Now we'R coding!

Bringing Agent Assistants Into Live R Sessions

Adam Foryś · Magdalena Krochmal – Roche useR! 2026

Coding assistants are powerful. With two-way communication we bring them inside a live R session – to read live objects, call R, and rewrite a running Shiny UI.

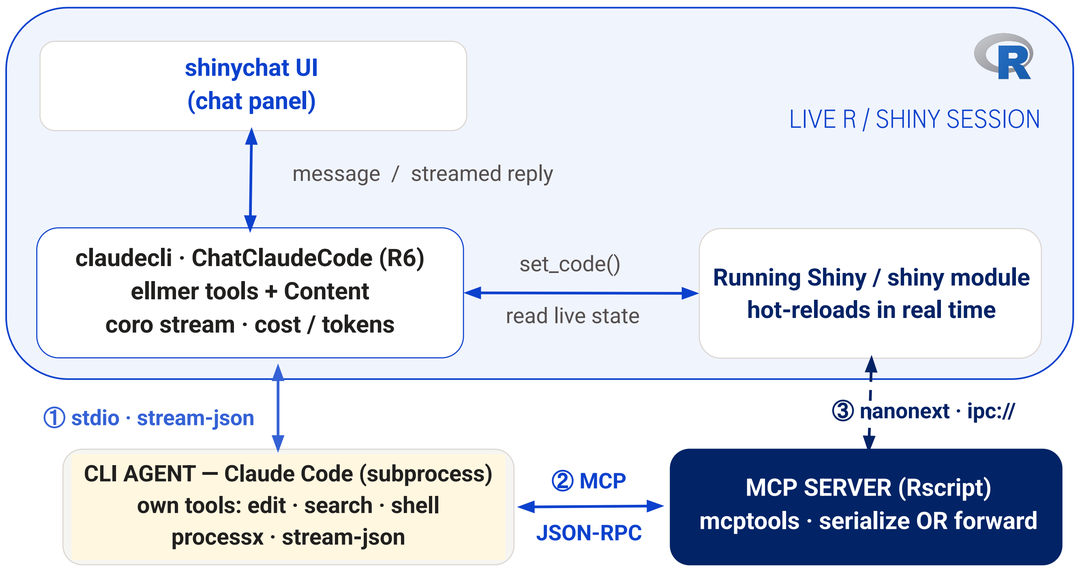
The gap

AI assistants (Claude Code, opencode, Aider) read, write and run code on disk – but they **run beside** your R session. They can't see live objects, call R on live state, or change a running Shiny app.

The idea

Run a **CLI agent** as an ellmer chat provider in R, and open a **two-way link** to the running process. Three protocol layers carry messages, tool calls and live results in both directions.

How the communication works



1. User Asks: The user submits a prompt via terminal. The channel remains strictly two-way, held open to asynchronously receive streamed response chunks.

2. Prompt via stdio: The prompt is piped to the Agent over a two-way stdio connection. The Agent evaluates the request and formulates a tool call.

3. Agent via MCP: The Agent sends a two-way JSON-RPC execution payload to the MCP Server, which can send progress notifications back.

4. Live Session via nanonext: The MCP Server forwards the tool payload into a warm R session over a high-speed, two-way nanonext socket (NNG Pair).

5. Module Hot-Reloads: Just before execution, the live R session checks file hashes and instantly hot-reloads modified modules into the active namespace.

6. Stream to ellmer: The tool yields data chunks that stream back up the exact same open sockets, formatted directly as ellmer Content for the user.

Three bidirectional layers

① stdio · stream-json

claudecli ↔ agent

NDJSON messages: system, assistant (tool_use), stream_event (text deltas), result (cost, tokens).
Parsed into ellmer Content; coro streams tokens to shinychat.

② MCP · JSON-RPC

agent ↔ R tools

ellmer tool() ⟶ JSON Schema via the type bridge.
Served over stdio or http by mcptools; tools return error strings so the agent self-corrects.

③ nanonext · ipc://

the live bridge

Forwards tool calls into the running session (real-time), and runs the fail-secure PreToolUse approval req/rep.

How tools reach live state — and stay safe

Two live-link transports

(a) nanonext forwarding – real-time. Tool runs IN the session via mcp_session(); manager$set_code() ⟶ immediate hot-reload.

(b) file IPC – Shiny-safe. Path baked in via bquote(); subprocess writes a file; reactiveFileReader(1 s) reloads.

Hot-reload without leaks

Agent code is rewritten so observe()/observeEvent() register in a tracking registry.

On reload the previous module's observers are destroyed, then the UI re-renders. Code is validated (parse + required tm_*_ui / tm_*_server); errors return to the agent.

Safe & governed

allow | block

PreToolUse hook asks the live session over nanonext before any tool runs (timeout / error ⟶ block).

Data-access modes – none ⟶ metadata ⟶ summary ⟶ full – decide how much live data the agent ever sees.

Built on three R packages

ellmer

The language – typed tool() objects; the agent's replies arrive as Content that shinychat renders directly.

MCP / mcptools

The wire – JSON-RPC tool calls over stdio or http; serialize in a subprocess, or forward into the session.

nanonext

The live bridge – ipc:// req/rep sockets that forward calls into the running app and return verdicts.

How it works: in dynamic-shiny a scientist describes a chart in a chat; the agent writes a shiny module, it hot-reloads in the preview, if invalid code returns an error the agent self-corrects from – no page refresh, no R expertise required.

The whole integration is small (custom ellmer)

```
# Start a coding assistant — one line
chat <- chat_claude_code(model = "sonnet")
chat$chat("Write a dplyr pipeline that summarises mpg by cyl in mtcars")
# Shiny integration (with shinychat)
chat <- claudecli::chat_with_tools(
  tools = c(core_tools(code_file),data_tools(mode))
  system_prompt = build_system_prompt(teal_data)
)
observeEvent(input$msg, chat_append("chat", chat$stream(input$msg)))
```

```
# Restrict the assistant to read-only tools
chat <- chat_claude_code(
  model = "sonnet",allowed_tools = c("Read", "Glob", "LS"),
  system_prompt = "You are an R pair-programmer."
)
chat$chat("Explain what R/utils.R does, then suggest a unit test")
```

adam.forys@roche.com · magdalena.krochmal@roche.com

Claude Code is the worked example - the pattern fits any MCP-capable coding agent, so we should build a Claude CLI Ellmer provider.