Skills, MCP, and Sub-agents

Teaching an agent your work — and pointing it at real data

SEAN DAVIS, MD, PHD · 2026-07-31

Where we left off

The first session argued that an agent is useful because it can **act** — read your files, run your tools, and check its own work.

This session is about the three things you add to it:

- **Skills** — your procedures, written down once
- **MCP** — connections to real data and real systems
- **Sub-agents** — parallel workers with their own context

…and then the part you cannot install: **how the two of you actually work.**

Speaker notes

The morning’s first session made the case for agents in general terms: what changes when a model can act on files rather than only produce text, what tokens and context windows actually are, and which habits make the tools safer and sharper.

This session is narrower and more practical. It covers the three extension points that turn a general-purpose agent into one that knows your project, and it covers them in the order you would actually adopt them.

The distinction between the three is worth stating plainly at the outset, because the names do not make it obvious and the confusion is common. A skill is *procedural knowledge* — a written procedure the agent loads when it becomes relevant. MCP is *plumbing* — a connection to something outside the agent’s own reasoning, such as a database or an API. A sub-agent is *an isolated worker* — a separate context window doing a side task and reporting back a summary.

They compose rather than compete. A well-built extension usually ships a skill that calls MCP tools and delegates part of the job to a sub-agent. The practical skill being taught here is picking the layer a given task actually lives on, instead of forcing everything into whichever primitive you learned first.

Part 1 — Skills

A skill is a folder

```
bioc-package-check/
├── SKILL.md        # required: frontmatter + instructions
├── scripts/        # optional: code the agent can run
├── references/     # optional: docs loaded on demand
└── assets/         # optional: templates, data
```

`SKILL.md` needs exactly two fields: **`name`** and **`description`**.

Everything else is optional.

Speaker notes

The entire required surface of a skill is a directory containing one Markdown file with YAML frontmatter. The only two mandatory fields are `name`, which must match the directory name and be lowercase with hyphens, and `description`, which is a sentence or two of prose.

That is worth dwelling on, because people arrive expecting a framework and find a text file. There is no build step, no registration, no manifest, and no code required. If you have written a README you have written most of a skill.

The optional subdirectories are what let a skill grow without becoming expensive. `scripts/` holds code the agent can execute rather than reason through — deterministic work belongs in a script, not in prose instructions. `references/` holds documentation the agent pulls in only when it turns out to need it. `assets/` holds templates and data files. The reason this structure exists rather than one enormous Markdown file is explained on the next slide.

Progressive disclosure — why skills are nearly free

Three stages, and only the first one is always paid for:

Stage	What loads	Cost
Discovery	`name` + `description`, for every installed skill	~100 tokens each
Activation	the full `SKILL.md` body, only when relevant	<5,000 tokens
Execution	`scripts/`, `references/`, `assets/` — only if needed	as required

This is the difference between a skill and a `CLAUDE.md` section: **`CLAUDE.md` is always resident.**

Speaker notes

This is the load-bearing design idea, and it is what makes a large skill library practical rather than ruinous.

At startup the agent reads only the name and description of every installed skill — on the order of a hundred tokens apiece. It does not read the bodies. When a task comes in that matches one of those descriptions, that skill's body loads into context. Only if the instructions then call for a script or a reference file does anything further load.

The practical consequence is that installing fifty skills costs you roughly five thousand tokens of permanent context, not fifty full documents. You can afford a broad library of rarely-used procedures.

Contrast this with `CLAUDE.md`, which is read in full at the start of every session and stays resident for the whole of it. That is the right place for facts that apply to everything you do in a repository, and the wrong place for a procedure you run once a month. The migration rule follows directly: when a section of `CLAUDE.md` has become a *procedure* rather than a *fact*, it wants to be a skill.

One caveat that matters once a skill is loaded: its body stays resident for the rest of the session. A five-hundred-line skill is not a one-time read, it is a recurring cost on every subsequent turn. Hence the convention of keeping bodies under roughly five thousand tokens and pushing detail into `references/`.

The description is the whole game

The agent never sees the body until it has already decided to load it.

Weak: “Helps with Bioconductor packages.”

Strong: “Reviews an R/Bioconductor package for common submission issues — NAMESPACE completeness, vignette presence, BiocCheck warnings, DESCRIPTION metadata. Use when the user asks to check, review, or prepare a package for submission, or mentions BiocCheck, R CMD check, or Bioconductor guidelines.”

Write what it does **and when to use it**. Include the words you would actually say. Lean pushy — the common failure is a skill that never fires.

Speaker notes

Because discovery loads only the description, that one field carries the entire triggering signal. A skill with a brilliant body and a vague description will sit unused forever, and you will conclude that skills do not work.

The guidance that follows from this is concrete. Write in the third person, since the description is injected alongside every other skill’s description and inconsistent voice makes matching harder. State what the skill does and then when to use it, in that order, with the load-bearing words early — descriptions are truncated if you run long. Most importantly, include the specific vocabulary someone would actually use. A Bioconductor developer says “BiocCheck” and “submission,” so those words belong in the description even though a human reading it would find them redundant.

Lean assertive rather than hedged. The observed failure mode is under-triggering, not over-triggering: agents are more likely to ignore a relevant skill than to invoke an irrelevant one. Descriptions that confidently assert relevance work better than descriptions that qualify.

For anything with side effects, do not try to write a description precise enough that it fires only sometimes. Set `disable-model-invocation: true` and invoke it yourself with a slash command. Deterministic timing is a frontmatter field, not a prompting problem.

It is not a Claude thing any more

`SKILL.md` is now an open standard, governed at agentskills.io, with **40+ products reading the same format**.

Cursor · **OpenAI Codex CLI** · **Google Gemini CLI** · GitHub Copilot · VS Code · JetBrains Junie · Goose · Letta · OpenHands · Amp · Kiro …

A format Anthropic shipped as a Claude Code feature is now read, unmodified, by OpenAI's own coding agent.

Speaker notes

This is the finding most worth carrying out of the session, because it changes the investment calculation.

Skills began as a Claude Code feature. Within a year the format was moved to an independent specification with its own governance, a reference validator, and a published client showcase, and it is now read natively by more than forty products. That list includes direct competitors: OpenAI’s Codex CLI and Google’s Gemini CLI both consume `SKILL.md` files without translation.

Cross-vendor cooperation of this kind is unusual in this market, and the reason to raise it with a room of package developers is that it answers the question they should be asking before writing anything. Procedures you encode as skills are not a bet on one vendor. The workshop prework told attendees to pick any of four agents; a skill written this morning works across all of them.

The same argument applies at the community level. If Bioconductor encodes its package-review conventions as skills, that investment is not contingent on which tool the community settles on, and it does not need redoing when someone switches editors.

Installing a skill — four mechanisms

```
# 1. Personal — every project you work on
mkdir -p ~/.claude/skills/my-skill      # then write SKILL.md

# 2. Project — this repo only, committed to git
mkdir -p .claude/skills/my-skill        # then commit it

# 3. Plugins and marketplaces — the native package manager
/plugin marketplace add anthropics/skills
/plugin install document-skills@anthropic-agent-skills

# 4. skills.sh — cross-agent installer (Vercel)
npx skills find
npx skills add <owner>/<repo>
```

Speaker notes

Four mechanisms, and the choice between the first two is the one that matters day to day.

A personal skill in `~/.claude/skills/` follows you across every project. That is right for things about *you* — how you like commit messages written, a review checklist you apply everywhere.

A project skill in `.claude/skills/` lives in the repository and is committed to git. That is right for things about *the project*, and it is the one that matters for a package with contributors, because it means the procedure ships with the code. A new contributor who clones your package gets your conventions without being told them. For a Bioconductor package with an established review history, this is the highest-value use of the whole mechanism.

Plugins are the packaged form: a single install can carry skills, hooks, MCP servers, and sub-agents together, namespaced so they cannot collide with your own. Marketplaces are just git repositories with a manifest, so an organisation can run its own.

`skills.sh` is a cross-agent installer maintained by Vercel, actively developed, which installs from GitHub shorthand, any git URL, or a local path, and works across more than twenty agents. There is a similarly-named but much smaller project at a different domain; check you are on the Vercel-backed one before piping anything from it.

One historical note to save confusion in older tutorials: `.claude/commands/*.md` files, the pre-2026 custom slash commands, still work and are now treated as a subset of skills. If both exist under one name, the skill wins.

Collections worth knowing

Star counts verified directly against the GitHub API, 2026-07-30 — the secondary write-ups about this ecosystem are full of stale numbers.

Collection	What it is
`obra/superpowers`	A whole methodology — TDD, systematic debugging, subagent-driven development. Most-starred of the three.
`mattpocock/skills`	“Skills for Real Engineers” — `/grill-me`, `/to-spec`, code review, domain modelling.
`anthropics/skills`	The official examples, the spec source, a starter template, and document generation (PDF/DOCX/XLSX).

Five minutes — go browse. Scan the code to open this deck on your own laptop, then click the three links above. Find one skill you would actually use this week.

Both community collections have **more stars than Anthropic’s own**.

Speaker notes

Three collections are worth knowing by name, and the ranking between them is mildly surprising: the two community collections are both larger than the official one.

This is the first of two short browsing breaks. Five minutes is not enough to evaluate a collection and is not meant to be — it is enough to discover that these are readable Markdown files rather than machinery, which is the thing that has to land before anyone will write one. Ask for one concrete answer at the end: which skill would you use this week, and why that one. A specific answer from four or five people is worth more than a general discussion.

The QR resolves to this deck, so people can open it on their own screens and follow the links rather than copying URLs off a projector. Worth saying out loud that it is the deck and not a sign-up of any kind.

Superpowers, from Jesse Vincent, is the most-starred and the most opinionated — it presents itself as a complete development methodology rather than a set of utilities, covering test-driven development, systematic debugging, verification before completion, and a set of skills for running work across parallel agents. It is carried in Anthropic's official plugin marketplace as well as its own, so either install path is legitimate.

Matt Pocock's collection is his own working `.claude/skills/` directory, published. His framing is a diagnosis of four failure modes — misalignment, verbosity, broken code, and poor architecture — with a skill answering each. His `/grill-me` appears later in this session.

Anthropic's repository is the place to go for the specification itself and a starter template, and its document-generation skills are directly useful for producing analysis reports.

The methodological point, more durable than any of the three: the star counts above were checked against the GitHub API rather than taken from articles, and the articles disagreed with each other. This corner of the web is thick with sites publishing large unverified numbers about how many skills exist. Check repository activity directly. A collection whose last commit is three months old is telling you something that its star count is not.

Bioconductor already has one

`Bioconductor/ai-agent-skills` — active, led by Levi Waldron with Marcel Ramos.

`analyze-r-package` · `improve-code-coverage` · `security-audit-r-package` · `update-r-news` · `create-package-instructions` · `bioc-pkg-finder` — thirteen in all.

And `BiocCheck` itself carries commits where an agent wrote the tests, marked with an `assisted-by:` trailer.

> **Five more — now read one of these.** Open `Bioconductor/ai-agent-skills` and read the `SKILL.md` closest to your own work.
>
> Would it have caught the last thing that **bounced in review**?

Speaker notes

This is not a hypothetical for this community. There is an active Bioconductor repository of agent skills, created in May 2026 and updated within the last few days, led by Levi Waldron with Marcel Ramos, carrying thirteen skills aimed squarely at package development: analysing a package, improving its code coverage, auditing it for security issues, updating its NEWS file, finding packages, and validating a skill you have written yourself.

Separately, and more quietly, agent-assisted work has already landed in core infrastructure. BiocCheck carries commits where an agent wrote the unit tests and refactored a checker, attributed with an `assisted-by:` trailer in the commit message.

That is a convention emerging in practice rather than by policy, which is exactly the sort of thing this workshop is well placed to examine. The afternoon focus group has a natural question waiting for it: Bioconductor's contribution guidelines currently say nothing at all about AI assistance, while rOpenSci — a sibling community with a comparable review culture — published an explicit disclosure policy for package review in February. The gap is real, citable, and squarely the community's own decision to make.

Write one, right now

```
---
name: bioc-package-check
description: Reviews an R/Bioconductor package for common submission
  issues — NAMESPACE completeness, vignette presence, BiocCheck warnings,
  and DESCRIPTION metadata. Use when the user asks to check, review, or
  prepare a Bioconductor package for submission, or mentions BiocCheck,
  R CMD check, or Bioconductor guidelines.
allowed-tools: Bash(R CMD check *) Bash(Rscript -e *)
---

## Instructions

1. Run `R CMD check --as-cran .` and
   `Rscript -e 'BiocCheck::BiocCheck(".")'`; capture full output.
2. Confirm `vignettes/` has at least one .Rmd/.qmd that builds.
3. Confirm every `NAMESPACE` export has an `@export` tag and a
```

Speaker notes

A complete, working skill, short enough to read on a slide, and a reasonable template for the first one you write for your own package.

Several deliberate choices are worth naming. The description front-loads what the skill does and then when to use it, and it contains the exact words a Bioconductor developer would say — “BiocCheck”, “submission”, “R CMD check” — so it matches real requests rather than paraphrases.

`allowed-tools` pre-approves exactly the two commands the procedure needs. Without it the agent stalls on a permission prompt in the middle of a check run. With it scoped this narrowly, the skill cannot do anything beyond what it declares.

The body states steps rather than narrating intentions. “Run this, confirm that, report the other” is what belongs here. Explanations of *why* Bioconductor wants a vignette do not, because they cost context on every turn once loaded and the agent does not need convincing.

The last line is the most important one and the easiest to leave out. Reporting and fixing are different operations, and an agent that silently fixes what it finds destroys the value of the check — you no longer know what was wrong. Say so explicitly. This generalises well beyond this example: when a procedure has a tempting adjacent action, forbid it in the text.

The failure you will not notice

You wrote `bioc-package-check`. A week later you type:

> “Can you look over my package before I submit it?”

Nothing happens.

Your description said *check*, *review*, *prepare*, *BiocCheck*, *R CMD check*. It never said **“look over”**.

A skill that does not fire is indistinguishable from a skill you never wrote.

Speaker notes

The obvious failure mode with skills is not a skill that misbehaves. It is a skill that sits there, never triggering, while you conclude the mechanism does not work.

Trace what happened. The description on the earlier slide was a good one by the usual standards — it named the operations and the vocabulary a Bioconductor developer uses. But a real request came in phrased as “look over,” which is not “check,” not “review,” and not “prepare,” and the model matched it against nothing. The skill was never loaded, so its body — however well written — never had a chance to matter.

This failure is quiet in a specific and expensive way. There is no error, no warning, and no indication that a capability existed and went unused. You simply get the answer you would have got without the skill, which looks like a normal answer. Contrast that with a skill whose *body* is wrong: you see the bad output and fix it. Trigger failures are invisible, which is why they survive.

The lesson generalises past skills. Anywhere a model chooses whether to invoke something based on a short description, the description is an interface with a matching problem, and it deserves the scrutiny you would give any other interface. The next slide is what scrutiny looks like.

So test the description, not the body

Should fire	Should not fire
“look over my package before I submit it”	“what does `biocViews` mean?”
“is this ready for Bioconductor?”	“write a testthat test for this”
“run BiocCheck and tell me what’s blocking”	“explain S4 dispatch to me”

```
/plugin install skill-creator@claude-plugins-official
```

Generates both lists, runs each in an isolated sub-agent, measures the hit rate, and rewrites the description until it passes — then benchmarks the task **with the skill against without it**.

Speaker notes

The test for a description is a pair of lists: phrasings that should load the skill, and phrasings that should not. Both halves matter. A description tuned only for recall will fire on everything, and a skill that loads when you asked what `biocViews` means has spent five thousand tokens to answer a one-line question.

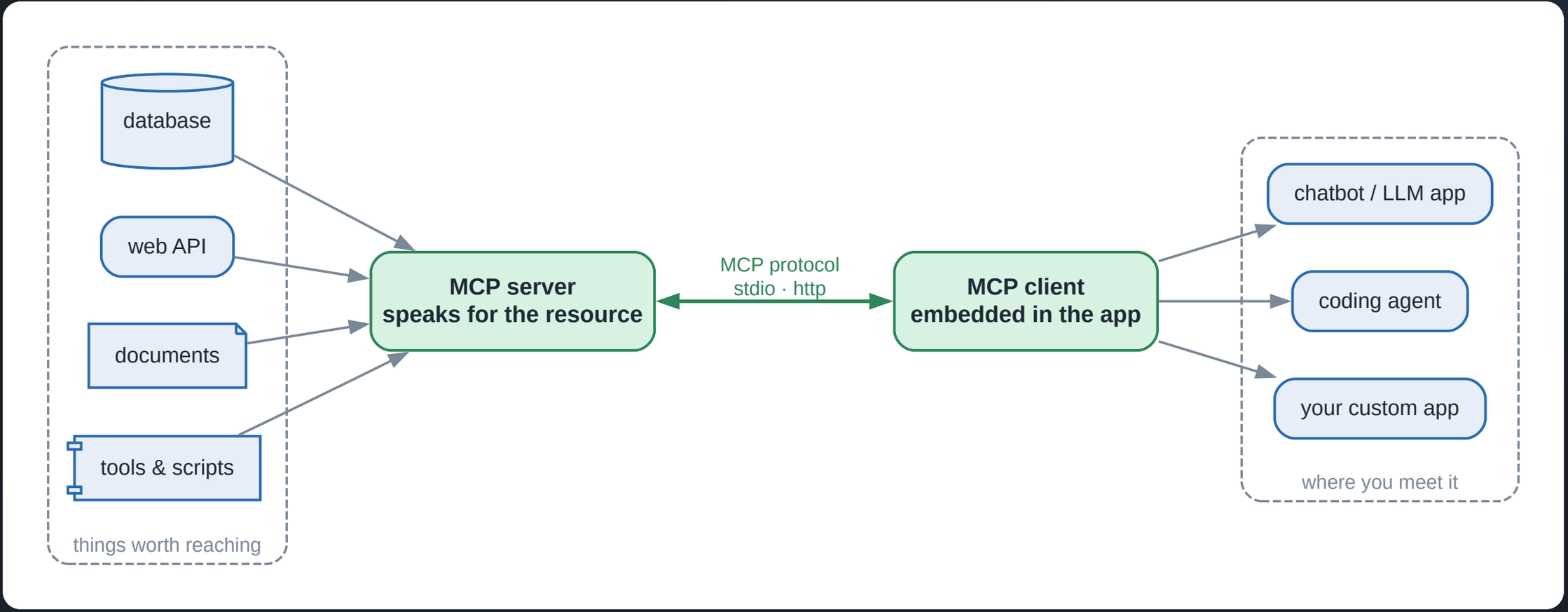
Write the should-fire list from how people actually ask, not from how you would describe the skill. “Look over my package,” “is this ready,” “what’s blocking” — those are the sentences that show up. The should-not-fire list is best drawn from neighbouring questions in the same domain, because those are the genuinely hard negatives. “Explain S4 dispatch” is about R packages and is not a request to check one.

Anthropic’s `skill-creator` automates the loop rather than just the scoring. It generates both lists, runs each prompt in an isolated sub-agent so cases cannot contaminate one another, measures the hit rate, and rewrites the description until it passes. It will also benchmark the task with the skill against the same task without it, which answers the question underneath all of this: is this earning the context it costs? A skill that changes nothing should be deleted, and without the comparison you would never find out.

For a room that already writes unit tests, this needs little defending. A skill is code and its description is an interface, so both get tests. If the workshop demonstrates one thing beyond writing a first skill, this is the better candidate — it is the step practitioners skip, and the one that separates a library that works from a folder of hopeful Markdown.

Part 2 — MCP, operationally

The whole architecture, one picture



A **server** speaks for something worth reaching; a **client** lives inside the app you already use. Everything between them is **one protocol** — so each side is written once.

Speaker notes

Before any configuration, the shape of the thing, because every operational decision in this part is a decision about one of these four columns.

Read it left to right. On the left are the things worth reaching — a database, a web API, a pile of documents, your own tools and scripts. None of them speak LLM. On the far right are the places you actually meet an agent — a chatbot, a coding agent, an application you build yourself. None of them speak Postgres, or REST, or your filesystem's conventions.

MCP's answer is the two boxes in the middle. A *server* sits in front of a resource and exposes it as tools with names and descriptions. A *client* sits inside the host application and consumes them. Between the two runs exactly one protocol, and the transports on the wire — stdio and http — are the next slide's subject.

The economics are the point of the picture. Without the middle layer, every app–resource pair needs its own integration: M apps times N resources. With it, each resource is wrapped once and each app implements the client once — M plus N. That arithmetic is why the registry from this morning can exist at all, and why a server someone wrote for Claude works unmodified from Codex.

One honest simplification to name if asked: in the real architecture the client lives *inside* the host application — one client connection per server, several per app. The diagram draws it as a single column because the thing worth remembering is the layering, not the multiplicity.

A server is three decisions

name	yours to pick — it namespaces every tool the server exposes, and it is how you remove it later
transport	`stdio` = a process **you** launch · `http` = a URL **someone else** runs
location	`command` + `args` for stdio · `url` for http

The name is not cosmetic. Call it `biocontext` and its tools arrive as `mcp__biocontext__search_uniprot` — that prefix is what keeps two servers with a `search` tool from colliding, and what you will read in a permission prompt at 2am.

Speaker notes

Before any command, the shape. Every MCP server you will ever add is these three decisions, and every tool on the next several slides is a different syntax for writing them down.

The **name** is the one people treat as throwaway and should not. It namespaces the tools: a server called `biocontext` contributes `mcp__biocontext__search_uniprot`, and a differently-named server offering its own `search` cannot collide with it. That prefix is also what appears in a permission prompt, so a name like `server1` costs you the ability to tell, at a glance, what is asking for what. Name it after the resource, not the software. It is also the handle for removal, which is the other reason to make it memorable.

The **transport** is the decision with real consequences. `stdio` means the client launches a process on your machine and talks to it over standard input and output — the server runs locally, inherits your filesystem access, and dies with your session. `http` means a URL, typically run by someone else, with nothing installed locally and nothing to break. For a workshop the second is safer; for anything touching data that must not leave the building, the first is the only option, which is exactly the argument the earlier institutional-cohort demo rested on.

There is a third transport you will meet in older documentation, HTTP with server-sent events. The specification revision of 2026-07-28 formally deprecated it on a twelve-month clock in favour of the newer streamable HTTP. You do not need to care unless you are writing a server, but you should recognise `sse` in a config file as something on its way out rather than a choice to copy.

The **location** follows from the transport, and that is the whole of it.

Scope is the decision

```
claude mcp add --transport http biocontext https://biocontext-kb.fastmcp.app/mcp
```

Scope	Where it lives	Use it for
`local`	this project, your machine	trying something out
`project`	**`.mcp.json`, committed to git**	a shared repo — a workshop, a package
`user`	every project you open	the two or three you genuinely always want

`--scope project` turns thirty laptops into **one clone and one approval prompt.**

Speaker notes

Adding a server is one command. The flag worth thinking about is the scope, because it is the one people get wrong in both directions.

Local scope keeps the server to one project on your machine, which is right for experiments. User scope makes it available in every project you ever open, which is right for the small number of servers you genuinely always want and wrong as a default — a standing grant to every session forever is more access than most servers need.

Project scope is the interesting one for this room. It writes a `.mcp.json` file in the repository root, which you commit. Anyone who clones the repository gets the same servers after a single approval prompt.

For a workshop that is the difference between thirty people following install instructions and thirty people running one clone. For a package with contributors it means the tools your project expects come with the project. And for reproducibility — a concern this community takes more seriously than most — it means the agent’s access to external resources is version-controlled alongside the analysis, rather than living in someone’s undocumented local configuration.

The approval prompt exists precisely because a cloned repository should not be able to launch a process or reach a remote host without your consent. Treat it as a real decision rather than a rubber stamp, and read the file first.

A workshop `.mcp.json`

```
{
  "mcpServers": {
    "biocontext": {
      "type": "http",
      "url": "https://biocontext-kb.fastmcp.app/mcp"
    },
    "pdbe": {
      "type": "stdio",
      "command": "uvx",
      "args": ["pdbe-mcp-server", "--server-type", "pdbe_api_server"]
    },
    "gget": {
      "type": "stdio",
      "command": "uvx",
      "args": ["--from", "gget-mcp@latest", "stdio"]
    }
  }
}
```

Verify with `claude mcp list` — each should report **connected**.

Speaker notes

Three servers, all free, all without authentication, chosen because they survive a conference network.

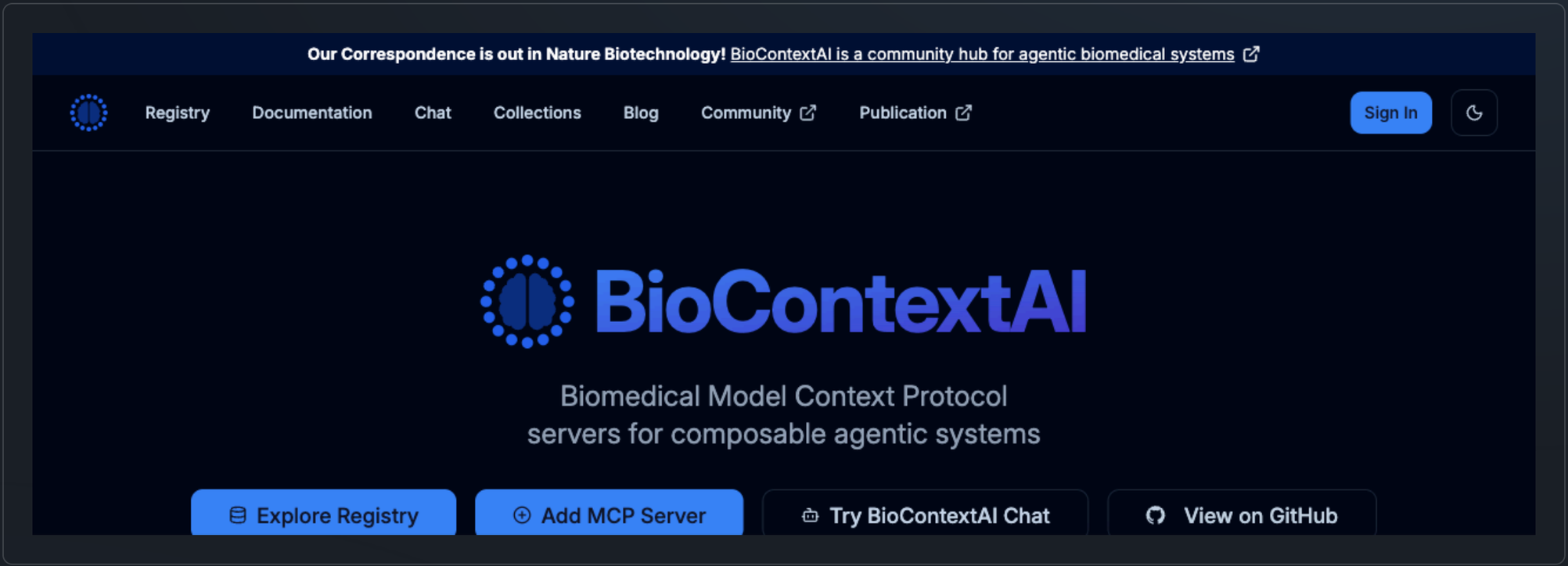
BioContextAI’s knowledgebase server is hosted remotely, so nothing installs locally and there is nothing to break on your laptop. It exposes on the order of fifty tools spanning UniProt, STRING, AlphaFold, Ensembl, Open Targets, Europe PMC, bioRxiv, ClinicalTrials.gov, PanglaoDB, and InterPro. Its endpoint was confirmed responding the day before this session.

The PDBe server is an official EMBL-EBI project covering structures, ligands, and associated publications. gget-mcp wraps the Pachter Lab’s `gget` toolkit, which reaches Ensembl, BLAST, AlphaFold, PDB, CELLxGENE, and COSMIC through one package.

Two operational details worth knowing before you troubleshoot. The `uvx` servers download their package on first run, so they can report a failed connection for a few seconds on a cold cache — retry before concluding anything is wrong. And if you test the hosted endpoint with `curl`, a `405` response means it is reachable and healthy: MCP endpoints answer POST only, so a rejected GET is the correct behaviour, not an error.

Deliberately absent is BioMCP, the best-known biomedical server. It is capable and worth exploring afterwards, but it is mid-rewrite and aggregates roughly twenty-five upstream APIs, which is a large surface for something to fail behind during a live demonstration.

`biocontext` — ten databases, one endpoint



Hosted — nothing installs. **~49 tools** across UniProt, STRING, AlphaFold, Ensembl, Open Targets, Europe PMC, bioRxiv, ClinicalTrials.gov, PanglaoDB, InterPro.

“Canonical PanglaoDB markers for microglia — and does recent literature agree?”

“What does UniProt say about TP53, and its highest-confidence STRING partners?”

Speaker notes

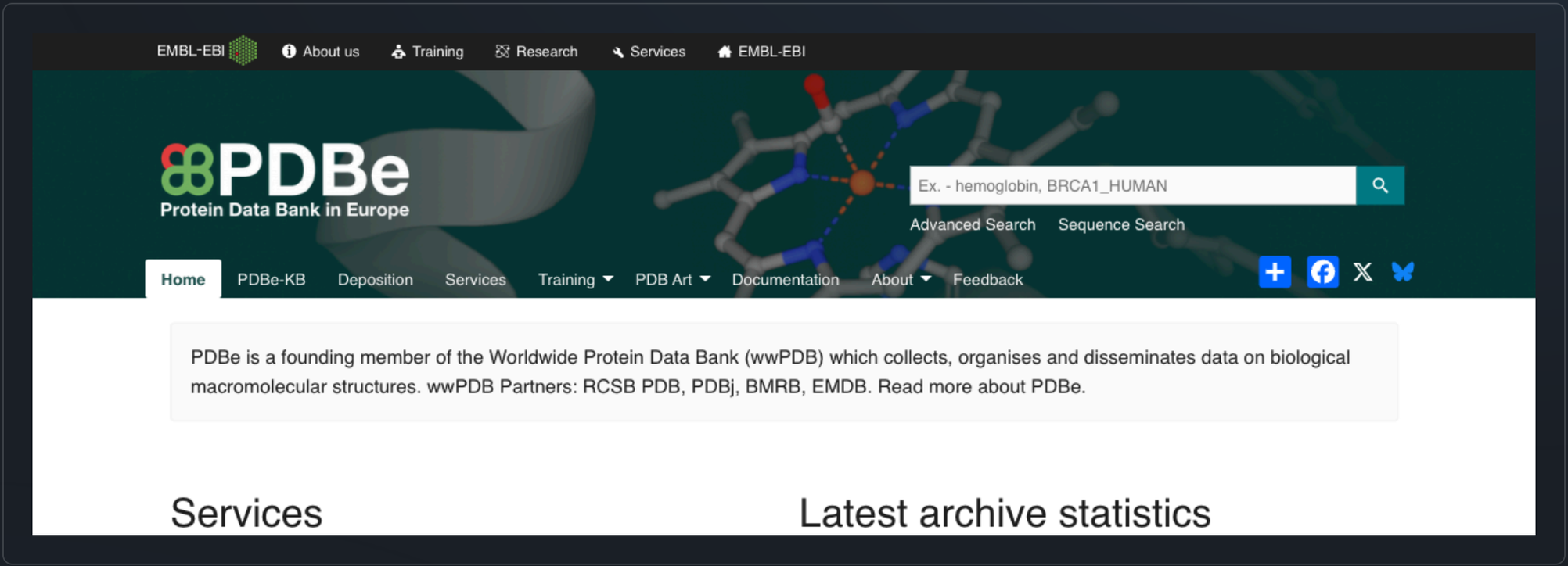
The broadest of the three, and the only hosted one, which is why it leads the workshop configuration: there is nothing to install and nothing to break locally.

BioContextAI maintains both this knowledgebase server and the registry catalogued earlier in the morning. The banner across the top of that page is their correspondence in *Nature Biotechnology*, which is the citable reference if you want one.

Its natural questions are the ones that cross resources, because that is what having ten databases behind a single interface actually buys you. A lookup against one of them is something a browser tab does perfectly well.

One live defect to route around, verified the day before this session and worth knowing because it is what this room would try first: the tools that resolve a gene *symbol* to a UniProt accession return the wrong accession — TP53, EGFR and BRCA1 all come back as other proteins. Tools that take a gene symbol directly are fine. It is the symbol-to-identifier resolver that is broken, and it fails silently and confidently rather than erroring.

pdbe — structures, from the people who curate them



EMBL-EBI's Protein Data Bank in Europe. Structures, ligands, and the publications behind them — an **official project of the group that owns the data**, not a community wrapper.

"What structures exist for human p53, and which have a bound ligand?"

"Get the InterPro domain architecture for P04637."

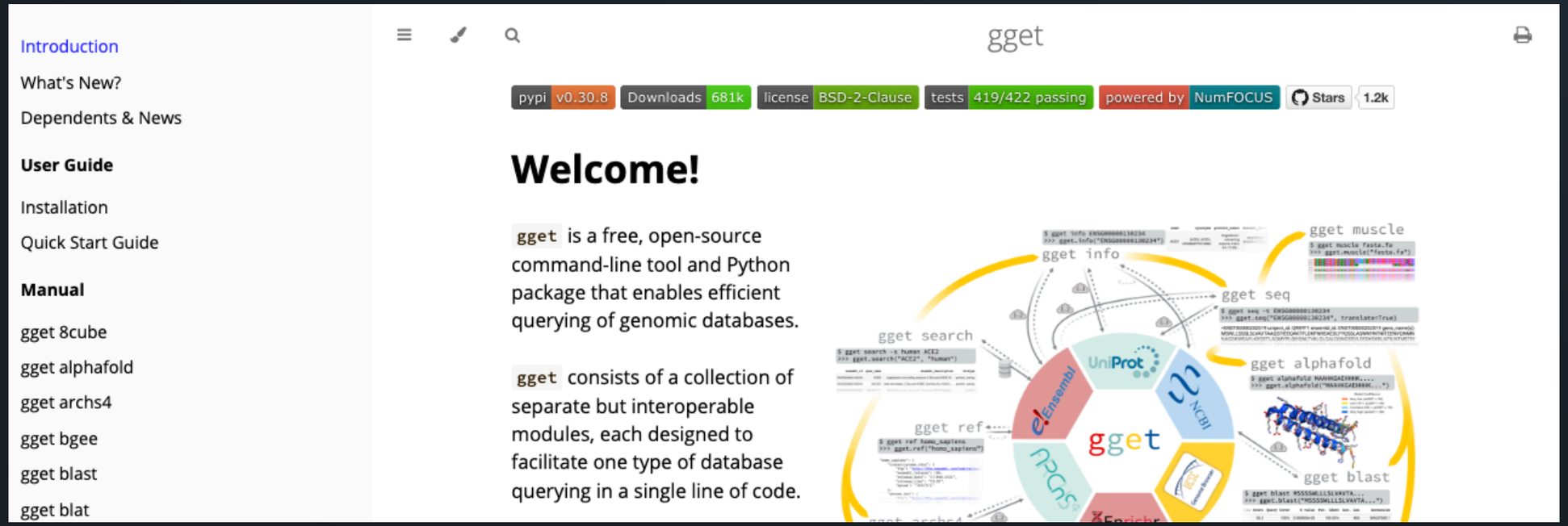
Speaker notes

The provenance is the reason to prefer this one. Its MCP server is published by EMBL-EBI, the organisation that curates the underlying archive, rather than by a third party wrapping a public API. That is the single most useful heuristic for judging a server you are about to trust with your session: does the group that owns the data also ship the server?

Practically it suits structural questions, and structures have the incidental advantage of being satisfying to look at, which matters more in a live demonstration than anyone admits.

Note the second example passes the accession directly rather than the gene symbol. That is deliberate — it is how you avoid the resolver defect from the previous slide while still asking a structural question about p53.

`gget` — a familiar toolkit, wrapped



The Pachter Lab's `gget`, which many of you already run from the command line. One server reaching **Ensembl, BLAST, AlphaFold, PDB, CELLxGENE, COSMIC**. · ▶ preview here

"What is the Ensembl ID for ACE2, and which tissues express it?"

"BLAST this sequence and tell me the closest annotated match."

Speaker notes

Included partly because a good number of people in this room already use `gget` as a command-line tool, which makes it the least abstract of the three: the MCP server wraps a toolkit they can already reason about.

It is also the best demonstration of how much surface a small server can cover. One package reaches Ensembl, BLAST, AlphaFold, PDB, CELLxGENE and COSMIC, which is a reminder that “how many servers do I need” is usually the wrong question.

One practical note: its documentation now redirects to a `scverse` address. The project moved under that umbrella and older links still resolve, but if you go looking for the docs and end up somewhere unexpected, that is why.

The three of them, in one file

```
{
  "mcpServers": {
    "biocontext": {
      "type": "http",
      "url": "https://biocontext-kb.fastmcp.app/mcp"
    },
    "pdbe": {
      "type": "stdio",
      "command": "uvx",
      "args": ["pdbe-mcp-server", "--server-type", "pdbe_api_server"]
    },
    "gget": {
      "type": "stdio",
      "command": "uvx",
      "args": ["--from", "gget-mcp@latest", "stdio"]
    }
  }
}
```

One hosted, two local. Commit it, and the room clones instead of installing.

Speaker notes

The same file as before, now that the three names mean something. Worth a second look rather than a second reading, because the shape says things the first pass could not.

`biocontext` is `http` with a `url`: somebody else runs it, nothing arrives on your laptop. `pdbe` and `gget` are `stdio` with a `command`: your machine launches those processes, and they inherit your environment. That is the transport distinction from the start of this part, visible in the file rather than described.

Note also what is absent — no API keys, no tokens, no auth block anywhere. That is a deliberate selection rather than a property of biomedical servers generally, and it is most of why these three can be demonstrated on conference wifi.

The `uvx` entries download their package the first time they run, so a cold cache can report a failed connection for a few seconds. Retry before concluding anything is wrong.

Two databases, one question

> *“What are the canonical PanglaoDB marker genes for microglia in humans, and can you find recent bioRxiv or Europe PMC papers about microglial markers to see if the field agrees?”*

Marker lookup and a live literature cross-check, in one sentence, across two unrelated resources — and the answer is checkable against what you already know from `scran` or `SingleR`.

Tested live 2026-07-30: returns ITGAM (CD11b) and ITGAX among the top markers.

Speaker notes

The demonstration worth giving is not “the agent can query a database” — a browser tab does that. It is that a single sentence crosses two unrelated resources and returns something you can immediately check.

This prompt asks for canonical microglia markers from PanglaoDB and then for recent literature on the same topic, and the agent decides on its own which tools to call and in what order. The marker lookup returns in well under a second. The answer is verifiable by the audience from memory, which is the property that makes it convincing: ITGAM and ITGAX are textbook microglia markers, and a room that works with single-cell data will recognise them without being told.

One live defect is worth knowing about before you improvise against this server, because it is exactly what a room of biologists would try first. The tools that resolve a gene *symbol* to a UniProt accession return the wrong accession — TP53, EGFR, and BRCA1 all come back with accessions belonging to other proteins. Tools that take a gene symbol directly, or an accession directly, are fine; it is the symbol-to-identifier resolver that is broken, and it is broken silently and confidently. The AlphaFold-structure-by-gene-symbol path goes through that resolver, so avoid it.

That is a useful accident for a teaching session. It is a concrete instance of the general warning: a confident answer from a tool is still an answer you check.

The tool result is untrusted input

An agent cannot reliably distinguish **instructions from you** from **text that arrived inside a tool result**.

So a compromised paper abstract, variant annotation, or search result can steer it.

- Keep **read-only query tools** separate from anything with **write or execute** power
- Prefer **institutionally-backed** servers for real research data
- Use **project scope**, not standing `user`-scope grants
- **Read `.mcp.json` before you approve it**

Speaker notes

The security risk that matters here is not network-layer attack. It is that the agent reads tool results as text, and text can contain instructions.

A malicious or merely compromised server can place instructions in a tool’s description, which the model reads and the human usually does not inspect, or in the data it returns — an abstract, an annotation, a search hit. The agent has no reliable way to separate “my user asked for this” from “this arrived inside a result.” Given both a poisoned source and the ability to write files or run commands, it can be steered into doing something neither you nor the server’s author intended.

Four practical mitigations follow, in rough order of value. Keep read-only biomedical query tools in a different session from anything with write or execute power; the combination is the attack surface, not either alone. Prefer servers backed by the institution that owns the underlying resource — EMBL-EBI for PDBe, for instance — not because small projects are malicious, but because a two-person repository has nobody auditing what a dependency update silently added to a tool description. Grant narrowly: project scope for the repository that needs it, rather than user scope forever. And read `.mcp.json` before approving it, which is the entire reason the prompt exists.

For this audience there is a sharper version. If there is unpublished patient or sample data anywhere in the session’s reach, the question is not whether the server is trustworthy in the abstract, but what it could reach if it were not.

R can serve MCP

The direction people forget: not R *calling* an agent, but an agent calling **your live R session**.

```
# .Rprofile — makes a running session discoverable
if (interactive()) btw::btw_mcp_session()
```

```
claude mcp add -s "user" r-btw -- Rscript -e \
  "btw::btw_mcp_server(list('docs','pkg','env','sessioninfo','cran'))"
```

The agent inspects objects **you already have loaded** — not a fresh subprocess.

`mcp_server()` alone surfaces **no useful tools**, and that list is read-only. Add `'run'` to the groups and the agent can execute R — and return **plots as images**. Off by default, deliberately.

Speaker notes

Every example so far has pointed an agent at somebody else's data. This points it at yours, and it is the one that tends to reframe how this audience thinks about the protocol.

`mcptools`, from Posit and on CRAN, lets R act as an MCP server. Two modes are worth distinguishing. `mcp_server()` runs a fresh R process for the agent to use, which is the conventional arrangement. `mcp_session()`, placed in your `.Rprofile`, makes a *running interactive session* discoverable — so the agent can inspect the object you have loaded right now, in the state you have got it into, mid-debugging.

The part that surprises people, and the reason the second command names `btw` rather than `mcptools`, is that `mcp_server()` on its own deliberately surfaces nothing useful. The documentation is blunt about it: the server alone "won't surface any tools that are useful for coding agents — instead, you need to provide tools to the MCP server." What is built in is only the infrastructure for finding and selecting a session. The capability is something you supply.

That is a design choice worth admiring rather than working around. It means the question "what can this agent do to my R session?" has an answer you wrote, rather than an answer you inherited.

`btw`'s wrappers supply the toolset, and the five groups named in the command above are btw's own documented recommendation for Claude Code: documentation lookup, package information, inspection of objects in your environment, session and platform metadata, and CRAN queries. Calling `btw_tools()` with no arguments would register more than that, including file, git, and shell groups — they are left out here because Claude Code already brings its own, and duplicating them adds capability without adding ability.

Note what that list does *not* include. The agent can look up documentation and describe objects already sitting in your session; it cannot execute new R code against them. Running code is a separate tool, `btw_tool_run_r()`, not enabled by default in any of btw's entry points and requiring a deliberate opt-in — an option, an environment variable, or adding `'run'` to the groups above.

That tool is worth knowing about even if you leave it off, because it is more capable than its name suggests. It evaluates in your global environment, so state persists between calls, and it captures not only printed output, messages, warnings, and errors but *plots* — a `ggplot` comes back as an actual image the client displays, not a file path. It also carries its own written operating rules, instructing the model to work incrementally, to make at most two attempts at a failing error before stopping to explain, and not to write files, run shell commands, install packages, or make network requests without showing you the code first.

Those rules are a prompt rather than a sandbox, so they are guidance the model usually follows rather than a boundary it cannot cross. Treat the grant as real: this is your session, with your filesystem and your network.

That distinction is worth holding onto, because it is easy to blur when describing this to a colleague. "The agent can see my session" and "the agent can run things in my session" are different grants, and the gap between them is exactly the least-privilege judgement this deck argued for earlier about MCP scopes — arriving here in a place where you hold the pen. An execution tool can do anything your R session can do, which includes your filesystem and the network.

There is a practical consequence for anyone reproducing this. Because the default toolset inspects rather than computes, materialise what you want asked about *before* you start: the agent can describe a data frame that already exists in your environment, but it cannot build one for you. Compute the summary table first, then ask about it.

Even so, for anyone who has spent an afternoon with a large object in memory that took twenty minutes to construct, the appeal is immediate. You can interrogate it in English, in the state you have got it into, without rebuilding it in a subprocess.

Custom tools are ordinary `ellmer::tool()` objects passed to `mcp_server()`, which is where you would expose a Bioconductor package's own functions as agent-callable operations. Servers can also be deployed to Posit Connect over HTTP.

One naming note that will save you a confusing search: this package was previously called `acquaint`, and the dependency direction has since flipped. Tutorials referencing `acquaint::mcp_server()` are describing what is now `btw::btw_mcp_server()`. The concepts carry over; only the call names changed.

Two honest caveats before you rely on this. The package is very new — version 1.0.1 reached CRAN days before this session — so expect rough edges. And there is a known failure mode worth recognising rather than debugging blind: the session mode communicates over a Unix domain socket, and endpoint-security software of the kind installed on managed institutional laptops can block it. The symptom is not an error, it is a tool call that never returns. If it hangs silently on a work machine, suspect that before suspecting your setup.

Registering it — one word apart

```
# read-only: docs, packages, objects in your session, session info, CRAN
TOOLS="list('docs','pkg','env','sessioninfo','cran')"

# ...and the same thing that can execute R and hand back plots
TOOLS="list('docs','pkg','env','sessioninfo','cran','run')"
```

```
# Claude Code
claude mcp add -s user r-btw -- Rscript -e "btw::btw_mcp_server($TOOLS)"

# Codex
codex mcp add r-btw -- Rscript -e "btw::btw_mcp_server($TOOLS)"
```

Everything after `--` is parsed by **your shell**, not the tool — so double quotes outside, R's single quotes inside, and nothing needs escaping.

Speaker notes

The read-only and the executing server differ by one string in a list. That is worth showing side by side, because it makes the grant legible: you are not choosing between “R integration on” and “off,” you are choosing whether the agent may run things.

Registering it is a single command in the two CLI-driven tools. The quoting looks alarming and is not, once you know the rule: everything after `--` is parsed by your shell and handed to the tool as an already-split argument list. The tool does not re-parse it. So double quotes on the outside protect the whole R expression from the shell, and R’s own single quotes inside are never seen by the shell at all.

The failure mode to warn a room about is not the syntax, it is the transport. Paste one of these through a slide deck, a chat window, or anything else that applies smart quotes, and the straight double quotes become curly ones. The shell does not recognise curly quotes as quoting, so the command breaks in a way that reads as a package problem rather than a typography problem. If a command copied from a presentation fails, retype the quotes before debugging anything else.

A note on confidence, since this deck is also a handout. The Claude Code form was run on the machine these slides were built on and the server reported connected. The Codex form is taken from its documentation and was not executed, because Codex is not installed here — treat it as accurate but unverified, and check `codex mcp list` after running it.

…or it is just a file

Antigravity and Copilot have no `mcp add` command. Same server, written down:

```
{
  "mcpServers": {
    "r-btw": {
      "command": "Rscript",
      "args": ["-e", "btw::btw_mcp_server(list('docs','env','run'))"]
    }
  }
}
```

- **Antigravity** → `~/.gemini/config/mcp_config.json` · `.agents/mcp_config.json` per project
- **Copilot** → `~/.copilot/mcp-config.json`
- **Codex** → same structure, TOML: `[mcp_servers.r-btw]` in `~/.codex/config.toml`

Removing it: `claude mcp remove r-btw` · everywhere else, delete the entry — or set `"disabled": true` to keep it but inert. Deleting the block always works.

Speaker notes

Two of the four tools have no registration command at all. They read a JSON file, and the file is the same `mcpServers` object in each — the same shape as the workshop `.mcp.json` from earlier in this session.

The tool list in the example above is shortened to three groups to keep the line on the slide; use the full five from the previous slide in a real file, adding `'run'` only if you want execution.

That repetition is the point worth drawing out. A server definition is a command and a list of arguments; every one of these tools wants exactly that, and mostly in the same JSON. Codex is the odd one out only in file format, expressing the identical structure as TOML. So “which agent do you use” turns out to matter far less than it appears to when you are choosing one.

There is a practical argument for preferring the file even where a command exists. A file has no shell in the middle of it, so the quoting problem from the previous slide disappears entirely — JSON and TOML care about double quotes and are indifferent to the single quotes inside our R expression. A file can also be committed, reviewed, and handed to a collaborator, which a command someone ran once cannot.

Removal is deliberately stated as deleting the entry rather than as a command per tool, because that is the claim that holds everywhere. Claude Code’s `mcp remove` was run here and reports which scope it removed from. Codex is reported by several third-party write-ups to have `codex mcp remove`, but its official documentation describes only hand-editing `config.toml`, and that disagreement could not be resolved without a live install — so the slide gives the form that works either way. Both file-based tools also support leaving an entry in place but disabled, which is useful for a server you want once a month rather than at every startup.

Confidence, because this is a handout and these details rot. Three things were confirmed on the machine these slides were built on: Claude Code’s registration, which connected; Copilot’s CLI genuinely having no `mcp` subcommand, only `login`, `help`, `init`, `update`, `version`, and `plugin`; and Antigravity’s config path, since `~/.gemini/config/mcp_config.json` exists here and `~/.gemini/` also holds `antigravity`, `antigravity-cli`, and `antigravity-ide` directories. That shared tree surprises people who go looking under `~/.antigravity` and find nothing.

Codex is the one entry taken from documentation alone, since it is not installed here. If its path is wrong, the structure above is still right — only the location would differ. One further Antigravity note worth knowing: its `/mcp` overlay is for viewing status and reloading, not for adding or removing, so it is not the management interface it looks like.

Part 3 — Sub-agents

Four primitives, one table

	What it is	Reach for it when
MCP	plumbing to an external system	the agent must *touch* real data
Skill	procedural knowledge, loaded on demand	you keep pasting the same steps
Sub-agent	isolated worker, own context window	a side task would flood your context
Slash command	a skill you invoke, not one that fires	side effects need deterministic timing

They compose: a skill can *run as* a sub-agent (`context: fork`); a sub-agent can *preload* skills.

Speaker notes

The four mechanisms are routinely confused, and the confusion produces the characteristic mistake of forcing everything into whichever one you learned first — writing a skill to do what MCP does, or pasting instructions repeatedly where a skill belongs.

The distinguishing question for each is different. For MCP: does the agent need to touch something outside its own reasoning? For a skill: am I repeating the same multi-step instructions? For a sub-agent: would this side task fill my context with material I will never look at again? For a slash command: do I need to control exactly when this happens?

The last one deserves emphasis because it is the newest. Slash commands are now formally skills that you invoke rather than ones the agent triggers. For anything with side effects — deploying, committing, submitting — that is what you want, and it is a frontmatter setting rather than a prompting trick.

The composition primitives are worth knowing because they are not symmetric. A skill with `context: fork` in its frontmatter runs its body as a sub-agent's task, in the background by default: that is "perform this procedure somewhere else and tell me the outcome." A custom sub-agent can instead list skills in its frontmatter, which injects their full content at startup as standing reference material: that is "give this specialist some background before it begins." One sends work out; the other sends knowledge in.

Sub-agents buy you context

A sub-agent has its own context window, system prompt, tool access, and optionally its own model.

- **Context preservation** — search results and logs never enter your main thread
- **Enforced constraints** — narrower tool access than your session has
- **Parallelism** — several investigations at once
- **Cost control** — route grunt work to a cheaper, faster model

The canonical case: *“read these forty files and tell me which three matter.”*

Speaker notes

The benefit that justifies sub-agents is not that they are smarter. It is that their debris stays out of your conversation.

Consider searching a large codebase for where a behaviour is implemented. Done in your main session, that fills the context with file contents, grep output, and dead ends you will never consult again — and everything afterwards is reasoned about through that clutter. Done in a sub-agent, the same search happens in a separate window and returns a paragraph. The exploration cost is paid once and discarded.

The other three benefits are real but secondary. A sub-agent's tool access can be narrower than your own, which is a genuine safety property when you want something investigated but not modified. Several can run at once, which suits independent questions. And a sub-agent can be routed to a cheaper model, which makes mechanical work cheap while your main session stays on a stronger one.

The failure mode to name is over-delegation. A sub-agent starts with no context from your conversation, so anything it needs must be in its prompt. Work that depends on the discussion you have been having transfers badly. The rule of thumb: delegate work that is *wide but shallow* — much material, simple question — and keep work that is narrow but deep.

Part 4 — Collaborating with an agent

Speaker notes

A deliberate boundary. Everything up to here has been about what you *attach* to an agent — procedures, connections, workers. Those are components, and you install them once.

What follows is not a component. It is how the two of you work, and it is where most of the difference between people who find these tools transformative and people who find them mildly useful actually lives. None of it requires installing anything. All of it is a habit.

That is worth saying out loud to a room of engineers, because the instinct in a session like this is to leave with a list of things to install. The install list is the smaller half. A developer with no skills, no MCP servers and a disciplined habit of specifying before implementing will outperform one with a loaded configuration and none.

The six habits here are ordered roughly by how much they change per unit of effort, and the first is by some distance the largest.

Spec before code

Unguided one-shot attempts succeed roughly **a third** of the time. Writing the plan first collapses the ambiguity the agent would otherwise guess at.

The pattern, in three moves:

1. Ask the agent to **interview you** until it has covered implementation, edge cases, and tradeoffs
2. Have it write `SPEC.md`
3. Start a **fresh session** and implement from the spec

Speaker notes

The single highest-leverage habit, and the one most resisted, because it feels like overhead in front of a tool that appears ready to start immediately.

The underlying mechanism is straightforward. A one-line request leaves a great many decisions unspecified, and the agent resolves every one of them by guessing. Most guesses are individually reasonable; the compound probability of all of them matching what you wanted is what produces the roughly one-in-three success rate for unguided attempts. Writing a plan first does not make the model cleverer. It removes the guesses.

The inverted move is what makes this practical: rather than writing a specification yourself, have the agent interview you. It asks about implementation, interface, edge cases, and tradeoffs until the ambiguity is gone, then writes the specification. Answering pointed questions is far easier than anticipating which details will matter, and the questions surface decisions you had not noticed you were making.

The third step is the one people skip and the one that most affects quality. Implement in a *fresh* session, from the specification. The interview conversation is long, full of rejected alternatives and half-formed ideas, and all of that is context competing with the actual instructions. The specification is the distillate. Hand over the distillate, not the transcript.

For a package developer the artifact has a second life. `SPEC.md` is a design document — reviewable by a collaborator, checkable against the implementation afterwards, and a record of why the interface is what it is.

Grill me

From Matt Pocock's collection — an agent that interrogates *you*, and refuses to start until you are done.

- restate the plan as it understands it
- name the highest-risk unknowns
- ask **one question at a time**
- **inspect the repository to answer its own questions first**
- offer a recommended answer with tradeoffs
- track decisions as accepted, rejected, or unresolved
- **refuse to implement until asked**

Speaker notes

An agent’s default disposition is agreeable, and agreeableness is precisely wrong at the design stage. “Grill me” inverts that on purpose.

The procedure has a documented shape, and two of the steps carry most of the value. Asking one question at a time sounds trivial and is not — a list of eight questions gets eight cursory answers, while one question at a time gets considered ones, and each answer reshapes what should be asked next.

The better move is the fourth: the agent inspects the repository to answer its own questions before spending one on you. If the answer is discoverable from the code, it should not be a question. That single constraint is the difference between an interrogation that feels like an interview and one that feels like being asked to read your own codebase aloud.

The last step matters for the same reason the earlier “report, do not fix” instruction did. Left unconstrained, an agent will start implementing partway through the conversation, which ends the design discussion prematurely and usually in the wrong place.

For this audience the natural target is a real design decision you are sitting on — an S4 class hierarchy, whether to add a dependency, how to deprecate an argument without breaking downstream packages. Those decisions have consequences measured in years of maintenance, and they are exactly where thirty minutes of being interrogated pays for itself.

Review the diff, not the chat

The session that wrote the code is **too close to its own reasoning** to review it honestly.

So review somewhere else:

- a **fresh context**, given only the diff and the spec
- told to report **correctness gaps only** — not style
- because a reviewer asked to find problems will always find some, and chasing all of them produces over-engineering

Speaker notes

The instruction “review your work” addressed to the session that just did the work is close to useless, and the reason is structural rather than a limitation of any particular model.

That session holds the entire chain of reasoning that produced the code, including the justifications for every decision. Asked to evaluate the result, it evaluates it against the intentions it already holds — and against those intentions the code usually looks correct. What a reviewer needs to see is what the code *does*, not what it was meant to do, and that information is contaminated by everything else in the window.

The fix is to review in a fresh context, handed the diff and the specification and nothing else. That reviewer sees only what a human reviewer sees. This is what sub-agents are for, and it is among the best uses of them.

The scoping instruction is as important as the isolation. A reviewer told to find problems will find problems, indefinitely, because there is always another edge case and another abstraction that could be cleaner. Told to report correctness gaps only — where does this do the wrong thing — the output is short and actionable. Left open, it produces a list of stylistic suggestions which, implemented in full, leave the code more complicated than it started.

For package maintainers the fit is close to what you already do. Bioconductor review is a fresh reader working from the submission, not from the author’s account of the submission. The same logic applies one level down.

First: what is an ADR?

An **architecture decision record** — one short file per consequential decision.

Context (what forced a choice) · *Decision* (what you chose) · *Consequences* (what it costs you)

For **humans**, it answers the question code cannot: **why is it like this?** Git tells you what changed. An ADR tells you what was rejected, and what would have to change for the answer to be different.

For **agents** it does three jobs at once — **memory** that survives the end of a session · **guardrails** on code that looks wrong and is not · **decisions** it should not silently reopen.

Speaker notes

Worth defining properly, because the term is common in some engineering cultures and unheard-of in others, and this room will be split.

An architecture decision record is a short Markdown file — a page, rarely more — written once per decision that would be expensive to reverse. The conventional skeleton is three headings. *Context*: what situation forced a choice. *Decision*: what you chose, stated in the active voice. *Consequences*: what you now have to live with, including the bad parts. They are numbered, appended rather than edited, and superseded rather than deleted, so the record of a decision you later reversed stays readable.

The human case is straightforward and predates agents by a decade. Code records what you decided; it cannot record what you rejected or why. Six months later nobody remembers whether the awkward interface was a considered tradeoff or an accident, and in the absence of a record the safe assumption is accident — so it gets tidied, and the reason it existed is rediscovered the expensive way.

The agent case is the same problem with the clock sped up. Three jobs, and they are genuinely distinct. **Memory**: a session starts with no knowledge of any conversation you have ever had, so an agreement reached last week exists only if it was written down. **Guardrails**: an agent is confident and tidy-minded, and code that looks wrong is exactly what it will helpfully correct — the ADR is what tells it the ugliness is load-bearing. **Decisions**: without a record it will re-litigate settled questions every time it touches the area, and argue each one plausibly.

For a Bioconductor package the candidates are easy to name: why a class extends `SummarizedExperiment` rather than wrapping it, why a dependency was taken or refused, why a deprecated argument is still accepted. All decisions with multi-year consequences, and all invisible in the code itself.

ADRs an agent will actually obey

An agent tidying “ugly” code has no way to know the ugliness was a deliberate response to an incident. Write decisions down — but not the way you write them for people.

- **imperative** language — MUST, MUST NOT, not “we generally prefer”
- an `applies_to` **glob**, so irrelevant decisions never load
- a **mechanical verify command** — a grep or a lint rule, not only prose

This repository’s own `docs/adr/` is the worked example: ten decisions, each with a `talks doctor` check enforcing it.

Speaker notes

Every long-lived codebase contains code that looks wrong and is not — a workaround for a platform bug, a deliberate inefficiency protecting against something worse, an interface kept awkward for backward compatibility. A human contributor learns these by being corrected in review. An agent has no such memory, and will helpfully undo them.

Architecture decision records solve this, but records written for agents want a different shape than records written for people. Three differences matter.

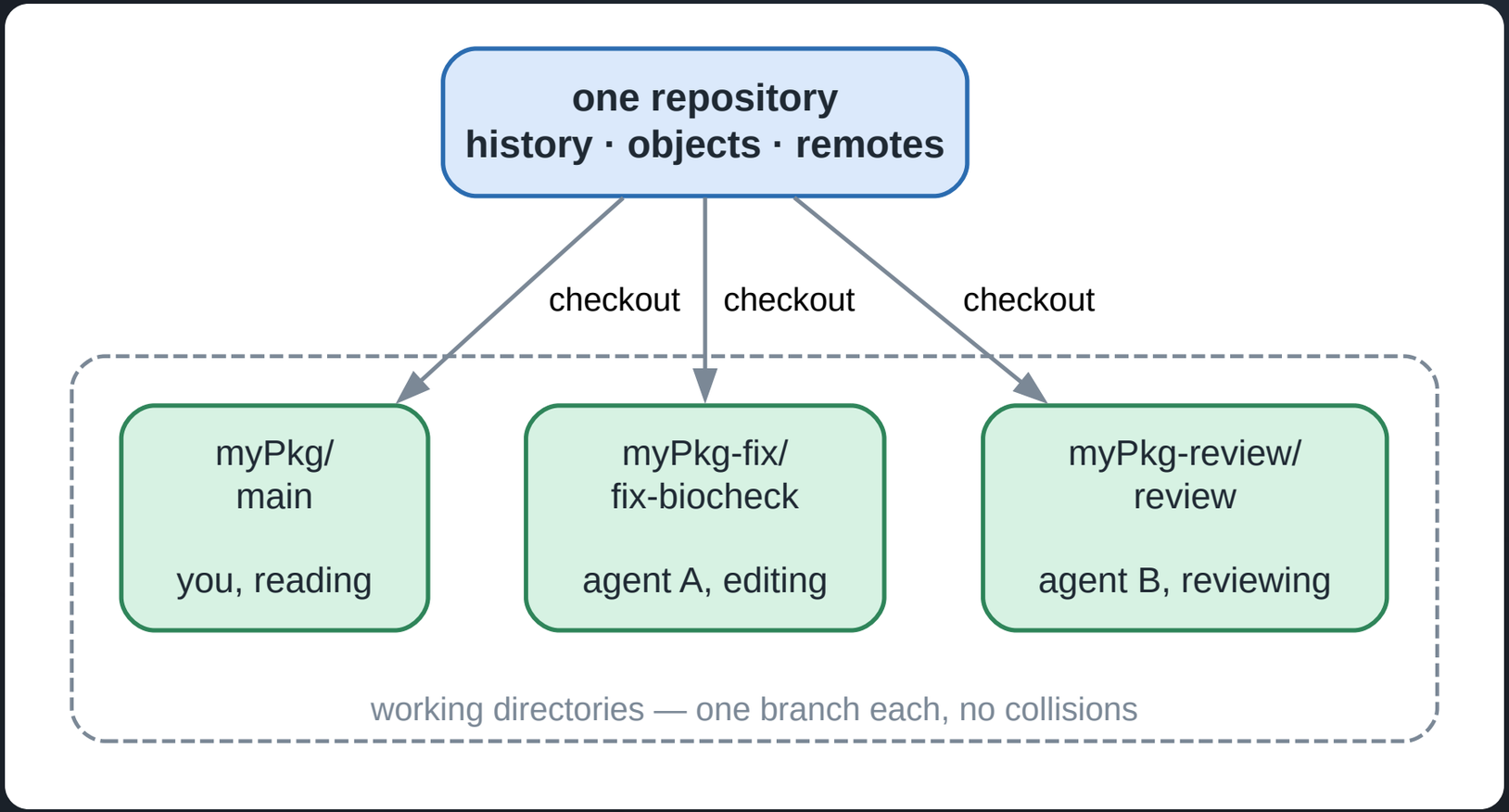
Use imperative language. “We generally prefer” is how humans write and is readable as advisory; an agent weighing it against other considerations may decide the other considerations win. MUST and MUST NOT do not have that ambiguity.

Scope each record to the files it governs with a glob, so a decision about database access is not loaded while someone edits documentation. This is the same context economy that makes skills affordable.

Give each one a mechanical check — a grep, a lint rule, a test — rather than prose alone. Prose can be reasoned around; a failing check cannot. This is also the difference between a decision that holds and a decision that quietly erodes over a year.

The talks repository these slides live in is the worked example: ten architecture decision records, each paired with a check in the build’s `doctor` command, so a violation fails the build rather than being noticed later or not at all.

First: what is a worktree?



Not a second clone. **One** repository and history, checked out into **several directories at once** — each on its own branch.

Speaker notes

Most people have never used one, so this is worth a minute rather than a sentence.

The wrong mental model, and the one almost everyone arrives with, is that a worktree is a second clone. It is not. `git worktree` gives you an additional *working directory* attached to the repository you already have: one `.git` store, one history, one set of remotes, several checkouts. Nothing is duplicated, there is no second fetch, and a commit made in one worktree is immediately visible to the others because they share the same object store.

Without agents this is a modest convenience — a way to look at `main` without stashing what you are doing. With agents it becomes structural, because a second agent working in the same directory as the first is not a nuisance, it is a correctness problem. Two processes editing the same files interleave their changes, and the resulting mess is hard to attribute: neither agent did anything wrong, and both produced the wrong result. Branches do not help, because a branch is a pointer and the working directory is still shared.

The pairing worth internalising is the one from the review slide. One worktree implements; a second, on its own branch, reviews. That is what makes the isolation genuine rather than nominal — the reviewer is not merely a fresh context, it is looking at a checkout the writer cannot be modifying underneath it.

They are cheap to make and cheap to discard. Create one for a piece of work, delete it when the branch merges. The main cost is disk, which for an R package is nothing.

Worktrees, and codifying the loop

```
claude --worktree fix-biocheck     # isolated checkout, own branch
```

Two agents, two worktrees, no collisions — the classic pairing being one writing and one reviewing.

Then encode the whole loop so it runs the same way every time: **worktree → small commits → PR → automated review → fix → merge on green.** `CLAUDE.md` for advice; **hooks** when it must be enforced.

Speaker notes

Worktrees solve a problem you meet as soon as you run more than one agent: two sessions editing one checkout collide, and the failure is confusing because neither is doing anything wrong.

A worktree is a second checkout of the same repository on its own branch, sharing the same history. Two agents in two worktrees cannot interfere. The pairing that motivates this most often is the one from the previous slide — one session implementing, another reviewing in genuine isolation rather than nominal isolation.

The larger point is that once a workflow works, it should stop depending on you remembering it. The loop most teams converge on is a worktree per unit of work, small commits throughout, a pull request, an automated review pass, fixes, and a merge only on green. Written down, it is repeatable and delegable. Left as habit, it degrades under deadline.

There is a real distinction in how you encode it. Instructions in `CLAUDE.md` are advisory — read, weighed, usually followed, occasionally not. Hooks are deterministic: they run on defined events and can block the action outright. If a step is a preference, write it down. If it is a rule that must not be skipped — never commit to main, never push without tests passing — a hook enforces it and prose does not.

Small commits deserve a note of their own, because their value changes with agents in the loop. A commit message is context for whoever comes next, and increasingly that is an agent reconstructing why the code is as it is. Frequent, well-described commits are a better substrate for that than a weekly squash.

Prune your `CLAUDE.md`

For every line, one question:

> **Would removing this cause a mistake?**

If not, cut it.

Keep: the commands an agent cannot guess (`devtools::load_all()`, `BiocCheck::BiocCheck(".")`, how your tests are actually run), the constraints that look wrong but aren't, the conventions specific to *this* package.

Cut: standard R idiom, anything derivable from the code, aspirational style guidance nobody enforces.

Speaker notes

A closing exercise that transfers immediately, because everyone in the room owns a file this applies to — or will within a day.

`CLAUDE.md` is loaded in full at the start of every session and stays resident throughout, so every line is a permanent tax on the context available for actual work. The instinct on first writing one is to be thorough, which produces a long document restating things the agent already knows and diluting the few lines that matter.

The test is a single question applied line by line: would removing this cause a mistake? Not “is this true,” not “is this good practice” — would its absence produce a wrong action. Most lines fail that test.

What survives, for an R package, is fairly specific. The commands an agent cannot guess: that tests run via `devtools::test()` rather than whatever it would assume, that this package needs `BiocCheck` and not only `R CMD check`, any non-obvious build step. Constraints that look like mistakes and are not. And conventions particular to this package rather than to R generally.

What goes is everything the agent already knows — that R uses `<-`, what roxygen comments are, standard idiom — along with anything discoverable from the code itself, and aspirational style guidance nobody actually enforces, which teaches the agent that this file can be disregarded.

Concision here is not tidiness. It is the difference between guidance that is followed and guidance that is averaged away among fifty other lines.

Takeaways

- A **skill** is a folder with a `SKILL.md` — and now an **open standard**, read by forty-plus tools
- The **description** carries the entire triggering signal; write the words you would actually say
- `--scope project` puts MCP servers in the repo, so tooling ships with the code
- **R can serve MCP** — an agent can inspect the session you already have open
- **Specify before implementing**, and **review in a fresh context**
- Bioconductor already has `ai-agent-skills` — and no policy on any of this yet

Speaker notes

Six things worth keeping.

Skills are simpler than expected and now portable across vendors, which means procedures encoded today are not a bet on one tool. The description is the whole triggering mechanism, so it deserves more care than the body. Project-scoped MCP configuration turns agent tooling into a property of the repository rather than of a developer’s laptop, which matters for reproducibility as much as convenience. R can serve MCP, and the live-session mode is a genuinely different capability rather than a convenience. Specification before implementation and review in a separate context are the two habits with the largest measured effect on outcomes.

The last point is the one for the afternoon. This community already has an active skills repository and already has agent-assisted commits in core infrastructure, attributed by an emerging convention that nobody has ratified. What it does not have is any statement in its contribution guidelines about disclosure, review, or what a maintainer owes users when agent-written code ships — while a sibling community published exactly that months ago.

That is not a gap this session can close. It is a gap the people in this room are the right ones to close, and it is a better use of the focus group than anything prepared in advance.

Resources

Skills — agentskills.io · code.claude.com/docs/en/skills · skills.sh · obra/superpowers · mattpocock/skills · anthropics/skills

MCP — code.claude.com/docs/en/mcp-quickstart · biocontext.ai/registry · biomcp.org · pdbe-mcp-servers · gget-mcp

R — posit-dev.github.io/mcptools · ellmer.tidyverse.org · posit-dev.github.io/btw

Bioconductor — Bioconductor/ai-agent-skills

All links verified 2026-07-30. Slides and a notes handout: talks.seandavis.net

Speaker notes

The reference slide, for afterwards rather than for now.

If you take three actions from this session, the order that gets you furthest fastest is: install one collection and watch a skill fire on your own code; write one project skill for your own package and commit it; and add one MCP server at project scope so your collaborators inherit it.

The R links are the ones least likely to be found by searching, because the ecosystem is young and the naming has shifted — `mcptools` was `acquaint`, and `ellmer` underpins most of the rest. Start at `mcptools` if the live-session demonstration was the part that interested you, and at `ellmer` if you want to build agent behaviour into a package rather than drive it from an editor.

Everything here was checked the day before the session. In this ecosystem that matters more than usual: several of these projects changed names, maintainers, or install commands within the last six months, and a fair amount of the writing about them online is describing a state of affairs that no longer holds.