Data Hub: Automated capture and processing of lab instrument data

Data Hub is a lightweight, open-source system that captures data from lab instruments and makes each run findable and reusable through three interfaces — a web app, a REST API, and a Model Context Protocol (MCP) server for AI agents.

Published Jul 30, 2026

Arcadia Science

DOI: 10.57844/arcadia-kdw3-emfz

Purpose

Scientific instruments generate a lab's valuable raw data, but that data is often confined to the PC attached to each instrument, where it's hard to systematically find, compare, or reuse, and nearly impossible for most software to reach. We built a self-hosted, open-source system called Data Hub to capture data from lab instruments and standardize runs through one data model and three interfaces: a web app for researchers, a REST API for pipelines and scripts, and a Model Context Protocol (MCP) server for AI agents.

A small watcher on each instrument PC detects new files, groups them into runs, and uploads them to cloud storage, where supported instruments are processed automatically as files arrive. Once a run is captured, a scientist can view it in a browser, a pipeline can make it accessible via an API, an AI agent can query it in plain language, and downstream automation can be triggered elsewhere.

We've intentionally scoped Data Hub quite narrowly. It captures and serves instrument data, rather than acting as a laboratory information management system (LIMS), so it complements the systems a lab already runs. It also makes data capture deeper and more structured — recording rich metadata, provenance

from instruments, user attribution, and more — which ultimately helps make publicly shared datasets reproducible and reusable.

Data Hub is currently in production at Arcadia Science, where it has captured roughly 2,500 runs across 10 instruments, totaling ~1.5 TB of data in just the last year and costing roughly $100 per month to host. We're releasing it as open source for scientists to use, adapt to their own instruments, or build on.

Motivation

A lab's instruments produce valuable raw data: Every plate read, gel image, and chromatogram is a measurement that someone may want to analyze, interpret, compare, or revisit. As biology becomes more data-driven and prospective [1], that data is worth most when it accumulates into a corpus that can be cumulatively learned from. For example, a cohesive data corpus would allow one to continuously compare runs across weeks, catch drift in assays, or ask questions that span many experiments. Getting there depends on the data being findable, comparable, and reusable in the first place.

That said, most labs aren't set up to make this seamless, and good data practices tend to take a lot of work. Each instrument writes files to the PC attached to it, and moving them to a useful location is manual: copied to a network drive, emailed, or carried off on a USB drive. Even once files are shared, a run's data and its metadata (which instrument, which settings, who ran it, when) drift apart, and nothing lists every run on an instrument or lets you compare them. A researcher can't readily find last month's runs for a strain, compare conditions across a fleet of instruments, or reproduce an analysis, and the lab's accumulated measurements never compound into an asset. The gap widens as analysis moves toward code and AI — a pipeline needs an endpoint it can call for the latest results, and an agent can only reason over data reachable through an interface. Files on a shared drive provide neither.

Systems for pulling data from lab instruments exist, but generally aren't simple, open source, AI native, and designed to be self-hosted. Commercial LIMS platforms are capable but are paid, hosted services, and general-purpose

ones are more complex than needed to capture instrument outputs. We also already run a LIMS alongside analysis pipelines, so we didn't need another system to do that work.

We designed Data Hub to serve as a narrow layer that captures files off instrument PCs, processes and indexes them, and exposes every run through one data model reachable by people, code, and AI agents. Because every run lands in one queryable place, a lab can do things that are impractical when data sits on scattered drives. Because Data Hub is open source and self-hosted, that record stays private until it's ready to share. By automatically recording rich metadata that scientists share along with their raw datasets, Data Hub makes research outputs easier to reuse and reproduce.

The rest of this pub describes how Data Hub works in more detail, including the various ways you can interact with it. Links to try it yourself are in the box below — we hope you'll give it a try and let us know what you think.

> The Data Hub **codebase** is on GitHub (DOI: 10.5281/zenodo.21707864), and the **documentation** is at datahub.arcadiascience.com/docs.

Data Hub in use

So what does it look like when scientists, scripts, or agents interact with Data Hub?

In the browser

The web app is where people work with runs. Sign-in is through single sign-on (SSO). A dashboard shows fleet health, recent activity, and per-instrument run counts (Figure 1); instrument pages list and filter runs (Figure 2); and a run page shows its files, processing status, and metadata, with comments and user attribution so a team can track who ran what (Figure 3).

Preprocessed results render in the page for seven supported instruments (the ones we run in the lab) — the Agilent 4150 TapeStation, Azure 600 Gel Doc, Azure Cielo qPCR, Epson V700 scanner, Ti2-E microscope, and SpectraMax iD3 and iD5 plate readers — so a scientist can view a result without downloading anything.

What renders depends on the instrument: a plate-reader run shows per-well values as a plate map (Figure 4), a gel run shows imaging mode and wavelengths pulled from TIFF tags, and a microscopy run shows a channel overlay built from its ND2 files.

Runs from instruments without a processor still get captured, indexed, and served — they just arrive without automatic metadata or derived results.

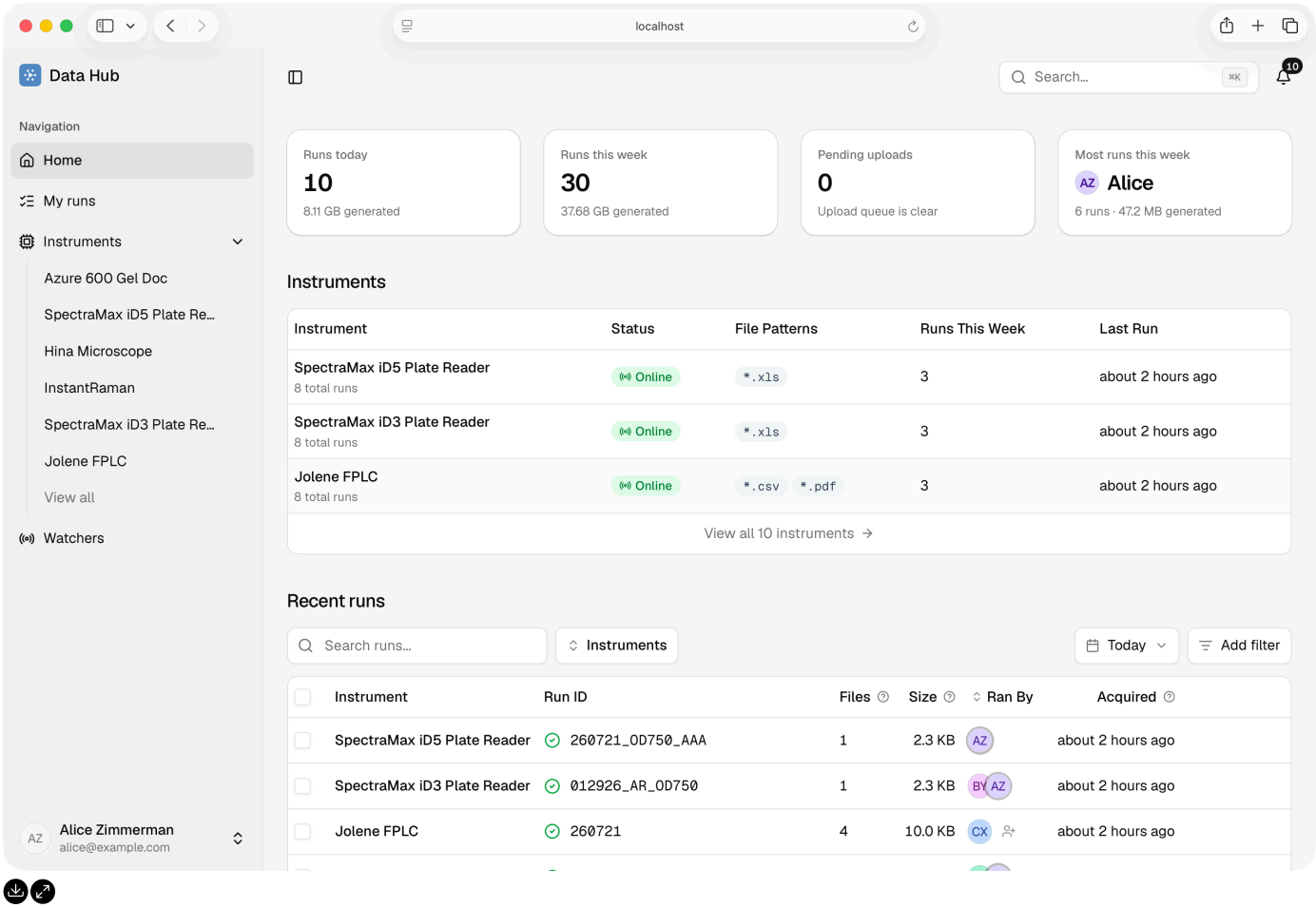

Figure 1. **The dashboard is the fleet-level view.**
Recent activity, pending uploads, and per-instrument status across every instrument reporting to the deployment.

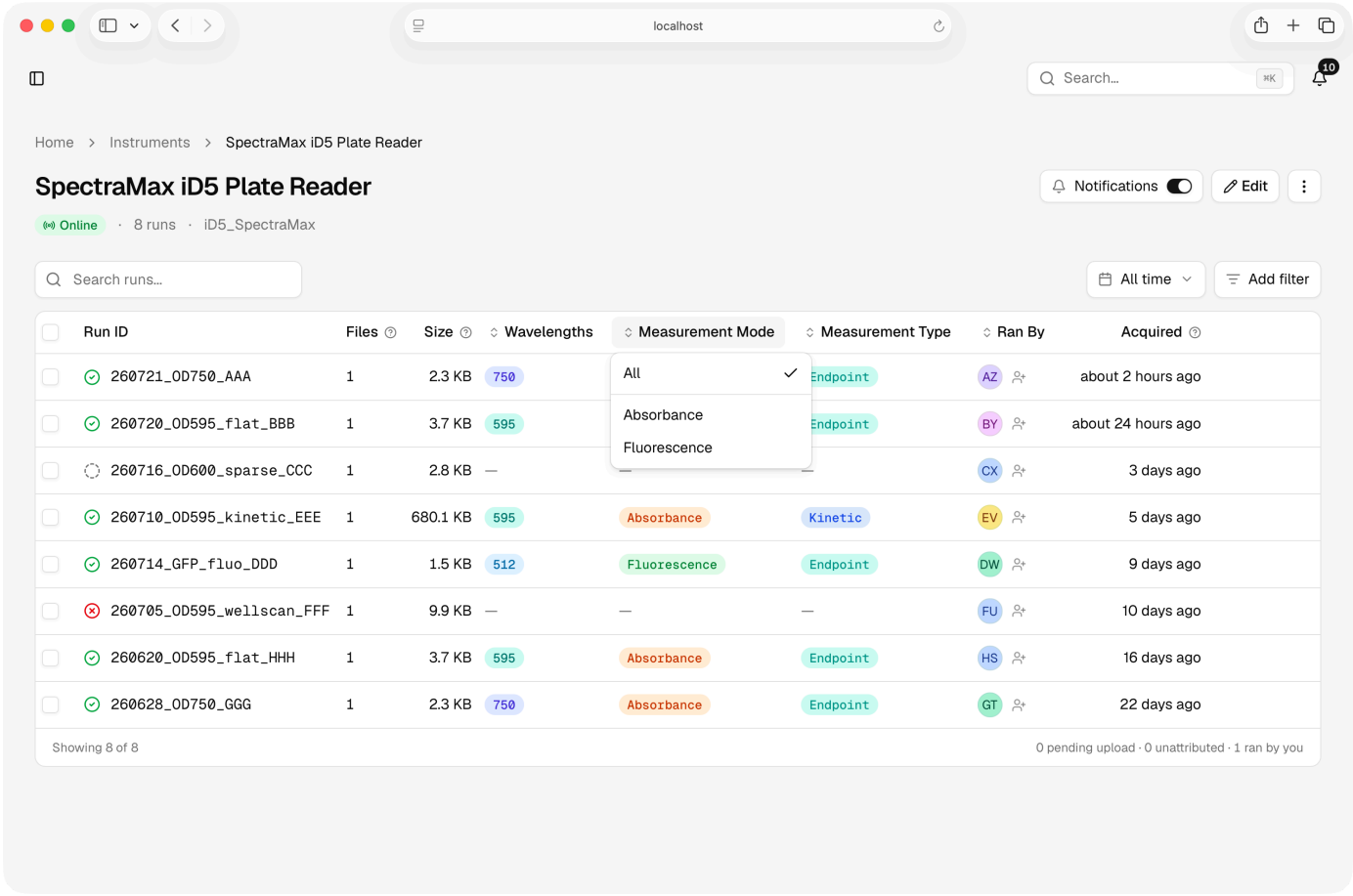

Figure 2. **Instrument runs can be searched and filtered.**

The instrument runs table lists each run with its files, size, and acquisition date, so a scientist can find and filter results quickly.

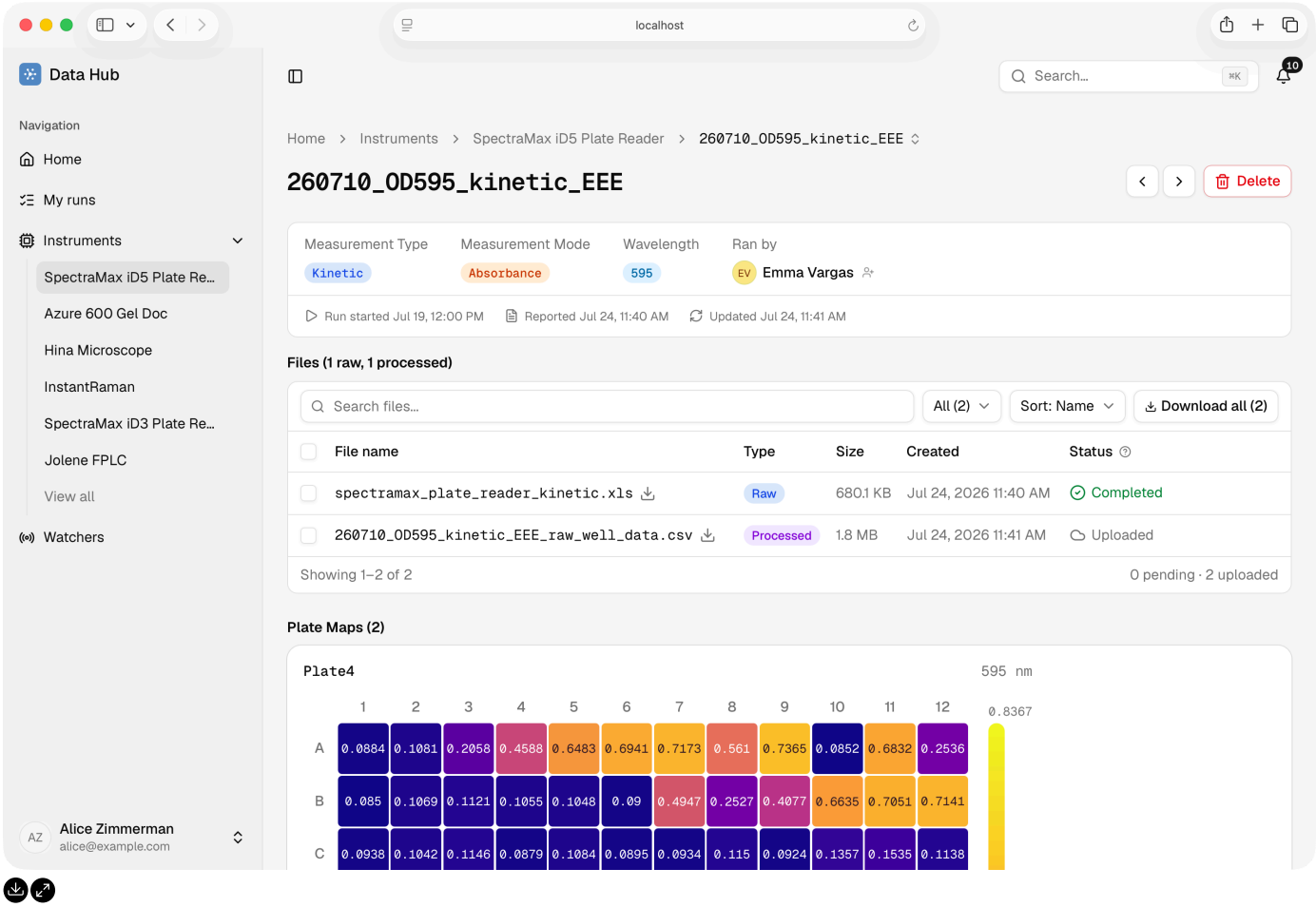

Figure 3. **Processed results render in the browser.**

A plate-reader run shows its raw and processed files and a per-well plate map, so a scientist can inspect a result in place.

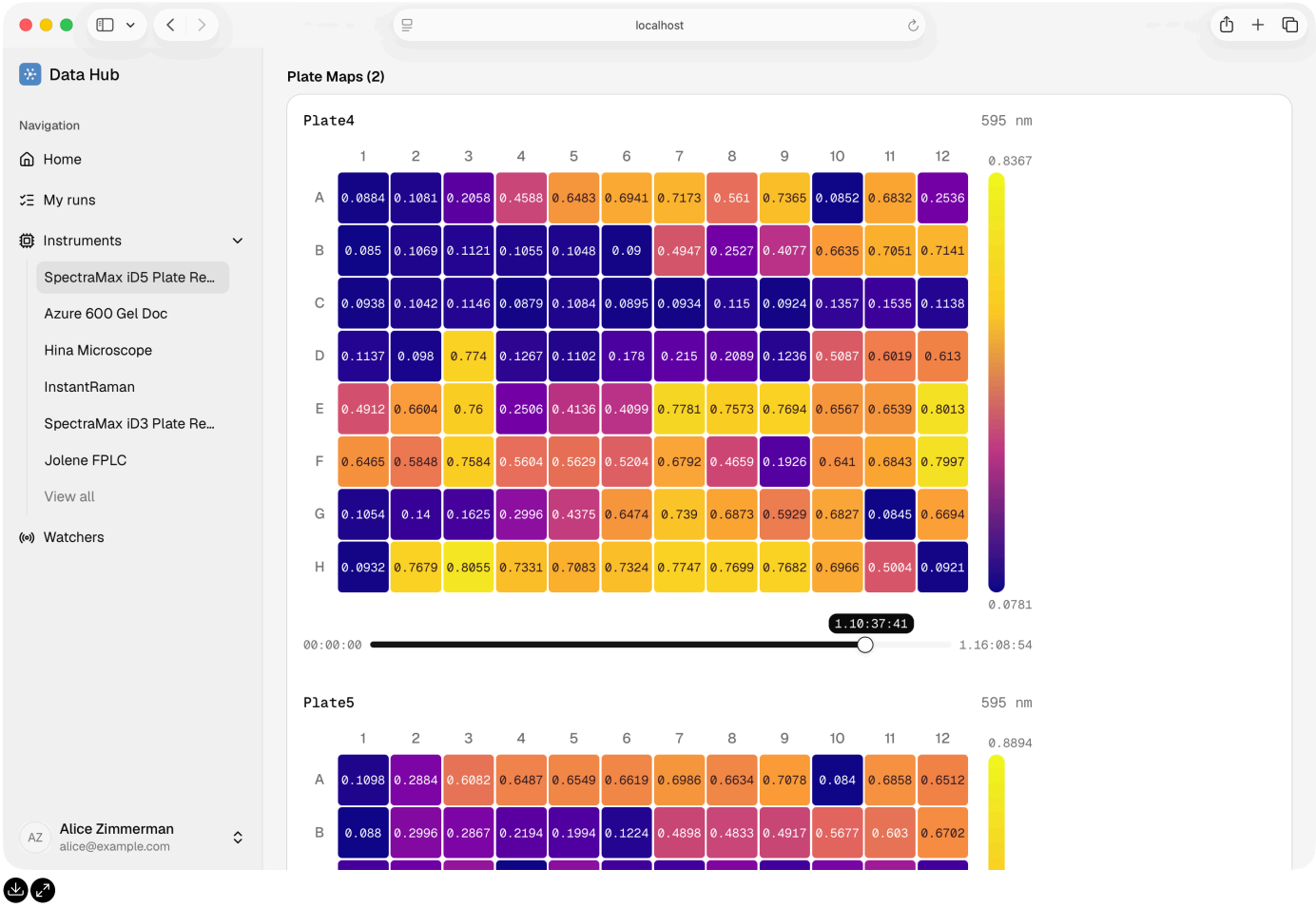

Figure 4. **Plate maps render with a color-scaled heatmap.**
Each well shows its absorbance value at 595 nm, with a time slider to scrub through kinetic reads, so a scientist can spot patterns across a plate.

Over the API

Everything the web app shows is available via the REST API at `/api/v1`, so pipelines and scripts can pull runs and files directly. Requests authenticate with a personal access token carrying least-privilege scopes (`runs:read`, `files:read`, and so on), and file downloads come back as short-lived signed URLs, so bytes stream from storage rather than through the app. A downstream analysis can list an instrument's runs, download the processed files it needs, and move on.

```
curl https://datahub.example.com/api/v1/instruments \
  -H "Authorization: Bearer dhub_your_access_token_here"
```

Tokens are scoped and are issued and revoked by admins, so an integration gets exactly the access it needs and nothing more.

For AI agents

The same web app serves an MCP server at `/api/v1/mcp` that any compliant client — Claude Desktop, Cursor, and others — can connect to with a token. Through it, an agent can list instruments, search runs with instrument-specific filters, fetch processed results, generate download links, and re-run processing (Figure 5). It enforces the same scope-based permissions as the REST API: a token's scopes limit what an agent can do exactly as they limit a script, so agent access is neither broader than nor separate from the rest of the system.

Alongside tools, the server exposes resources (grounding context such as the instrument catalog and each instrument's valid filter values) and prompts (scripted workflows such as a daily activity summary, an instrument troubleshooter, and a run-to-run comparison).

This is the second MCP server we've built for scientific work, after Agentic PyMOL [2], and it reflects the same idea: Give an agent a typed, permissioned interface to a scientific resource rather than leaving it to scrape a UI or shell out to scripts. In practice, a scientist can ask plain questions — "which plate-reader runs failed to process last week?" or "summarize yesterday's instrument activity" — and the agent answers by calling Data Hub, then works with the result in the same place it reads papers and runs analyses.

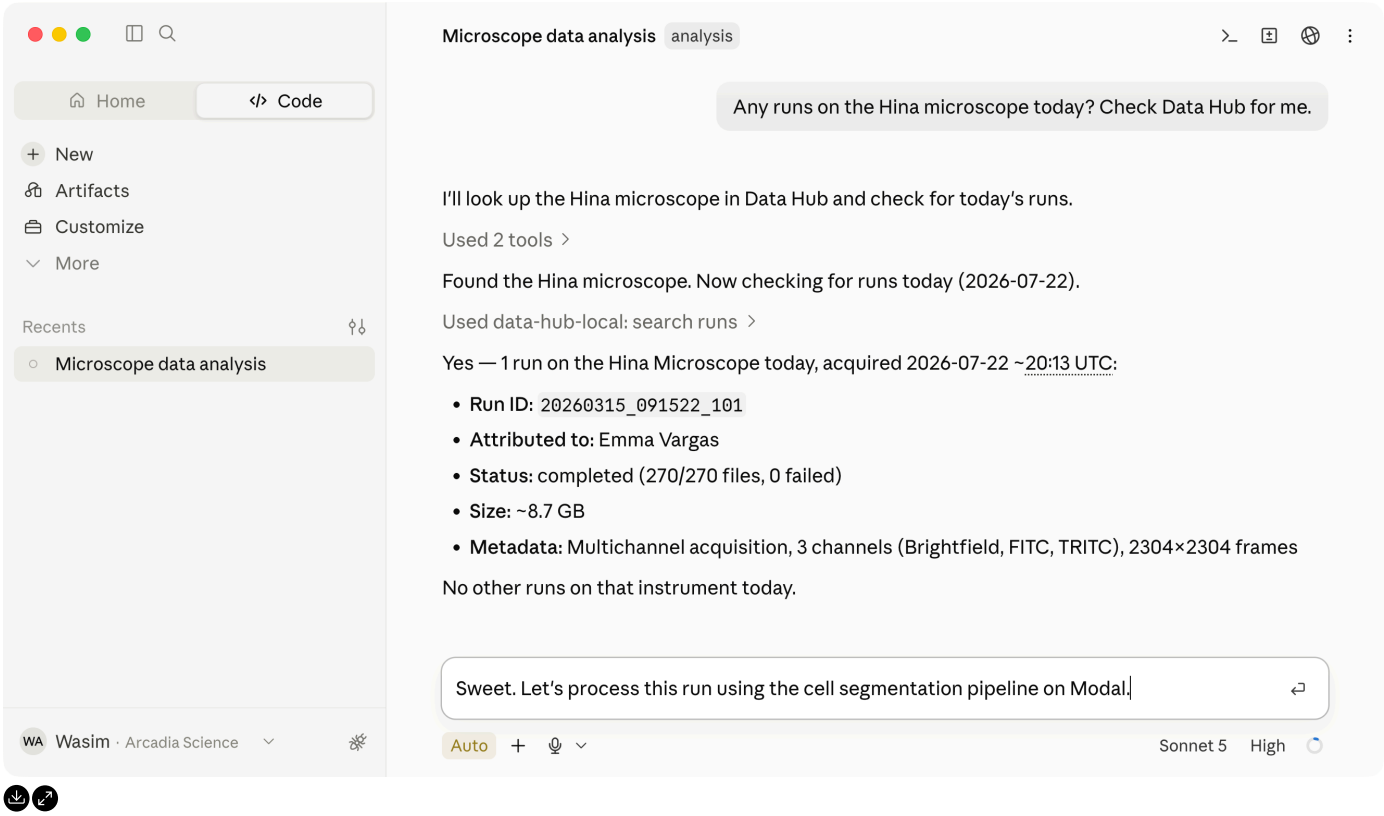

Figure 5. **An agent querying Data Hub over MCP.**
A client uses the Data Hub MCP tools to fetch an instrument run.

Triggering work elsewhere

Because Data Hub reports each new run to its API, a run can also push work outward: It posts to Slack when a run arrives, and a webhook can hand the same event to another system to kick off downstream analysis, quality control, or archiving. Data Hub stays the capture layer; the automation lives wherever a lab already runs it.

A worked example

Consider a scientist running growth assays on fission yeast (*Schizosaccharomyces pombe*), measuring optical density (OD) across strains and conditions on a plate reader. Over a few weeks, that's dozens of runs. Data Hub captures each plate read as it comes off the instrument and processes it into a per-well table, indexed and ready to work with.

Data Hub can make analysis easier too. To compare a knockout against wild type at a glance, the scientist opens both runs side by side in the browser. To fit growth curves, they point an existing pipeline or notebook at the API and pull every OD

run for those strains. Or, they could query an agent connected to the MCP server ("chart growth for these strains across this month's runs"), and the agent can retrieve the runs and plot them.

How Data Hub works

Data Hub has four components that coordinate through cloud storage, a REST API, and a shared library (Table 1). The shape of the system follows its job: capture files at the instrument, process them centrally, and serve them.

Component	Role
Watcher (`data-hub-watcher`)	A command-line agent on each instrument PC. Detects new files, groups them into runs, uploads them, and reports status.
Processor (`data-hub-lambda`)	A serverless function triggered by each upload. Runs instrument-specific processing.
Web app (`data-hub-web`)	Serves the web dashboard, the REST API, and the MCP server.
Shared library (`data-hub-shared`)	Common code: storage utilities, instrument definitions, and test infrastructure.

Table 1. **Data Hub components.**

The reference design is intentionally generic. In our own deployment, the web app runs on a managed application platform, cloud storage is an object store, the processor is a serverless function, sign-in uses a hosted identity provider, and state lives in a relational database. However, none of those choices are essential, and you could map the design onto equivalent services (Figure 6). In our case, we run the app on Vercel, store files in Amazon S3, process with AWS Lambda, sign in with Google, and use PostgreSQL.

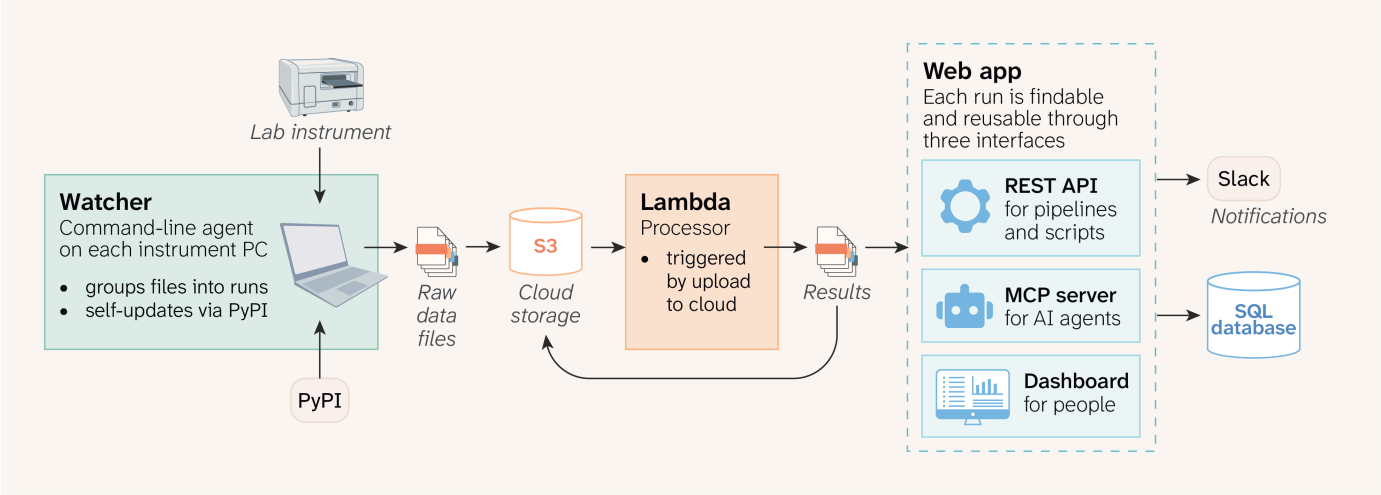

Figure 6. **Data Hub's data flow.**
The watcher uploads raw files to cloud storage and reports runs to the API. An upload event triggers the serverless processor, which processes files and writes results back through the API. The web app, REST API, and MCP server all read the same indexed data; file bytes move directly to and from storage over signed URLs.

In the common case, a file's path through the system runs as follows:

1. The instrument writes output to a watched directory on its PC.
2. The watcher detects each file once it's stable, groups related files into a run, and reports the run to the API.
3. The watcher uploads the raw files to cloud storage, keyed by instrument and run.
4. The upload triggers the processor, which — for supported instruments — extracts metadata, parses results, and generates derived files.
5. The processor writes the runs, files, and results back through the API.
6. The run appears in the web app and is reachable over the API and MCP.

Setting up an instrument requires that the instrument is connected to a PC you can install software on. An operator installs the watcher from the Python Package Index (PyPI) and runs a setup wizard that asks for the watch directory, which file patterns to upload, how to group files into runs, and whether uploads go out automatically or wait for approval.

A new instrument registers as pending until an admin confirms it in the web app and sets its instrument type — that type is what selects a processor. From there, the watcher runs as a background service so it survives reboots and logouts,

reports its health regularly so the dashboard can show which instruments are online, and updates itself when a new version is published.

Grouping is driven by a configurable rule that maps each file to a run, either by a shared filename prefix or by the folder a file lands in. A plate-reader run, for instance, groups its raw file and the table derived from it under one run identifier, so everything from a single acquisition stays together.

A few decisions keep the system decoupled and safe to run. Cloud storage is the integration boundary: the watcher writes and the processor reads, and the two never talk directly. File bytes never route through the app. Instead, it hands out short-lived signed URLs, so uploads and downloads (including whole-run archives) go straight to and from storage, and the watcher holds no storage credentials of its own. For instruments that need review before data leaves the building, a manual mode reports runs but holds the files until an admin releases them.

Open source and self-hosting

Data Hub is self-hosted: A team stands up the backend on its own infrastructure (a database, the web app, and the storage and processing pieces) following the deployment guide in the developer docs. Adding support for processing data from a new instrument is a straightforward change in the codebase, which keeps the core small while leaving room to extend it. It's open source under the MIT license, so other labs can run it as is, fork it for instruments or workflows we haven't covered, or contribute changes back.

If you start playing around with the code and run into any problems, please create a GitHub issue. If you have big-picture feedback or want to share how you've adapted Data Hub for your own work, please comment here! We'd love to hear your thoughts.

AI usage

We used Claude (Sonnet 5 and Opus 4.6 through 4.8) and Cursor with Composer 2.5 and Grok 4.5 to help write code, clean up code, comment our code, and review our code and selectively incorporated its feedback. We also used Claude (Opus 4.7 and 4.8) to suggest wording ideas, write text that we then edited, rearrange text we provided to fit the structure of one of our pub templates, expand on summary text that we provided before we then edited, help copy-edit draft text to match Arcadia's style, help clarify and streamline text that we wrote, and generate a mockup of Figure 6 prior to manually creating the final figure in Adobe Illustrator. All output was reviewed by humans and agents.

Key takeaways

Lab instrument data can often be confined to the PC that generated it and hard to reuse in the context of other research artifacts or documentation. Data Hub automates the path from instrument to a shared, queryable record: A watcher captures new files, cloud processing turns them into indexed runs with metadata and derived results, and every run is reachable through one data model.

That record is meant to be used by people in the browser, by pipelines over the API, by agents over MCP, and by downstream automation triggered on each new run. Data Hub complements a lab's existing systems rather than replacing them, and because it's open source and self-hosted, anyone can adapt it to their needs. By capturing instrument information and other metadata per run, once data is shared, it's more reproducible and reusable.

> The **codebase** is on GitHub, and the **documentation** is at datahub.arcadiascience.com/docs.

Contributors (A–Z)

- **Wasim Amiri**: Conceptualization, Software, Visualization, Writing
- **Audrey Bell**: Visualization

- **Ahmed Hosny**: Supervision, Validation
- **Peter Kohli**: Supervision
- **Ryan Lane**: Critical feedback, Software, Validation

References

1. Avasthi P, York R. (2026). Biology needs to become prospective. https://doi.org/10.57844/arcadia-68c2-7g2y
2. Golden J, Kiefl E. (2026). Agentic PyMOL: Structural analysis in PyMOL through natural language. https://doi.org/10.57844/arcadia-gsbx-phd2