reverse-engineering Warp's cli-agent notification protocol

Warp's agent sidebar runs on one OSC 777 escape sequence and a six-field JSON envelope. here's the full protocol, reverse-engineered from the three official adapter repos.

Yiğit Konur · Apr 21, 2026 · 20 min read

▶ The useful bits

the “Warp agent sidebar” isn’t magic. when Claude Code, Gemini CLI, or OpenCode finishes a turn, what actually happens is one escape sequence gets written to `/dev/tty` — a single OSC 777 call with a JSON body. Warp picks it up on the pane’s output stream, parses it, and routes it into the sidebar.

there’s no public spec for any of this. but Warp ships three open-source adapters — claude-code-warp, gemini-cli-warp, opencode-warp — and reading them side by side surfaces the whole protocol. six envelope fields, seven events, one transport primitive, one feature flag.

this post is a reverse-engineered field guide. everything here is observed behavior from commits `b8ad3cc`, `953e05b`, and `e60e068` of those three repos. if your agent can emit a tiny piece of JSON over the tty, Warp will pick it up too.

the 30-second version

agents notify Warp by writing **one OSC 777 escape sequence** directly to `/dev/tty`:

Skip to content

```
ESC ] 7 7 7 ; notify ; <TITLE> ; <BODY> BEL
```

as a raw string (`\x1b` = ESC, `\x07` = BEL):

```
\x1b]777;notify;warp://cli-agent;<JSON-body>\x07
```

- **TITLE** is the literal string `warp://cli-agent`. any other title produces a plain-text notification (the legacy path).
- **BODY** is a compact JSON object.

Warp parses every OSC 777 it sees arriving on a pane's tty. if the title matches `warp://cli-agent`, the body goes into the structured agent channel. if not, it's rendered as plain text in the notification center.

that's the whole transport. everything else is schema.

the transport

`OSC 777` is an old xterm operating-system command historically used for desktop notifications (gnome-terminal, urxvt). Warp co-opts it and adds a private title namespace for structured agent events.

emit from any language:

```
# POSIX shell
printf '\033]777;notify;%s;%s\007' "warp://cli-agent" "$JSON_BODY" > /dev/tty
```

```
// Node / Bun
import { writeFileSync } from "fs";
const seq = `\x1b]777;notify;warp://cli-agent;${body}\x07`;
writeFileSync("/dev/tty", seq);
```

Skip to content

```python
# Python
with open("/dev/tty", "w") as tty:
    tty.write(f"\x1b]777;notify;warp://cli-agent;{body}\x07")
```

```rust
// Rust
use std::fs::OpenOptions;
use std::io::Write;
let mut tty = OpenOptions::new().write(true).open("/dev/tty")?;
write!(tty, "\x1b]777;notify;warp://cli-agent;{}\x07", body)?;
```

why /dev/tty

stdout and stderr both fail for this.

hook scripts in agent hosts usually have **stdout captured** as a structured return channel — the host reads JSON back from stdout to decide whether to block, mutate, or log the event. writing an escape sequence to stdout would corrupt that JSON.

stderr is often piped to a log file. the user never sees it, and the escape never reaches the terminal.

`/dev/tty` is the controlling terminal of the process, bypassing any pipes. writes hit the pane exactly once. all three reference adapters write to `/dev/tty` and silently swallow any error (e.g. when there is no controlling terminal). you should too.

SSH works for free

OSC 777 travels over the wire like any other terminal byte. `ssh` into a machine from Warp, run an agent there, and notifications still fire — Warp sees the sequence when it arrives at the local pty. this is literally why the OpenCode adapter's `notify.ts` comment says "working over SSH": the transport *is* the terminal stream.

Skip to content

if Warp is **not** on the receiving end (e.g. you're tmux'd into a remote server and the outermost terminal is iTerm), nothing breaks. the escape gets silently dropped by

terminals that don't understand it. but don't emit blindly — you'll see garbage in plain-text pagers if you do. that's what the capability gate is for.

the capability gate

before emitting anything, check that the terminal on the other end is a Warp build that understands `warp://cli-agent`. Warp signals support via two env vars:

variable	meaning
`WARP_CLI_AGENT_PROTOCOL_VERSION`	highest protocol version the client understands. currently `1`. presence is the primary feature flag.
`WARP_CLIENT_VERSION`	Warp's client version string, e.g. `v0.2026.04.15.08.24.stable_03`. used to detect known-broken builds.

the full bash gate from all three adapters:

Skip to content

```
# known-broken Warp releases per channel. these builds advertise protocol
# support via WARP_CLI_AGENT_PROTOCOL_VERSION but don't actually render
# structured notifications — the feature was behind a flag that wasn't
# enabled in the shipped binary.
LAST_BROKEN_DEV=""
LAST_BROKEN_STABLE="v0.2026.03.25.08.24.stable_05"
LAST_BROKEN_PREVIEW="v0.2026.03.25.08.24.preview_05"

should_use_structured() {
    # no protocol version advertised → not Warp, or too old.
    [ -z "${WARP_CLI_AGENT_PROTOCOL_VERSION:-}" ] && return 1

    # no client version → can't rule out broken builds.
    [ -z "${WARP_CLIENT_VERSION:-}" ] && return 1

    # channel-specific "broken floor" check.
    local threshold=""
    case "$WARP_CLIENT_VERSION" in
        *dev*)     threshold="$LAST_BROKEN_DEV" ;;
        *stable*)  threshold="$LAST_BROKEN_STABLE" ;;
        *preview*) threshold="$LAST_BROKEN_PREVIEW" ;;
    esac

    if [ -n "$threshold" ] && [[ ! "$WARP_CLIENT_VERSION" > "$threshold" ]]; then
        return 1
    fi

    return 0
}
```

the `[[ ! X > Y ]]` is bash lexicographic string comparison. Warp's version strings are lexicographically sortable because the date components dominate, so it works.

the simplified TypeScript gate from opencode-warp skips the broken-build check entirely:

Skip to content

```
function warpNotify(title: string, body: string): void {
  if (!process.env.WARP_CLI_AGENT_PROTOCOL_VERSION) return;
  try {
    writeFileSync("/dev/tty", `\x1b]777;notify;${title};${body}\x07`);
  } catch {
    /* /dev/tty unavailable — swallow */
  }
}
```

OpenCode's adapter was authored after the broken stable release, so env-var presence alone is enough. for new integrations you can do the same **unless** you care about users on old Warp builds — in which case copy the full bash gate above.

when the gate fails:

- **subprocess-hook model (Claude / Gemini):** `exit 0` silently, or fall back to a plain-text OSC for old Warp versions.
- **in-process plugins (OpenCode):** return without writing. never error — the user is probably running in a different terminal.

the payload envelope

every structured notification carries the same six-field envelope. the rest of the object is event-specific.

```
{
  "v": 1,
  "agent": "claude",
  "event": "prompt_submit",
  "session_id": "01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app"
}
```

Skip to content

field	type	required	description
`v`	integer	yes	negotiated protocol version. currently only `1` is live.
`agent`	string	yes	host CLI identifier. known values: `claude`, `gemini`, `opencode`. pick a stable slug.
`event`	string	yes	event name. see the catalog below.
`session_id`	string	yes	unique id for the agent session. lets Warp correlate events over time. empty string acceptable when unavailable.
`cwd`	string	yes	absolute working directory of the agent. empty string acceptable.
`project`	string	yes	`basename(cwd)` — the short project label shown in the sidebar. compute it yourself.

version negotiation

```
negotiated_v = min(PLUGIN_MAX_PROTOCOL_VERSION, $WARP_CLI_AGENT_PROTOCOL_VERSION)
```

only `v=1` is defined right now. the mechanism exists so that when Warp introduces a v2 schema, old adapters keep speaking v1 and new adapters down-negotiate when talking to old Warp builds.

```
PLUGIN_CURRENT_PROTOCOL_VERSION=1
negotiate_protocol_version() {
    local warp_version="${WARP_CLI_AGENT_PROTOCOL_VERSION:-1}"
    if [ "$warp_version" -lt "$PLUGIN_CURRENT_PROTOCOL_VERSION" ] 2>/dev/null; then
        echo "$warp_version"
    else
        echo "$PLUGIN_CURRENT_PROTOCOL_VERSION"
    fi
}
```

session binding is implicit

you do not — and cannot — pass a pane or tab id. Warp owns that:

1. Skip to content ce gets written to `/dev/tty`, which is the controlling terminal *of nt runs in*.
2. Warp is reading that pane's output stream, so it sees the sequence in context.

3. the `session_id` in the payload lets Warp group events from the same conversation even if you clear the pane or span multiple windows.

the seven events

seven event values exist in the wild. the first six are emitted by all three reference adapters (modulo host support — see the matrix below). the last two (`question_asked`, `permission_replied`) are OpenCode extensions.

session_start

fires once when the agent starts a new session or resumes an old one. Warp registers the pane in the sidebar and checks whether the installed plugin is up to date.

```
{
  "v": 1,
  "agent": "claude",
  "event": "session_start",
  "session_id": "01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app",
  "plugin_version": "2.0.0"
}
```

extra field: `plugin_version` (string) — the adapter's own version. Warp compares it against a minimum required version and surfaces an "outdated plugin" banner if it's below the floor.

prompt_submit

fires the instant the user submits a prompt. transitions the tab from **idle** / **done** → **running**.

Skip to content

```
{
  "v": 1,
  "agent": "claude",
  "event": "prompt_submit",
  "session_id": "01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app",
  "query": "refactor the auth middleware to use the new session store"
}
```

extra field: `query` (string) — user's prompt, truncated to 200 chars (`...` suffix if longer). truncation rule is consistent across adapters:

```
if [ -n "$QUERY" ] && [ ${#QUERY} -gt 200 ]; then
    QUERY="${QUERY:0:197}..."
fi
```

tool_complete

fires after each tool call completes. transitions the tab from **blocked-on-tool** → **running**.

```
{
  "v": 1,
  "agent": "claude",
  "event": "tool_complete",
  "session_id": "01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app",
  "tool_name": "Bash"
}
```

ex[Skip to content]`ne` (string) — identifier of the tool that just ran. free-form; Warp doesn't enumerate a fixed set.

permission_request

loudest event in the protocol. fires when the agent wants to run a tool that needs user approval. triggers a native OS notification and marks the tab as **blocked-awaiting-permission**.

```
{
  "v": 1,
  "agent": "claude",
  "event": "permission_request",
  "session_id": "01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app",
  "summary": "Wants to run Bash: rm -rf node_modules && npm install",
  "tool_name": "Bash",
  "tool_input": {
    "command": "rm -rf node_modules && npm install",
    "description": "Clean reinstall"
  }
}
```

extra fields:

field	type	description
`summary`	string	human-readable one-liner for the notification. adapters build this as `Wants to run <tool>: <preview>` with preview truncated to 120 chars.
`tool_name`	string	same contract as `tool_complete`.
`tool_input`	object	full tool-input payload, passed through as-is. Warp may render specific keys (`command`, `file_path`) richly.

summary-building heuristic from `on-permission-request.sh`:

Skip to content

```
TOOL_PREVIEW=$(echo "$INPUT" | jq -r '
    (.tool_input | if .command then .command
                   elif .file_path then .file_path
                   else (tostring | .[0:80]) end) // ""
')
SUMMARY="Wants to run $TOOL_NAME"
if [ -n "$TOOL_PREVIEW" ]; then
    if [ ${#TOOL_PREVIEW} -gt 120 ]; then
        TOOL_PREVIEW="${TOOL_PREVIEW:0:117}..."
    fi
    SUMMARY="$SUMMARY: $TOOL_PREVIEW"
fi
```

replicate this logic or your notifications will read as `Wants to run Bash` with no clue what the command actually is.

idle_prompt

fires when the agent has been idle long enough that it probably needs input. the `event` value is whatever the host's notification system labels it as — most commonly `idle_prompt` — so pass it through rather than hardcoding.

```
{
  "v": 1,
  "agent": "claude",
  "event": "idle_prompt",
  "session_id": "01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app",
  "summary": "Claude is waiting for your input"
}
```

extra field: `summary` (string) — free-form message shown in the native notification. def[illegible] `eeded` if the host didn't provide one.

Skip to content

stop

fires when the agent finishes a turn — not session end, just “done talking for now”. transitions the tab to **done** and raises a native notification with the last prompt/response pair.

```
{
  "v": 1,
  "agent": "claude",
  "event": "stop",
  "session_id": "01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app",
  "query": "refactor the auth middleware to use the new session store",
  "response": "I refactored `middleware/auth.ts` to use `SessionStore.get()` and updated the two call sites. Tests pass locally.",
  "transcript_path": "/Users/alice/.claude/projects/my-app/conversation-01J9K7P2.jsonl"
}
```

extra fields:

field	type	description
`query`	string	last user prompt of the turn, truncated to 200 chars.
`response`	string	last assistant response of the turn, truncated to 200 chars.
`transcript_path`	string	absolute path to a JSONL conversation transcript if the host exposes one. can be empty. Warp uses this to let users click through to the full conversation.

the stop-hook race. Claude Code specifics, but relevant to any host that writes transcripts async:

- Claude Code fires `Stop` *before* the transcript file is flushed. the adapter sleeps 0.3s and then reads the last user + assistant messages via `jq`.
- always check a `stop_hook_active` flag if your host exposes one — it’s set to `true` ent is being replayed (e.g. after a recovery), to prevent double

Skip to content

```
STOP_HOOK_ACTIVE=$(echo "$INPUT" | jq -r '.stop_hook_active // false')
[ "$STOP_HOOK_ACTIVE" = "true" ] && exit 0

sleep 0.3  # let transcript flush

TRANSCRIPT_PATH=$(echo "$INPUT" | jq -r '.transcript_path // empty')
# ... read last user + assistant messages from JSONL
```

question_asked (OpenCode extension)

OpenCode has a built-in `question` tool the agent uses to ask clarifying questions. when it's invoked, the adapter sends this event so Warp can distinguish "needs input" from a generic tool call.

```
{
  "v": 1,
  "agent": "opencode",
  "event": "question_asked",
  "session_id": "sess_01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app",
  "tool_name": "question"
}
```

if you're building an agent with a similar meta-tool pattern, reuse this event name — Warp already has UI wired up for it.

permission_replied (OpenCode extension)

fires when the user has responded to a permission request and didn't reject. lets Warp clear the "awaiting permission" state preemptively instead of waiting for the follow-up `tool_complete`.

Skip to content

```
{
  "v": 1,
  "agent": "opencode",
  "event": "permission_replied",
  "session_id": "sess_01J9K7P2E5S8V1Z3B2C4D6F8G0",
  "cwd": "/Users/alice/projects/my-app",
  "project": "my-app"
}
```

only emit this when the user **allowed** the action. rejections are typically followed by a `stop` or another `permission_request` anyway.

two integration shapes

the three official adapters demonstrate two integration shapes: subprocess-hook (bash) and in-process plugin (TypeScript). pick the one that matches your host's extension model.

subprocess hooks (claude-code-warp, gemini-cli-warp)

the host CLI has a hook system where each lifecycle event invokes an external command, passing event data as JSON on stdin. the command:

1. reads stdin.
2. optionally emits structured JSON on stdout to influence the host's behavior (e.g. block a tool call).
3. emits side effects — in our case, an OSC 777 to `/dev/tty`.

registration is a JSON file checked into the adapter. Claude Code's format:

Skip to content

```
{
  "description": "Warp terminal notifications",
  "hooks": {
    "SessionStart": [
      {
        "matcher": "startup|resume",
        "hooks": [
          { "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/on-session-start.sh" }
        ]
      }
    ],
    "UserPromptSubmit": [
      { "hooks": [
        { "type": "command",
          "command": "${CLAUDE_PLUGIN_ROOT}/scripts/on-prompt-submit.sh" }
      ]}
    ],
    "PostToolUse": [
      { "hooks": [
        { "type": "command",
          "command": "${CLAUDE_PLUGIN_ROOT}/scripts/on-post-tool-use.sh" }
      ]}
    ],
    "PermissionRequest": [
      { "hooks": [
        { "type": "command",
          "command": "${CLAUDE_PLUGIN_ROOT}/scripts/on-permission-request.sh" }
      ]}
    ],
    "Notification": [
      {
        "matcher": "idle_prompt",
        "hooks": [
          { "type": "command",
            "command": "${CLAUDE_PLUGIN_ROOT}/scripts/on-notification.sh" }
        ]
      }
```

Skip to content

```
      { "hooks": [
```

```
        { "type": "command",
          "command": "${CLAUDE_PLUGIN_ROOT}/scripts/on-stop.sh" }
      ]}
    ]
  }
}
```

Gemini's format is nearly identical with different event names: `SessionStart`, `BeforeAgent`, `AfterTool`, `Notification`, `AfterAgent`.

per-event script skeleton:

```
#!/bin/bash
# on-prompt-submit.sh
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
source "$SCRIPT_DIR/should-use-structured.sh"

if ! should_use_structured; then
    exit 0
fi

source "$SCRIPT_DIR/build-payload.sh"

INPUT=$(cat)                                          # hook input on stdin
QUERY=$(echo "$INPUT" | jq -r '.prompt // empty')
[ ${#QUERY} -gt 200 ] && QUERY="${QUERY:0:197}..."

BODY=$(build_payload "$INPUT" "prompt_submit" \
    --arg query "$QUERY")

"$SCRIPT_DIR/warp-notify.sh" "warp://cli-agent" "$BODY"
```

the envelope factory (`build-payload.sh`):

Skip to content

```
PLUGIN_CURRENT_PROTOCOL_VERSION=1

negotiate_protocol_version() {
    local warp_version="${WARP_CLI_AGENT_PROTOCOL_VERSION:-1}"
    if [ "$warp_version" -lt "$PLUGIN_CURRENT_PROTOCOL_VERSION" ] 2>/dev/null; then
        echo "$warp_version"
    else
        echo "$PLUGIN_CURRENT_PROTOCOL_VERSION"
    fi
}

build_payload() {
    local input="$1"
    local event="$2"
    shift 2

    local protocol_version session_id cwd project
    protocol_version=$(negotiate_protocol_version)
    session_id=$(echo "$input" | jq -r '.session_id // empty')
    cwd=$(echo "$input" | jq -r '.cwd // empty')
    project=""
    [ -n "$cwd" ] && project=$(basename "$cwd")

    jq -nc \
        --argjson v "$protocol_version" \
        --arg agent "myagent" \
        --arg event "$event" \
        --arg session_id "$session_id" \
        --arg cwd "$cwd" \
        --arg project "$project" \
        "$@" \
        '{v:$v, agent:$agent, event:$event, session_id:$session_id, cwd:$cwd, project:$project} +
$ARGS.named'
}
```

the transport (`warp-notify.sh`):

Skip to content

```
#!/bin/bash
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
source "$SCRIPT_DIR/should-use-structured.sh"
should_use_structured || exit 0

TITLE="${1:-Notification}"
BODY="${2:-}"
printf '\033]777;notify;%s;%s\007' "$TITLE" "$BODY" > /dev/tty 2>/dev/null || true
```

in-process plugin (opencode-warp)

the host CLI exposes a plugin API (TypeScript callbacks, Go plugin interfaces, Python entry points…). you register event handlers that run in the host's own process and write the OSC sequence from there.

sketched in TypeScript:

Skip to content

```ts
import { writeFileSync } from "fs";
import path from "path";

const PLUGIN_VERSION = "0.1.0";
const PLUGIN_MAX_PROTOCOL_VERSION = 1;
const NOTIFICATION_TITLE = "warp://cli-agent";

function negotiateV(): number {
  const w = parseInt(process.env.WARP_CLI_AGENT_PROTOCOL_VERSION ?? "1", 10);
  return isNaN(w) ? PLUGIN_MAX_PROTOCOL_VERSION : Math.min(w, PLUGIN_MAX_PROTOCOL_VERSION);
}

function buildPayload(
  event: string,
  sessionId: string,
  cwd: string,
  extra: Record<string, unknown> = {},
): string {
  return JSON.stringify({
    v: negotiateV(),
    agent: "myagent",
    event,
    session_id: sessionId,
    cwd,
    project: cwd ? path.basename(cwd) : "",
    ...extra,
  });
}

function warpNotify(body: string): void {
  if (!process.env.WARP_CLI_AGENT_PROTOCOL_VERSION) return;
  try {
    writeFileSync("/dev/tty", `\x1b]777;notify;${NOTIFICATION_TITLE};${body}\x07`);
  } catch { /* no tty */ }
}

function truncate(s: string, n: number): string {
  return s.length > n ? s.slice(0, n - 3) + "..." : s;
}

export function onSessionStart(sessionId: string, cwd: string) {
```

Skip to content

```
  warpNotify(buildPayload("session_start", sessionId, cwd, { plugin_version: PLUGIN_VERSION }));
}

export function onPromptSubmit(sessionId: string, cwd: string, prompt: string) {
  warpNotify(buildPayload("prompt_submit", sessionId, cwd, {
    query: truncate(prompt, 200),
  }));
}

export function onToolComplete(sessionId: string, cwd: string, tool: string) {
  warpNotify(buildPayload("tool_complete", sessionId, cwd, { tool_name: tool }));
}

export function onPermissionRequest(
  sessionId: string, cwd: string,
  tool: string, input: Record<string, unknown>,
) {
  const preview =
    typeof input.command === "string" ? input.command
    : typeof input.file_path === "string" ? input.file_path
    : JSON.stringify(input).slice(0, 80);
  warpNotify(buildPayload("permission_request", sessionId, cwd, {
    summary: `Wants to run ${tool}${preview ? `: ${truncate(preview, 120)}` : ""}`,
    tool_name: tool,
    tool_input: input,
  }));
}

export function onIdle(sessionId: string, cwd: string, message = "Input needed") {
  warpNotify(buildPayload("idle_prompt", sessionId, cwd, { summary: message }));
}

export function onStop(
  sessionId: string, cwd: string,
  lastQuery: string, lastResponse: string, transcriptPath = "",
) {
  warpNotify(buildPayload("stop", sessionId, cwd, {
    query: truncate(lastQuery, 200),
    response: truncate(lastResponse, 200),
    ranscriptPath,
}
```

Skip to content

compatibility matrix

which host event maps to which structured event — and who supports what.

event	claude-code-warp	gemini-cli-warp	opencode-warp
`session_start`	`SessionStart` (`startup|resume`)	`SessionStart` (`startup` only)	`session.created`
`prompt_submit`	`UserPromptSubmit`	`BeforeAgent`	`chat.message` callback
`tool_complete`	`PostToolUse`	`AfterTool`	`tool.execute.after`
`permission_request`	`PermissionRequest`	`Notification` w/ `notification_type=ToolPermission`	`permission.updated`, `permission.asked`
`idle_prompt`	`Notification` (`idle_prompt`)	`Notification` (non-ToolPermission)	—
`stop`	`Stop` + transcript parse + 0.3s wait	`AfterAgent` (prompt/response inline)	`session.idle` + SDK fetch
`question_asked`	—	—	`tool.execute.before` when tool === “question”
`permission_replied`	—	—	`permission.replied` (allow only)

legacy fallback

for Warp builds that predate structured notifications, the Claude Code adapter includes a parallel tree of `legacy/*.sh` scripts that emit plain-text OSC 777 notifications with a human-readable title and body.

```
printf '\033]777;notify;%s;%s\007' "Claude Code" "Task complete: $RESPONSE" > /dev/tty
```

these show up in Warp’s notification center as generic text without the sidebar integration. dispatch pattern:

```
if ! should_use_structured; then
    [ "$TERM_PROGRAM" = "WarpTerminal" ] && exec "$SCRIPT_DIR/legacy/on-stop.sh"
```

Skip to content

new integrations can skip the legacy tree entirely. the stable-channel broken build is a year old at time of writing and most users have updated.

edge cases that bite

jq is a hard requirement (for bash adapters)

all bash adapters build payloads with `jq -nc` for proper JSON escaping. if `jq` is missing, the Claude Code `SessionStart` hook emits a visible `systemMessage` telling the user to install it:

```
if ! command -v jq &>/dev/null; then
    cat << 'EOF'
{"systemMessage": "Warp notifications require jq! Install it with brew install jq"}
EOF
    exit 0
fi
```

do not try to build JSON by hand with `printf` — payloads contain user-supplied data (commands, file paths, prompts) that break naive escaping immediately.

never trust stdout to be silent

in the subprocess-hook model, whatever your script writes to stdout gets interpreted by the host CLI. if you emit debug logs to stdout, they'll be parsed as control JSON and may crash the host or produce confusing behavior. log to stderr or to a file.

Gemini CLI's adapter is especially paranoid about this — every script ends with `echo '{}'` to give the host a well-formed empty JSON object so it doesn't misinterpret silence:

```
# ...on ...
# ...o intervention" to the host
```

Skip to content

message.updated fires many times — filter

OpenCode's `message.updated` event fires on every partial token stream update, not once per message. using it as a `prompt_submit` trigger produces dozens of duplicates, and a late one clobbers the `stop` notification. use whatever "message complete" / "user message finalized" signal the host offers instead — OpenCode's is `chat.message`.

if your host doesn't have one, debounce by `session_id` plus a monotonic message counter.

don't emit from non-tty contexts

if your agent runs inside CI, a subprocess without a pty, or a non-Warp terminal, writing to `/dev/tty` will either fail or corrupt the output of whatever parent is reading the stream. the env-var gate (`WARP_CLI_AGENT_PROTOCOL_VERSION`) is the load-bearing check. the `try { ... } catch {}` around the write is a safety net, not a filter.

session-id stability

Warp uses `session_id` to correlate events. if your host regenerates it mid-conversation (e.g. on resume), Warp will treat it as a new session and spawn a new sidebar entry. prefer the host's canonical session id (ULID, UUID, etc.) rather than inventing your own.

protocol version: round-down, not round-up

if Warp advertises `v=2` and your adapter only knows `v=1`, emit `v=1`. Warp must keep parsing `v=1` forever (or at least through the deprecation window). never emit a version you don't actually produce, even if Warp says it supports a higher one.

debugging

Skip to content

se emitting

pipe your script's tty writes to a file temporarily:

```
# before:
printf '\033]777;notify;%s;%s\007' "$TITLE" "$BODY" > /dev/tty

# while debugging:
printf '\033]777;notify;%s;%s\007' "$TITLE" "$BODY" | tee -a /tmp/warp-osc.log > /dev/tty
```

then `cat -v /tmp/warp-osc.log` to see the escape sequences in human-readable form.

validate JSON before emitting

```
BODY=$(build_payload "$INPUT" "stop" --arg query "$QUERY")
echo "$BODY" | jq . >/dev/null 2>&1 || {
    echo "[warp-adapter] malformed body: $BODY" >&2
    exit 0
}
```

malformed JSON in the body makes Warp silently drop the notification. you'll see no error anywhere.

test without Warp

set the env vars manually in a regular terminal:

```
export WARP_CLI_AGENT_PROTOCOL_VERSION=1
export WARP_CLIENT_VERSION="v0.2026.04.21.08.24.stable_01"
```

now `should_use_structured` returns true and your scripts run their full code path. the OSC sequence ends up printed as garbage in the terminal — that's the point, you can `cat -v` it. great for unit tests.

en Skip to content harness

both bash adapters ship a `tests/test-hooks.sh` that stubs stdin with sample Claude / Gemini hook inputs and asserts the emitted bytes. worth reading:

- `warpdotdev/claude-code-warp/tests/test-hooks.sh`
- `warpdotdev/gemini-cli-warp/tests/test-hooks.sh`

OpenCode's adapter has a proper vitest suite in `tests/*.test.ts`.

a complete reference implementation

drop this into any host that lets you run a shell command per lifecycle event. change `AGENT_SLUG`, wire the event handlers, done.

Skip to content

```bash
#!/bin/bash
# warp-adapter/warp-notify.sh
set -euo pipefail

# ----- config -----
AGENT_SLUG="myagent"
PLUGIN_VERSION="1.0.0"
PLUGIN_MAX_PROTOCOL_VERSION=1

LAST_BROKEN_STABLE="v0.2026.03.25.08.24.stable_05"
LAST_BROKEN_PREVIEW="v0.2026.03.25.08.24.preview_05"

# ----- gate -----
should_use_structured() {
    [ -z "${WARP_CLI_AGENT_PROTOCOL_VERSION:-}" ] && return 1
    [ -z "${WARP_CLIENT_VERSION:-}" ] && return 1
    local threshold=""
    case "$WARP_CLIENT_VERSION" in
        *stable*)  threshold="$LAST_BROKEN_STABLE" ;;
        *preview*) threshold="$LAST_BROKEN_PREVIEW" ;;
    esac
    if [ -n "$threshold" ] && [[ ! "$WARP_CLIENT_VERSION" > "$threshold" ]]; then
        return 1
    fi
    return 0
}

# ----- payload -----
negotiate_v() {
    local w="${WARP_CLI_AGENT_PROTOCOL_VERSION:-1}"
    if [ "$w" -lt "$PLUGIN_MAX_PROTOCOL_VERSION" ] 2>/dev/null; then
        echo "$w"
    else
        echo "$PLUGIN_MAX_PROTOCOL_VERSION"
    fi
}

# usage: build_payload <event> <session_id> <cwd> [--arg key val ...]
b
                    ession_id="$2" cwd="$3"
    shift 3
```

Skip to content

```
    local v project
    v=$(negotiate_v)
    project=""
    [ -n "$cwd" ] && project=$(basename "$cwd")
    jq -nc \
        --argjson v "$v" \
        --arg agent "$AGENT_SLUG" \
        --arg event "$event" \
        --arg session_id "$session_id" \
        --arg cwd "$cwd" \
        --arg project "$project" \
        "$@" \
        '{v:$v, agent:$agent, event:$event, session_id:$session_id, cwd:$cwd, project:$project} +
$ARGS.named'
}

# ----- transport -----
emit() {
    local body="$1"
    should_use_structured || return 0
    printf '\033]777;notify;warp://cli-agent;%s\007' "$body" > /dev/tty 2>/dev/null || true
}

# ----- public API -----
warp_session_start() {
    emit "$(build_payload "session_start" "$1" "$2" --arg plugin_version "$PLUGIN_VERSION")"
}
warp_prompt_submit() {
    local q="$3"
    [ ${#q} -gt 200 ] && q="${q:0:197}..."
    emit "$(build_payload "prompt_submit" "$1" "$2" --arg query "$q")"
}
warp_tool_complete() {
    emit "$(build_payload "tool_complete" "$1" "$2" --arg tool_name "$3")"
}
warp_permission_request() {
    local summary="Wants to run $3"
    if [ -n "$4" ]; then
        local p="$4"
        [ ${#p} -gt 120 ] && p="${p:0:117}..."
        summary="$summary: $p"
    fi
    emit "$(build_payload "permission_request" "$1" "$2" \
```

Skip to content

```
        --arg summary "$summary" \
        --arg tool_name "$3" \
        --argjson tool_input "${5:-{\}}")"
}
warp_idle_prompt() {
    emit "$(build_payload "idle_prompt" "$1" "$2" --arg summary "${3:-Input needed}")"
}
warp_stop() {
    local q="$3" r="$4"
    [ ${#q} -gt 200 ] && q="${q:0:197}..."
    [ ${#r} -gt 200 ] && r="${r:0:197}..."
    emit "$(build_payload "stop" "$1" "$2" \
        --arg query "$q" \
        --arg response "$r" \
        --arg transcript_path "${5:-}")"
}

if [[ "${BASH_SOURCE[0]}" == "$0" ]]; then
    cmd="${1:-}"; shift || true
    case "$cmd" in
        session_start)      warp_session_start "$@" ;;
        prompt_submit)      warp_prompt_submit "$@" ;;
        tool_complete)      warp_tool_complete "$@" ;;
        permission_request) warp_permission_request "$@" ;;
        idle_prompt)        warp_idle_prompt "$@" ;;
        stop)               warp_stop "$@" ;;
        *) echo "unknown: $cmd" >&2; exit 64 ;;
    esac
fi
```

use it:

Skip to content

```
source ./warp-adapter/warp-notify.sh

SESSION="sess_$(uuidgen)"
warp_session_start "$SESSION" "$PWD"
warp_prompt_submit "$SESSION" "$PWD" "refactor the retry loop"
warp_tool_complete "$SESSION" "$PWD" "Edit"
warp_permission_request "$SESSION" "$PWD" "Bash" "rm -rf node_modules" '{"command":"rm -rf
node_modules"}'
warp_stop "$SESSION" "$PWD" "refactor the retry loop" "Done, tests pass." "/tmp/transcript.jsonl"
```

tldr

the `warp://cli-agent` protocol is deliberately small. six envelope fields, seven events, one transport primitive, one feature flag. that minimalism is what makes it portable — the three reference implementations share a transport helper that's under 25 lines of code in any language.

adding Warp support to a new agent is almost entirely work in the **host adapter layer** — mapping your CLI's native lifecycle events onto the seven structured events above. the protocol itself fits in an afternoon.

three things to watch as this evolves:

- **protocol v2.** the negotiation machinery exists but only v1 is live. if Warp ever bumps it, expect new optional fields in the envelope (cost, token counts, structured error codes) rather than a rework.
- **new events.** `question_asked` and `permission_replied` started as OpenCode extensions and may get promoted to the core set. if your host has a "clarifying question" concept, emit `question_asked` now.
- **outdated-plugin banner.** Warp compares `plugin_version` in `session_start` against a hardcoded floor. when your adapter ships a breaking change, bump the
Skip to content
dinate with Warp to update their `MINIMUM_PLUGIN_VERSION`

all three adapter repos are MIT licensed. copy the parts you need.

Sources & further reading

1. warpdotdev/claude-code-warp
2. warpdotdev/gemini-cli-warp
3. warpdotdev/opencode-warp
4. Warp notifications docs
5. xterm control sequences (OSC reference)

Questions & answers

▶ Why `/dev/tty` instead of stdout or stderr?

▶ Does this work over SSH?

▶ Do I need jq?

See you down the next rabbit hole.

More notes →