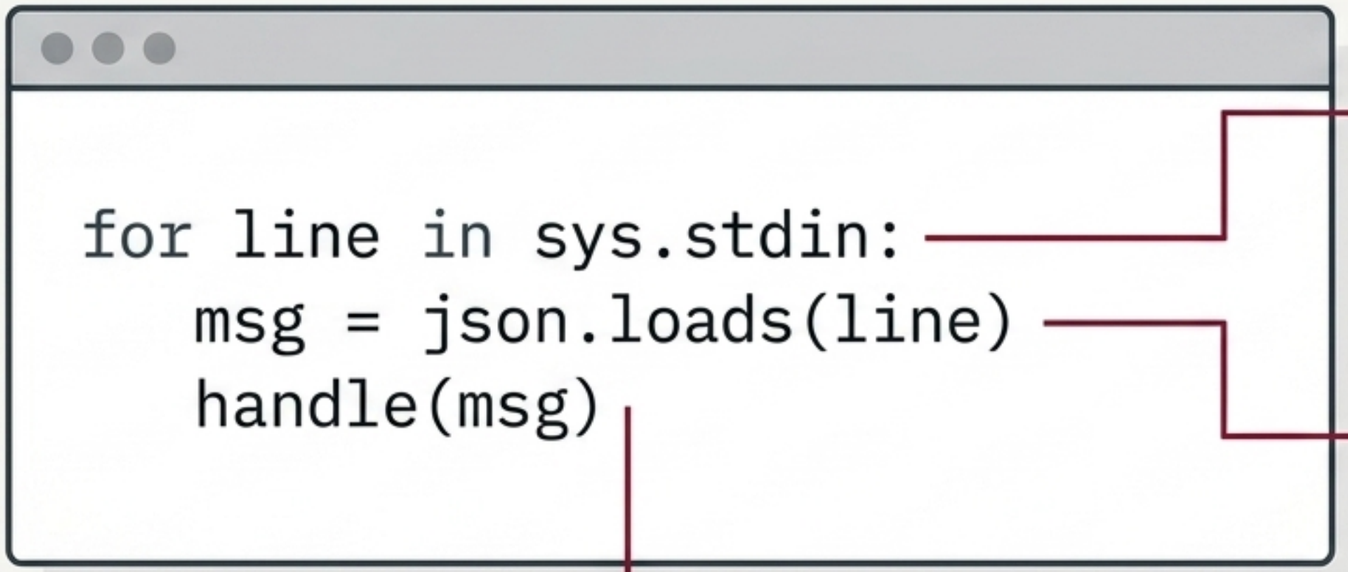
Transport Selection Diagnostic

The choice between Stdio and HTTP Streaming dictates your security, session, and deployment models. Choose the transport that matches your operational reality.

Dimension	Stdio (Local)	HTTP Streaming (Remote)
Session Management	⚓ Implicit (1 pipe = 1 session)	✔ Explicit (Requires Session ID headers)
Server-Push Mechanics	✔ Native full-duplex (write to stdout anytime)	⚓ Requires dedicated SSE endpoint or chunked body
Authentication	⚓ None (relies on OS process isolation)	✔ Requires HTTP Headers (Bearer tokens, DPoP)
Ideal Use Cases	⚓ IDE plugins, local dev tools, CI/CD actions	✔ SaaS endpoints, multi-tenant microservices

Building Bare-Metal: The 60-Line Server

By discarding the SDK, the underlying mechanics are exposed. A server is simply a continuous loop waiting for `stdin`, parsing JSON, routing the method, and printing a JSON string back to `stdout`.



Missing: Handle EOF shutdown cleanly. Production servers must trap the loop exit and close gracefully.

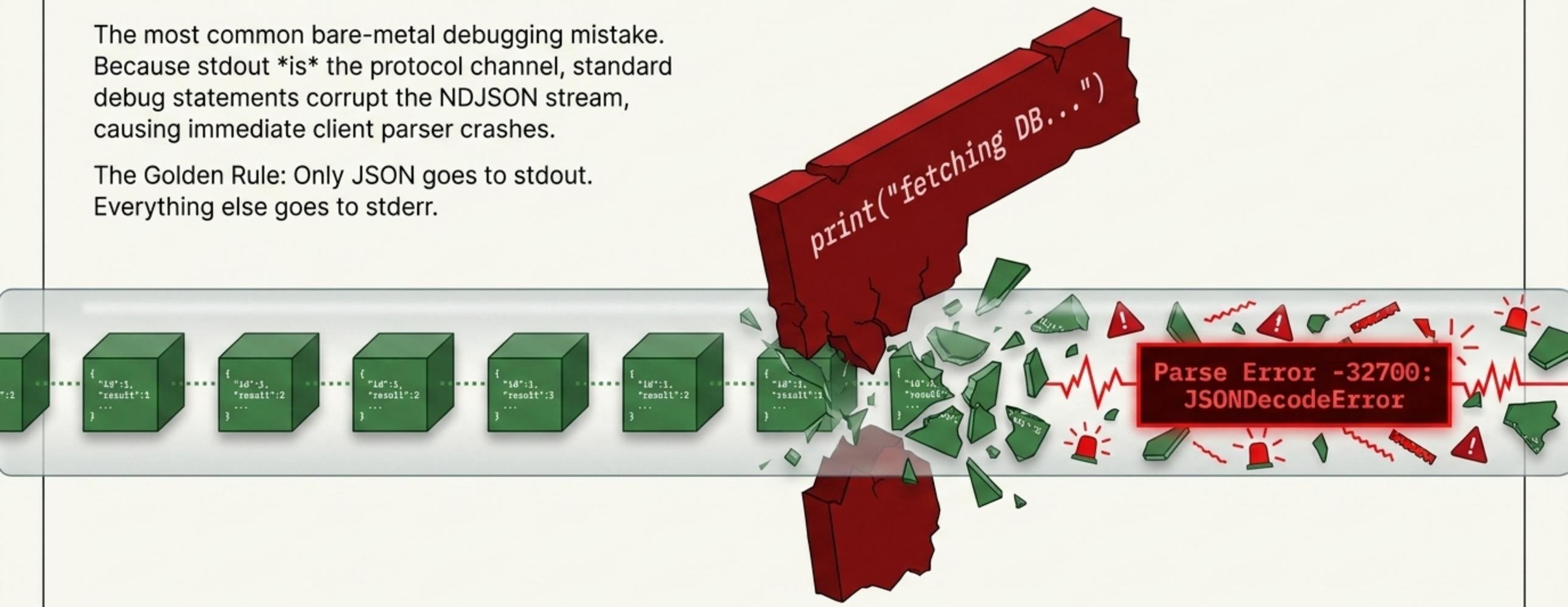
Missing: Validate jsonrpc='2.0' envelope structure. Never trust raw loads().

Missing: State Machine constraint. Server must verify **'notifications/initialized'** state before accepting any tool call requests.

The Deadliest Trap: **Print()**

The most common bare-metal debugging mistake. Because stdout *is* the protocol channel, standard debug statements corrupt the NDJSON stream, causing immediate client parser crashes.

The Golden Rule: Only JSON goes to stdout. Everything else goes to stderr.



The MCP Architecture Stack

You now possess full visibility into the engine. You understand how the Carrier delivers the Envelope, and how the Envelope manages state and failure. Next, we move from the Wire to the Primitives: filling these envelopes with Tools, Resources, and Prompts.