ChatGPT

CLI Communication Patterns: Analysis and Recommended Solution

Overview of Sources and Findings

We surveyed the repositories and documentation provided by the user to identify available patterns for tool-to-tool communication in CLI orchestration systems. The main sources were:

1. **Aider (Aider-AI/aider)** – The scripting documentation describes how to send one-shot commands to Aider using the `--message` flag or via Python API. These options support injecting prompts and loading session files but do not implement any cross-tool IPC on their own.
2. **PowerShell MCP (PowerShell Gallery)** – The `PowerShell.MCP` module exposes a Model Context Protocol server for PowerShell that communicates over **local named pipes**, offering enterprise-grade security and eliminating network exposure. The package notes highlight universal compatibility with Claude Desktop and any MCP client and emphasize "Local-only named pipe communication, no network exposure" [1].
3. **baby2cli (DICKY1987/baby2cli)** – This repository contains a full toolbox of IPC mechanisms designed for CLI orchestration. The `ipc/` directory includes:
4. **http_bus/** – a loopback HTTP bus with publish/subscribe endpoints for tool coordination (optional token authentication) [2].
5. **named_pipes/** – a Windows named-pipe server and client; the client uses `multiprocessing.connection.Client` to send JSON records through a pipe [3]. The server logs messages and responds to control commands like `stop` [4].
6. **stdio_bridge.py** – a JSON-RPC over stdio bridge. The `JsonStdIoClient` sends a JSON payload on stdin, starts a subprocess, and waits for a JSON response from stdout, returning a dictionary or an error [5].
7. **process_runner.py** and **secure_channel.py** – helper modules for launching subprocesses and adding optional HMAC signatures to messages.
8. **SHOE_STRING_CLI** – This repo contains general performance and orchestration improvements but doesn't introduce new IPC patterns for CLI communication; its content is mostly about optimizing existing scripts.

File Tree Diagram for Relevant Sources

```
baby2cli/ipc/
├── http_bus/
│   ├── server.py             # Loopback HTTP bus for tool coordination;
provides /publish and /last endpoints [2].
│   └── client scripts        # (not shown) Example clients for posting to the
HTTP bus.
├── named_pipes/
```

```
│   ├── client.py              # Python client that connects to a Windows named
pipe, sends JSON payloads, and prints acknowledgements [3].
│   ├── server.py              # Named-pipe server that accepts JSON messages,
writes logs, and handles a stop control message [4].
│   ├── client_demo.ps1        # PowerShell script demonstrating how to send
messages via named pipes (not opened).
│   └── server_demo.ps1        # PowerShell script demonstrating how to run the
named-pipe server (not opened).
├── process_runner.py          # Utility to run subprocesses and stream their
output; used by IPC bridges.
├── secure_channel.py          # Adds optional HMAC signatures to JSON messages
for message authentication.
└── stdio_bridge.py            # JSON-RPC over stdio bridge; sends JSON via
stdin to a subprocess and expects a JSON object on stdout [5].

PowerShell Gallery – PowerShell.MCP module
└── (module files not listed) – offers an MCP server that uses Windows named
pipes for local-only communication [1].
```

File List with 1-Line Descriptions

File/Module	1-Line Description
baby2cli/ipc/http_bus/server.py	Implements a loopback HTTP server with `/publish` and `/last` endpoints for tool coordination, supporting optional token authentication [2].
baby2cli/ipc/named_pipes/client.py	Demonstrates how to send JSON log or control messages over a Windows named pipe using `multiprocessing.connection.Client` [3].
baby2cli/ipc/named_pipes/server.py	A named-pipe server that logs received messages and acknowledges clients; also handles a `stop` control operation [4].
baby2cli/ipc/stdio_bridge.py	Provides a simple `JsonStdIoClient` class to send JSON requests to a subprocess via stdin and parse a JSON response from stdout [5].
baby2cli/ipc/process_runner.py	Helper for spawning subprocesses and capturing their output streams.
baby2cli/ipc/secure_channel.py	Adds HMAC signatures to JSON messages for secure IPC over untrusted channels.
PowerShell.MCP module	A PowerShell module that exposes a Model Context Protocol server communicating via local named pipes; emphasizes enterprise security and compatibility with Claude Desktop [1].

File/Module	1-Line Description
Aider's `--message` CLI option	Allows one-shot prompt injection or message file loading but does not provide cross-tool IPC by itself (documentation highlights this feature).

Evaluation of Available Solutions

The repositories present three main patterns for CLI communication:

1. **JSON-RPC over stdio** – `stdio_bridge.py` shows how to send a JSON request on stdin and receive a JSON response from stdout. This pattern is extremely portable, works on Windows and Unix, and integrates well with LLM tools or deterministic AI scripts. The simplicity of `JsonStdIoClient` (single request in, single response out) makes it ideal for one-shot tool invocations [5]. However, it doesn't inherently support streaming or multi-producer fan-in; each call spawns a new process.
2. **Named pipes** – The `named_pipes` server and client illustrate how to create a long-running log aggregator or controller on Windows. Named pipes support multiple producers feeding one consumer and can provide low-latency, bidirectional communication. The PowerShell.MCP module uses named pipes for secure local-only CLI integration; it is widely compatible and avoids network exposure [1]. The downside is platform specificity (AF_PIPE only on Windows) and the complexity of managing long-running servers and concurrency.
3. **Loopback HTTP bus** – The HTTP server offers a micro-service pattern with publish/subscribe endpoints [2]. While flexible and language agnostic, it violates the user's prohibition on HTTP communication.

Recommended Solution (Objective Improvement)

To meet the user's constraints (no HTTP, no TUI) and still provide robust tool-to-tool communication, the following solution is objectively better:

Use JSON-RPC over stdio for one-shot tool invocations, and Windows named pipes for long-running multi-producer scenarios. Combine this with deterministic git worktree isolation and a patch-first gate for safe integration.

Why this is better

- **Portability and Simplicity** – JSON-RPC over stdio works on any platform, requires no special OS features, and keeps tools decoupled. The `JsonStdIoClient` pattern from baby2cli is trivial to implement and ensures that each tool invocation returns a structured response [5].
- **Secure Local Channel** – For Windows environments, named pipes provide enterprise-grade security and efficient bidirectional communication. They support multiple producers sending messages to a long-running orchestrator or log sink, as illustrated by the `named_pipes` client and server [3] [4]. The PowerShell.MCP module leverages this pattern and is already battle-tested [1].
- **Deterministic Integration via Worktrees** – To avoid race conditions and ensure reproducibility, run each AI tool invocation in its own git worktree. After the tool produces a patch or a diff, use a **patch-first gate** in an isolated worktree to apply the diff, run auto-fixers and linters, and commit if green. This eliminates the fragility of shared folders and yields atomic commits.

- **No HTTP or TUI** – The proposed solution avoids loopback HTTP completely and uses only command-line communication; there is no need for interactive UIs.

Implementation Outline

1. **One-shot invocations** – Wrap each AI or helper tool in a JSON-RPC over stdio interface. When a higher-level script needs to call a tool (e.g., Aider or a PowerShell script), it constructs a JSON request (method, arguments, input text) and sends it through stdin; the tool returns a JSON response on stdout. This can be done directly using `stdio_bridge.py` or re-implemented in PowerShell.
2. **Named-pipe bus (optional)** – For scenarios where multiple producers (e.g., parallel workstreams) need to feed one orchestrator (e.g., for logging, scheduling, or cancellation), run a named-pipe server as shown in `named_pipes/server.py`. Producers use the corresponding client to send JSON messages [3] [4].
3. **Patch gate** – After receiving a diff from an AI tool, run a patch gate in a throwaway git worktree. Apply the patch with `git apply --index`, run auto-fixers (`ruff --fix`, `black`, `Invoke-Formatter`), and quick verifiers (`mypy`, `ruff`, `Invoke-ScriptAnalyzer`). If everything passes, commit the change in the worktree and merge into the main workstream. If not, reject the patch with a quality report. This ensures deterministic, safe integration.
4. **Audit** – Emit structured JSONL logs for every run (`.runs/<RUN_ID>/events.jsonl`), recording start/end times, commands, and outcomes. This makes post-mortem and troubleshooting straightforward.

By combining JSON-RPC over stdio for portability with Windows named pipes for more advanced scenarios, and enforcing deterministic git worktrees with a patch gate, the resulting solution is robust, secure, and adheres to the no-HTTP, no-TUI requirements.

---

[1] PowerShell Gallery | PowerShell.MCP 1.2.1
https://www.powershellgallery.com/packages/PowerShell.MCP/1.2.1

[2] raw.githubusercontent.com
https://raw.githubusercontent.com/DICKY1987/baby2cli/main/ipc/http_bus/server.py

[3] raw.githubusercontent.com
https://raw.githubusercontent.com/DICKY1987/baby2cli/main/ipc/named_pipes/client.py

[4] raw.githubusercontent.com
https://raw.githubusercontent.com/DICKY1987/baby2cli/main/ipc/named_pipes/server.py

[5] raw.githubusercontent.com
https://raw.githubusercontent.com/DICKY1987/baby2cli/main/ipc/stdio_bridge.py