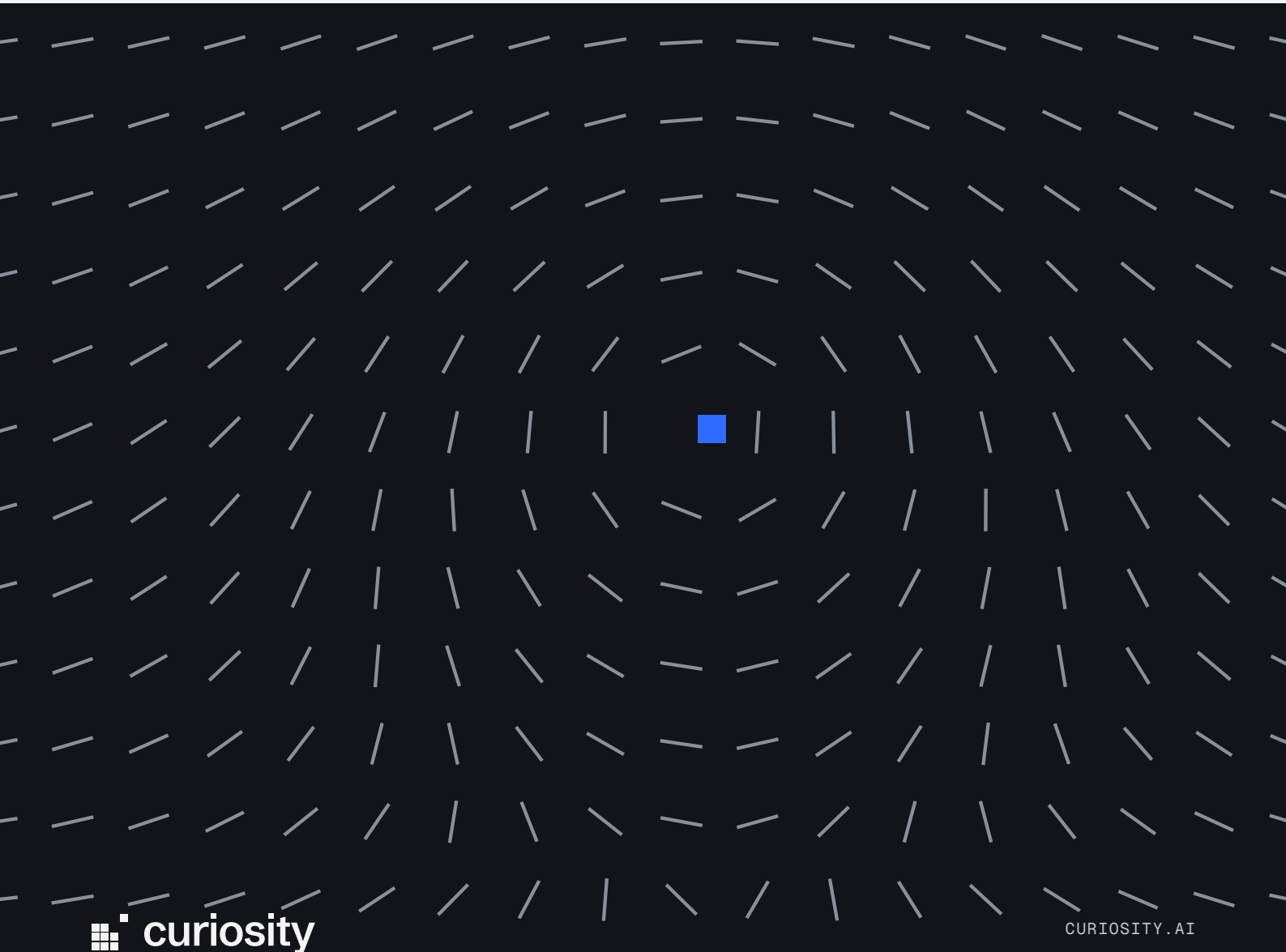


Build it from open source? Count the seams first.

Every part of an enterprise AI stack has an open source equivalent. This paper counts the work between them, shows what Curiosity Studio ships in its place, and why that makes it the place to describe an app and ship it.

TOPIC	FOR	DATE
Build or buy, and prompt to production	Architects and engineering leads	October 2026



The parts are free. The seams are not.

01	Every part exists. The seams do not What the assembled stack leaves your team to build.	03
02	Integration carries its permissions Connectors, extraction and OCR that write one graph.	06
03	One permission model, on every read Why access checked in one place beats five copies.	09
04	One process, one transaction log Graph, text and vectors that cannot drift apart.	12
05	Run it like one system Sizing, replicas, backups and configuration in git.	15
06	Describe it, then ship it Why generated apps need everything already in the box.	18
07	When to build it yourself The trade-offs, and a test on your own data.	22

IN ONE PARAGRAPH

A graph database, a search engine, a vector store, connectors, OCR and an LLM framework can each be had for free. What is not free is the code between them: permissions copied into every index, deletes that reach every copy, one audit trail across all of it. Studio ships those seams closed. That is also why an app described in a sentence can go to production on it.

01

Every part exists. The seams do not.

The stack a capable team would assemble, part by part

What each part leaves to the code between them

Why the seams cost more than the parts

1.1

The work is between the boxes

Each open source part is good at its job.
The cost is the code that makes six of
them agree on one record and one user.

A capable team can assemble an enterprise AI stack in a quarter. A graph database for the relationships. A search engine for text. A vector store for meaning. A connector framework, a document parser, an OCR engine and an LLM framework on top.

Each of those is a fair choice. None of them is the problem.

The problem is that each keeps its own copy of the record and its own idea of who may see it. A ticket now lives in four places. Its access rule has to be copied into four indexes and kept in step.

When the ticket is deleted, four copies have to go. When a user leaves a team, four filters have to change.

That glue is the part no open source project ships, because it depends on how the parts are joined. It is also the part an auditor asks about first.

A SEAM

Code your team writes so two systems agree: on a record, on a user, on what changed.

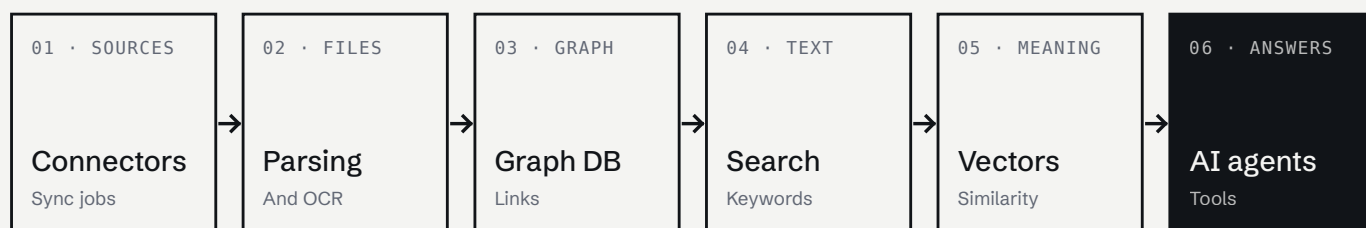


FIG. 1.1 · THE ASSEMBLED STACK. EVERY ARROW IS A SEAM YOUR TEAM WRITES AND RUNS.

TABLE 1.1

Part by part: what is left to build

FUNCTION	TYPICAL OPEN SOURCE PICK	LEFT TO YOUR TEAM	IN STUDIO
Connectors	A sync framework	Source permissions, deletes	Built in, with permissions
Files	A parser, an OCR engine	Routing, page numbers, scans	One extraction, text and markdown
Relationships	A graph database	Filling it, keeping it current	Typed graph, same process
Text search	A search engine	Your access filter per index	Indexed with access rules
Meaning	A vector database	A second copy, its staleness	Vector indexes beside the graph
AI	An LLM framework	Tools that respect permissions	Tools that search as the user
Interface	A web framework	Login, routing, the whole UI	Search, chat, your own front end

READ IT THIS WAY

The second column is free to download. The third column is the code your team writes, runs and defends in an audit, for as long as the stack exists. The right column is what Studio ships in its place.

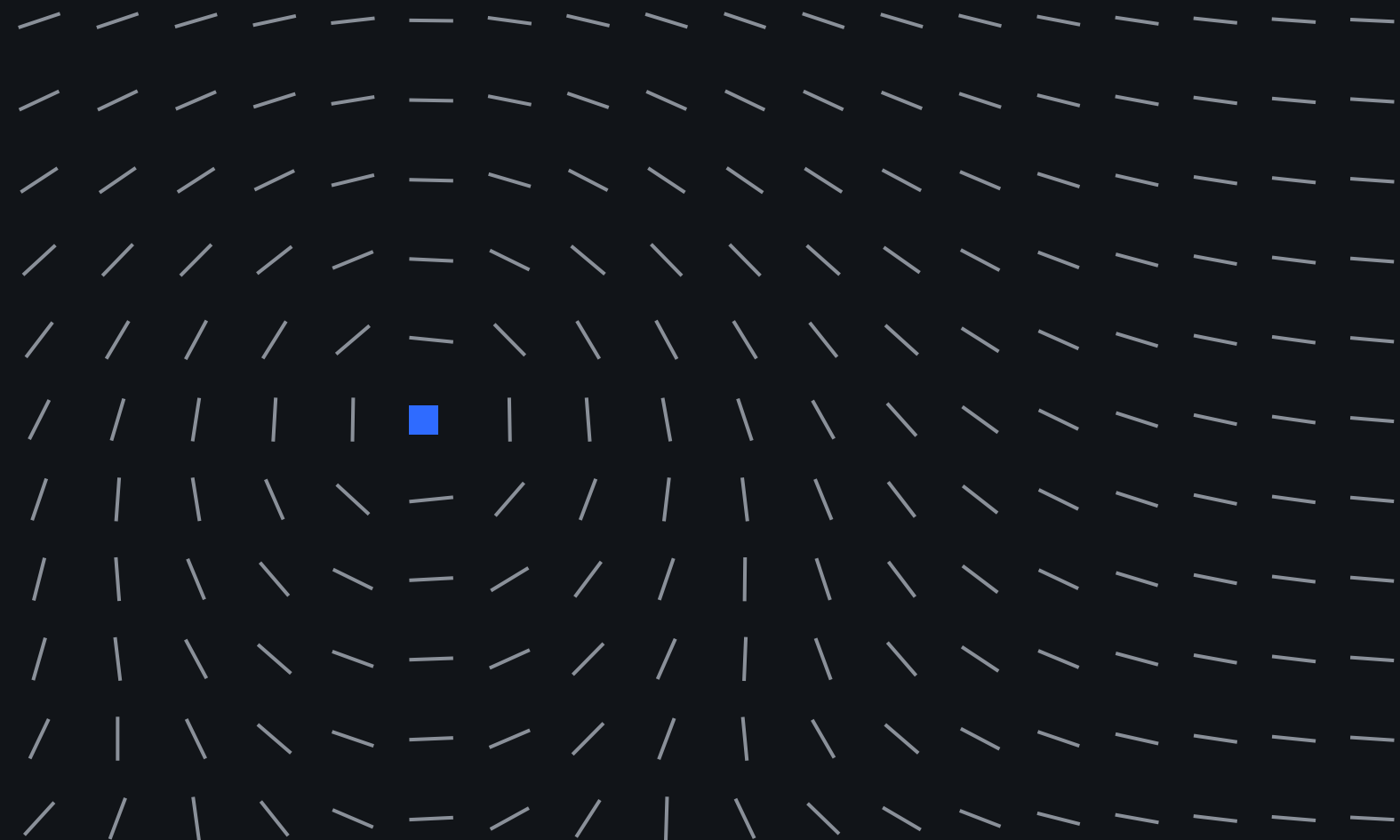
1.2 The seams grow faster than the parts

Six parts give fifteen pairs that may need to agree. Each upgrade of one part can break the glue to the other five.

The work is rarely the first version. It is keeping every copy correct while the sources, the users and the parts all change.

WHERE THE GAP SHOWS

"The chat tool can see what the search API cannot." When permissions live in several places, the places disagree, and a reader finds a record they should not.



02

Integration carries its permissions

Connectors that bring the access rules with the record

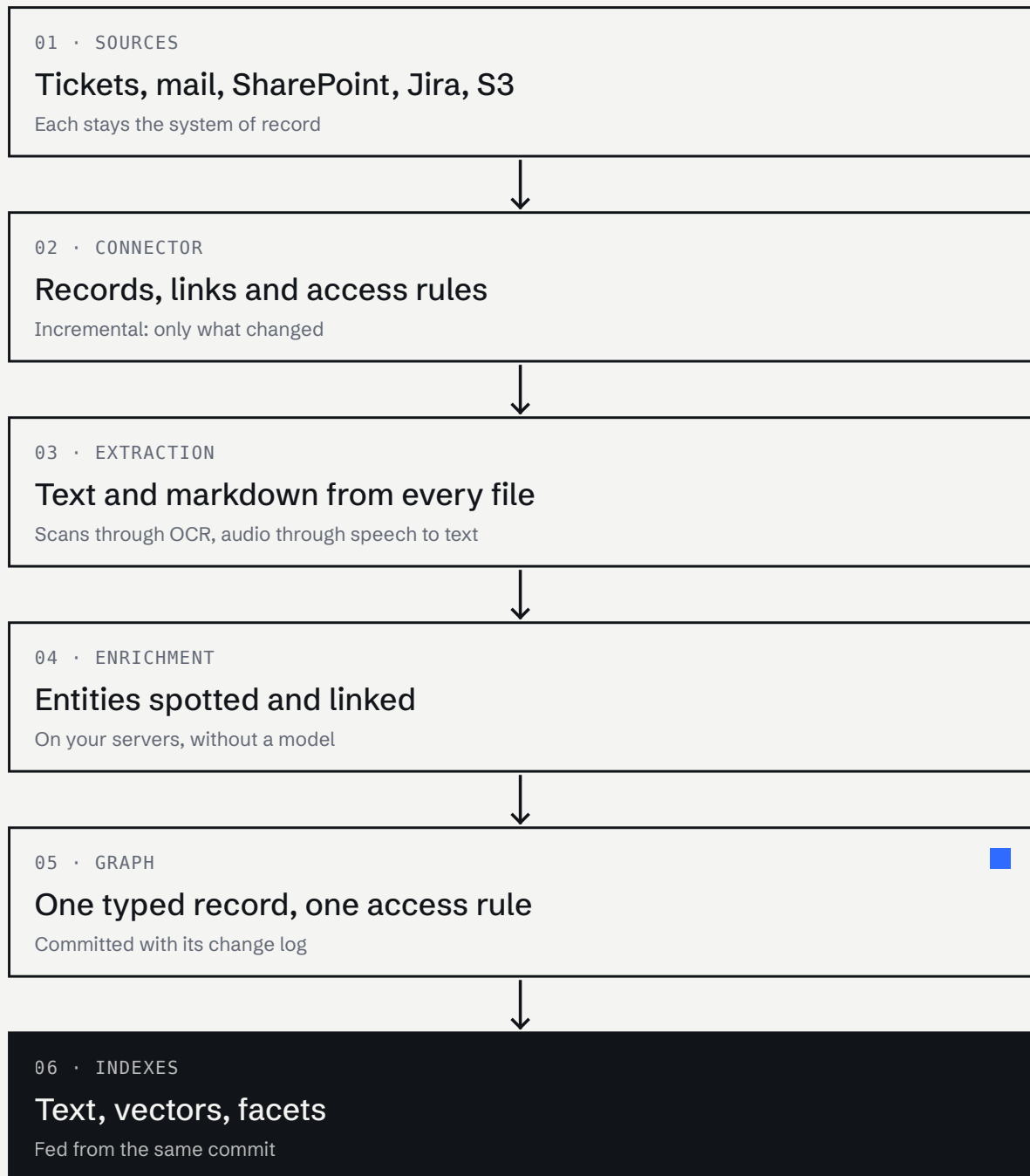
One extraction for every file, scans included

An SDK for the systems that have no connector

FIGURE 2.1

Source to graph

A record, its file and its access rule travel together into one graph. No step hands off to another product.



02

The connector writes who may see a record as edges in the graph, beside the record itself.

05

The marked step: one write holds the record, its links and its access rule.

06

Indexes follow from a queue kept with the data, so a crash resumes rather than rebuilds.

2.2

Connectors, files and the systems with none

Studio ships connectors for 70+ enterprise systems: SharePoint, OneDrive, Teams, Outlook and Exchange, Confluence, Jira, Salesforce, Zendesk, GitHub, Slack, Google Drive, Box, S3 and more. They sync incrementally, using each source's own change feed where it has one.

A connector brings the access rules with the record where the source has them. Where it has none, as in a storage bucket, you choose who may see it when you connect it.

Every file goes through one extraction, in a separate process so a bad file cannot take the server down. PDFs, Office files, email, archives, drawings and code come out as text and as markdown, page by page. Scans and photos go through OCR, which ships built in and runs on every platform with nothing to download. Heavier engines for handwriting and photos can be added for the pages that need them.

A system with no connector is a connector you write, in C# or Python, with the same SDK. It writes records, links and permissions into the same graph the built in connectors fill.

INCREMENTAL SYNC

Reading only what changed since the last run, instead of copying the whole source again.

IN AN ASSEMBLED STACK

A parser, an OCR engine, a queue and a sync job, each with its own failure mode.

KEY POINT

The access rule arrives with the record, in the same write.

03

One permission model, on every read

Where access rules end up in an assembled stack

How Studio checks one rule, for every feature

What stays out of reach even for an administrator

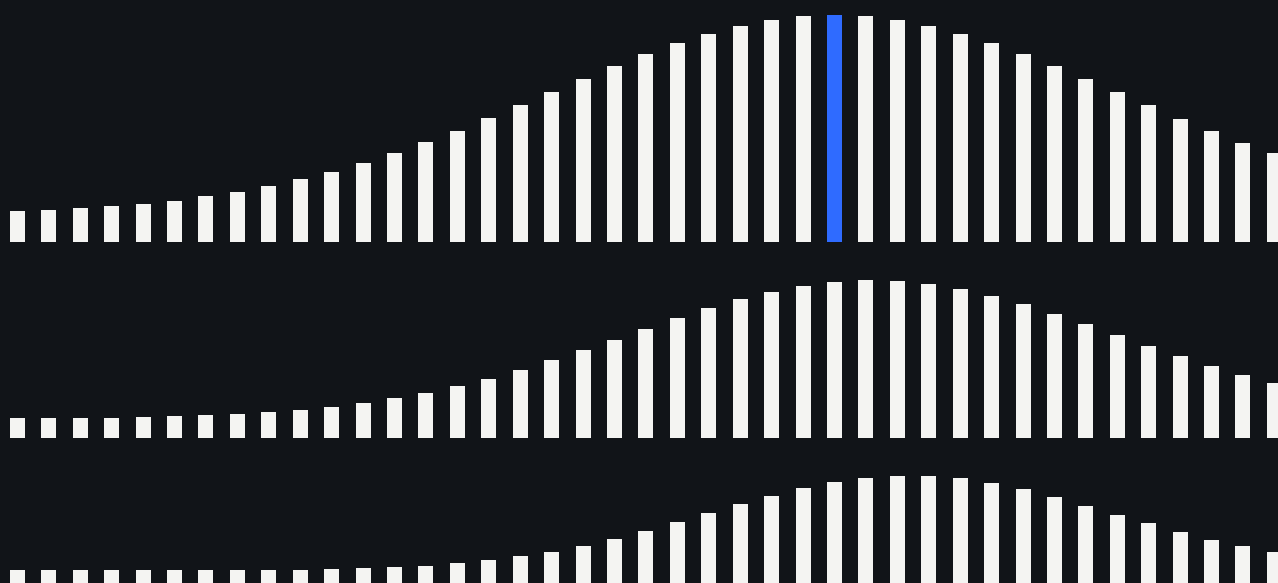


TABLE 3.1

Many copies of a rule, or one

WHERE A RULE IS NEEDED	ASSEMBLED STACK	STUDIO
Text search	A filter you add per index	☒ Applied before ranking
Vector search	A second filter, kept in step	☒ Same rule, same check
Graph traversal	Rules in the query, by hand	☒ Checked per record
AI tools and chat	Up to each tool's author	☒ Built in tools search as the user
Exports and APIs	Another code path to secure	☒ Hidden fields left blank
Revoking access	Reindex every copy	☒ The next query, no reindex

☒ YES, AS SHIPPED ☐ IN PART ☐ NO

READ IT THIS WAY

In the middle column every row is code that has to agree with every other row. In Studio the rule is an edge in the graph, and every feature reads the same edge. Custom tools you write get the same check, and choose explicitly when they need more.

3.2 Counts and facets that do not leak

A filter applied after the fact is not enough. A result count of 412 when the user may see 30 says that 382 records exist.

Studio checks access once, while a result set is built, before any count or facet is computed. The pages, the count and the facets all describe what that user may see.

PERMISSION-AWARE

Results filtered by what this user may see in the source system, checked per record, at query time.

3.3

Access, identity and the record of both

Access in Studio is relationships in the graph. A user belongs to groups, groups can nest, and a record is owned by the users or groups that may see it. Access means a path exists. A rule can restrict a whole record type or a single field of it, and a field the reader may not see is left blank in an export.

Some records never leave the server at all. Tokens, secrets, scheduled tasks and file blobs are marked so no generic reader returns them, even to an administrator. A secret is write only: code may use it, nobody reads it back, and every use is logged.

People sign in through your identity provider: Microsoft Entra ID, Google, Okta, Auth0, AWS Cognito, OneLogin, or any SAML 2.0 provider. Multi factor sign in is enforced there.

Data at rest is encrypted under a master key you supply. An audit log records the actions an administrator turns on, in files chained with SHA-256 so a gap or an edit shows.

RELATIONSHIP-BASED ACCESS

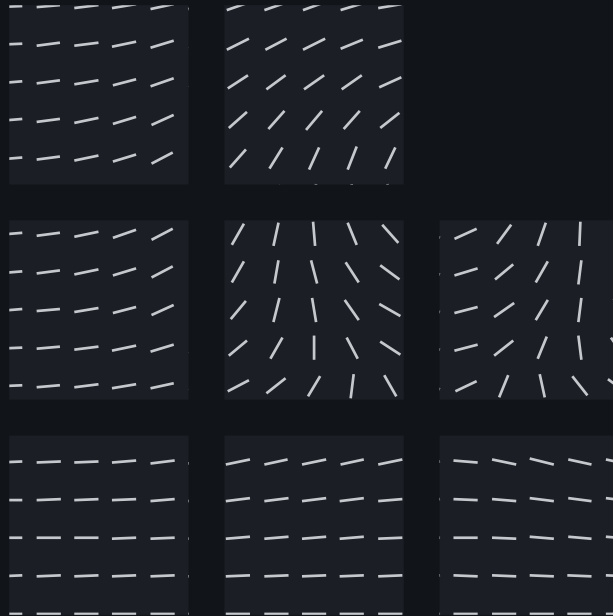
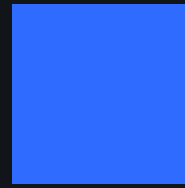
Access decided by a path in the graph, from a user through groups to a record.

READ ONLY ADMINS

Write protection makes every administrator read only until they unlock it for a set time.

KEY POINT

One rule, one place, read by every feature.



04

One process, one transaction log

Graph, text and vectors stored side by side

Why the copies cannot drift apart

What runs in memory, and what that buys

4.1

Graph, text and vectors in one store

Studio is one server process. Its graph engine is its own, on RocksDB. The text index pages and the vectors live in the same store, so one backup holds all of them.

```
// Tickets that mention a part, and that this user may see
var tickets = await Q(userID).StartSearch("Ticket", "Text", "hydraulic pump")
    .Out("Part", "Mentions")
    .Out("Ticket", "MentionedIn")
    .FilterRestrictedNodes()
    .Take(20)
    .ToListAsync();
```

LISTING 4.1 · A TRAVERSAL, A SEARCH AND THE ACCESS CHECK IN ONE QUERY. ILLUSTRATIVE, NOT THE EXACT API.

ASSEMBLED STACK	STUDIO
Three stores, three backups	One store, one backup
A sync job between each pair	One commit feeds every index
Deletes chased into each copy	Edges and indexes fixed on delete

4.2

A write is one write

The record and its change log are committed together. The text and vector indexes are fed from a queue kept in the same store, so after a crash they resume where they stopped.

Deleting a record fixes every edge that pointed at it and removes it from every index.

WHY IT MATTERS

In an assembled stack, "the search still shows a deleted ticket" is a bug class. Here there is one record, so there is nothing left behind to show.

4.3

Fast because nothing crosses a network

Traversal, ranking and the access check run in memory, in one process. No result set is shipped from one product to the next to be filtered.

Every node and its edges are held in memory, read without locks. A multi hop question walks edges in RAM rather than issuing a query per hop to another server.

A result set is cached once it is built, so the next page and the facets do not run the search again.

Search is hybrid by default.

Keyword hits come from the text index. Meaning comes from vector indexes built in, with embedding models that run on your servers, so no text has to leave them.

Vector matches above a similarity cutoff join the keyword results or rerank them.

Linking the graph to the text

happens locally too. Entity spotting and linking turn mentions in a field report into edges to the part or the asset they name, without a model call.

HYBRID SEARCH

Keyword and vector retrieval combined into one ranked result.

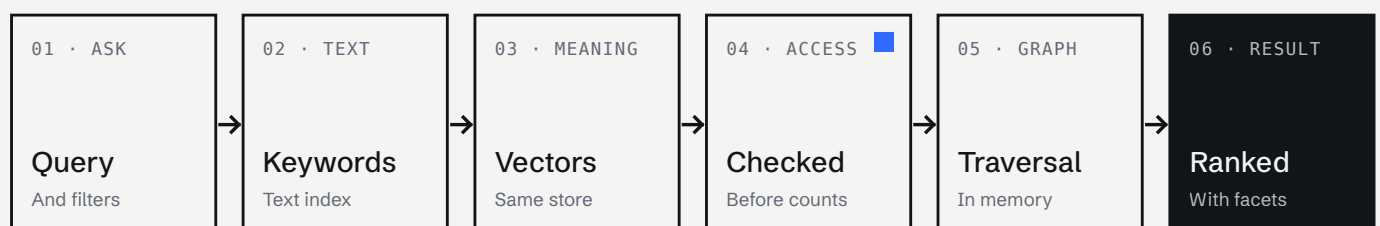


FIG. 4.1 · ONE QUERY, ONE PROCESS. THE MARKED STEP IS THE ONE AN ASSEMBLED STACK SPREADS ACROSS PRODUCTS.

05

Run it like one system

Sizing from a pilot to tens of thousands of users

Replicas, backups and upgrades as one product

Your configuration as files in your git

TABLE 5.1

Sizing from the deployment guide

TIER	USERS	NODES	QUERIES A DAY	HARDWARE
Pilot	up to 50	up to 1M	up to 50k	8 vCPU, 32 GB
Small	up to 500	up to 10M	up to 500k	16 vCPU, 64 GB
Medium	up to 5,000	up to 100M	up to 5M	32 vCPU, 128 GB
Large	up to 25,000	up to 1B	up to 50M	64+ vCPU, 256+ GB

READ IT THIS WAY

These are the starting points the deployment documentation gives, not limits. One server holds the whole stack at every tier. Above 25,000 users, sizing is set per engagement.

5.2

Scale reads out, keep one writer

Studio scales up first: the graph is held in memory, so RAM is the main dial. For reads it scales out. Read only replicas each keep a full copy, catch up from the primary's log, and keep search available while the primary restarts.

There is one writer per deployment, by design. That is what keeps one transaction log.

BACKUPS

A built in task backs up the graph, its indexes and its definitions together, to a folder you choose. Restoring one store restores all of it.

5.3

Operations you do not assemble

Studio installs as one container image, or with the Windows installer. It runs on premises, in a private cloud, or air gapped with an offline license, local models and usage reporting switched off.

Everything you configure is a file. Schemas, access groups, endpoints, AI tools, agents, scheduled tasks and indexes are exported as definitions, committed to a git repository on every change, and pushed to your remote. Staging to production is a review and an import.

Your code is kept as source and compiled by Studio. An upgrade compiles it against the new version, so a break shows as a compile error, not a silent failure. Two people editing the same endpoint get a conflict and a diff, not a lost change.

The server reports its own metrics, health and integrity checks. Scheduled tasks, connector runs and agent runs are each recorded, so a failed sync shows where it failed.

IN AN ASSEMBLED STACK

Six upgrade cycles, six backup procedures, six sets of metrics, and the glue tested against each.

DEFINITIONS

Studio's configuration as files: what a schema, endpoint or agent is, never the data itself.

KEY POINT

One product to upgrade, back up and watch.

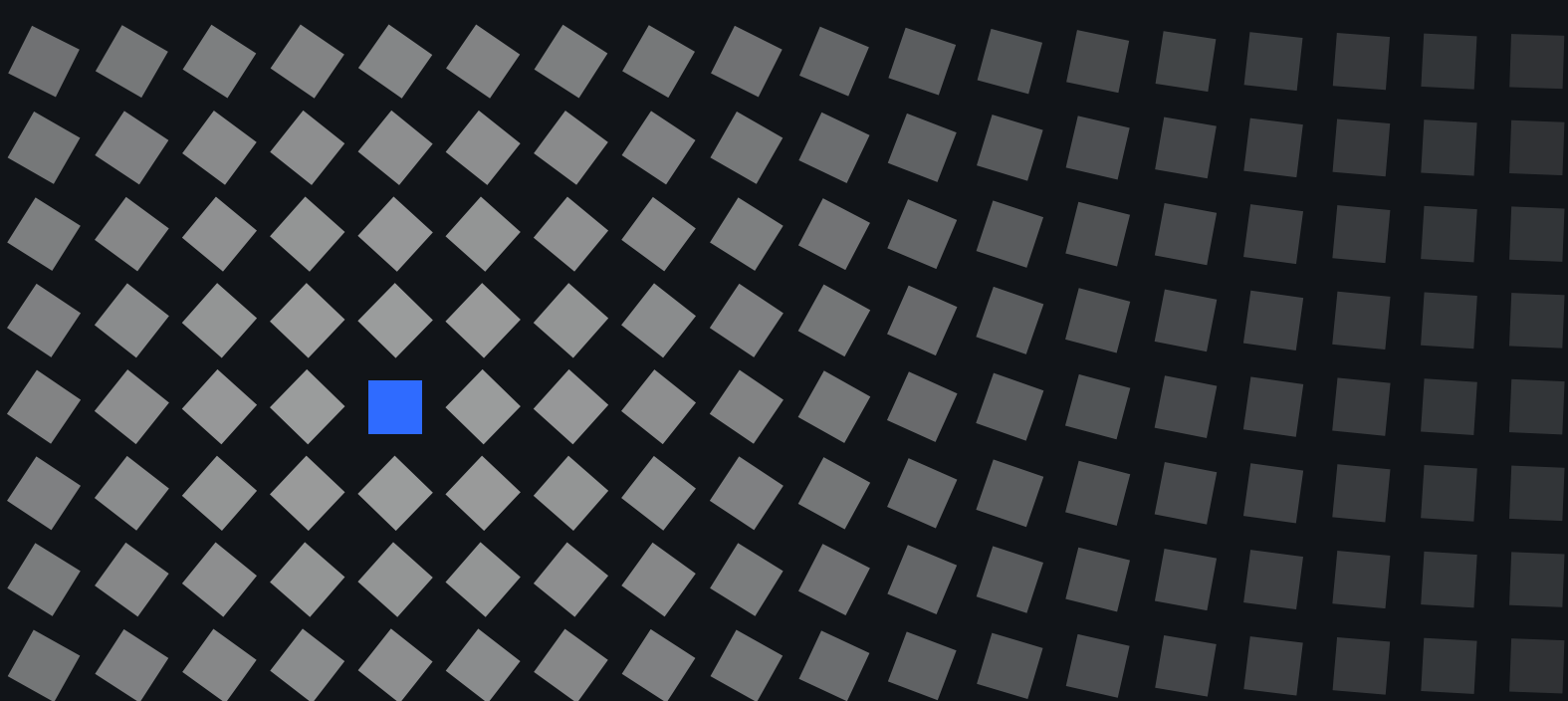
06

Describe it, then ship it

Why an app described in a sentence needs a platform under it

What Studio already holds for the app it builds

How every generated line stays yours to change



6.1

The interface is the easy part

Describing an app and getting one back, often called vibe coding, is easy to demo. Prompt-to-app is easy. Prompt-to-app that a regulator would accept is not. The hard part was never the interface. It is the data, the permissions and the proof.

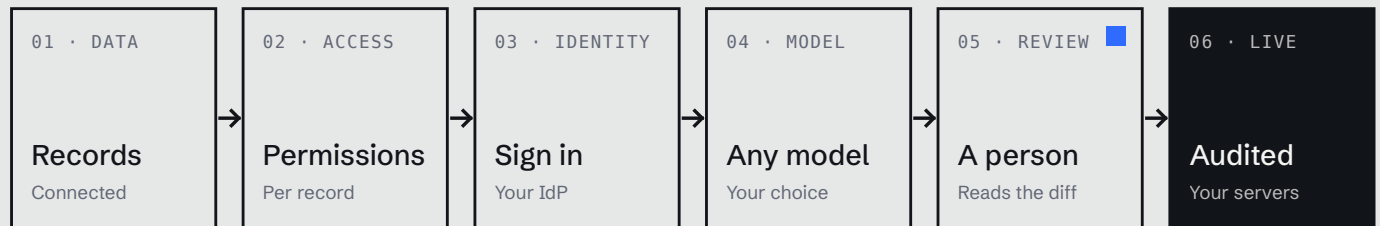


FIG. 6.1 · WHAT A GENERATED APP NEEDS BEFORE ITS FIRST SCREEN. IN STUDIO EVERY BOX EXISTS BEFORE THE PROMPT.

A generic code generator starts from an empty folder. It has to invent a database, a login, a permission model and somewhere to host the result. Each invention is a security review, and the app sees whatever data someone copied into it.

Studio starts from a deployment that already holds your connected records, their access rules, your sign in and your model. A tool built for the A320 line is a small piece of code on top. It inherits everything below it.

It cannot show a user anything the source system would not already show them, because it reads through the same check as search and chat.

KEY POINT

The prompt writes a small piece of code. Everything that makes it safe was already there.

VIBE CODING

Describing software in plain language and letting a model write it. Fast to start, risky without the platform under it.

6.2

Generated, then reviewed, then yours

Sudo writes ordinary Studio code in a sandbox. Nothing runs until a person approves the diff, and then it is a file in your git.

Sudo is the assistant in Studio that builds. Ask for a supplier risk review, and Sudo reads your schema, drafts the endpoint, the AI tool and the screen it needs, and compiles them against your deployment.

A failed build goes back to the draft, not to production.

The front end too. Sudo edits the sources of your own interface, builds a preview and hands you a link to try it. Approving the diff makes a build. Making it the live site is a separate step.

Sudo reads with your permissions, never elevated ones, and writes none of your data.

What comes out is not a black box. It is C# against a documented API, kept with the rest of your configuration. Your engineers can read it, change it by hand, or ask Sudo to change it again.

SUDO

The assistant in Studio that changes your configuration in a sandbox, for a person to approve.

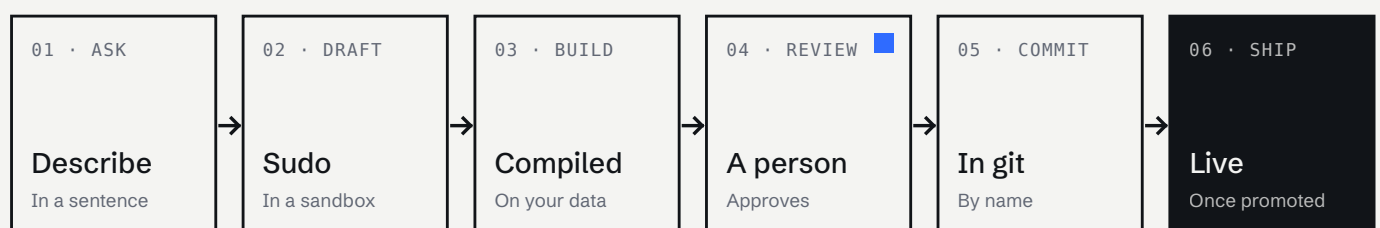


FIG. 6.2 · FROM A SENTENCE TO PRODUCTION. THE MARKED STEP IS THE ONE GENERIC GENERATORS SKIP.

TABLE 6.3

Out of the box, and open where it counts

WHAT YOU CHANGE	WRITTEN IN	REACHED THROUGH
The data model	Schema files	Studio, or a definitions import
Endpoints and actions	C#, compiled by Studio	REST, permission-aware
AI tools and agents	C# and prompts	Chat, MCP, the APIs
Connectors	C# or Python	The connector SDK
The interface	C#, with Tesseract	Your git, a preview per build
The model	A setting, or a C# provider	Any OpenAI compatible endpoint

READ IT THIS WAY

Every row ships working on day one: search, chat, the built in tools and interface. Every row can also be replaced with your own, by a person or by Sudo, and kept as source in your git.

6.4

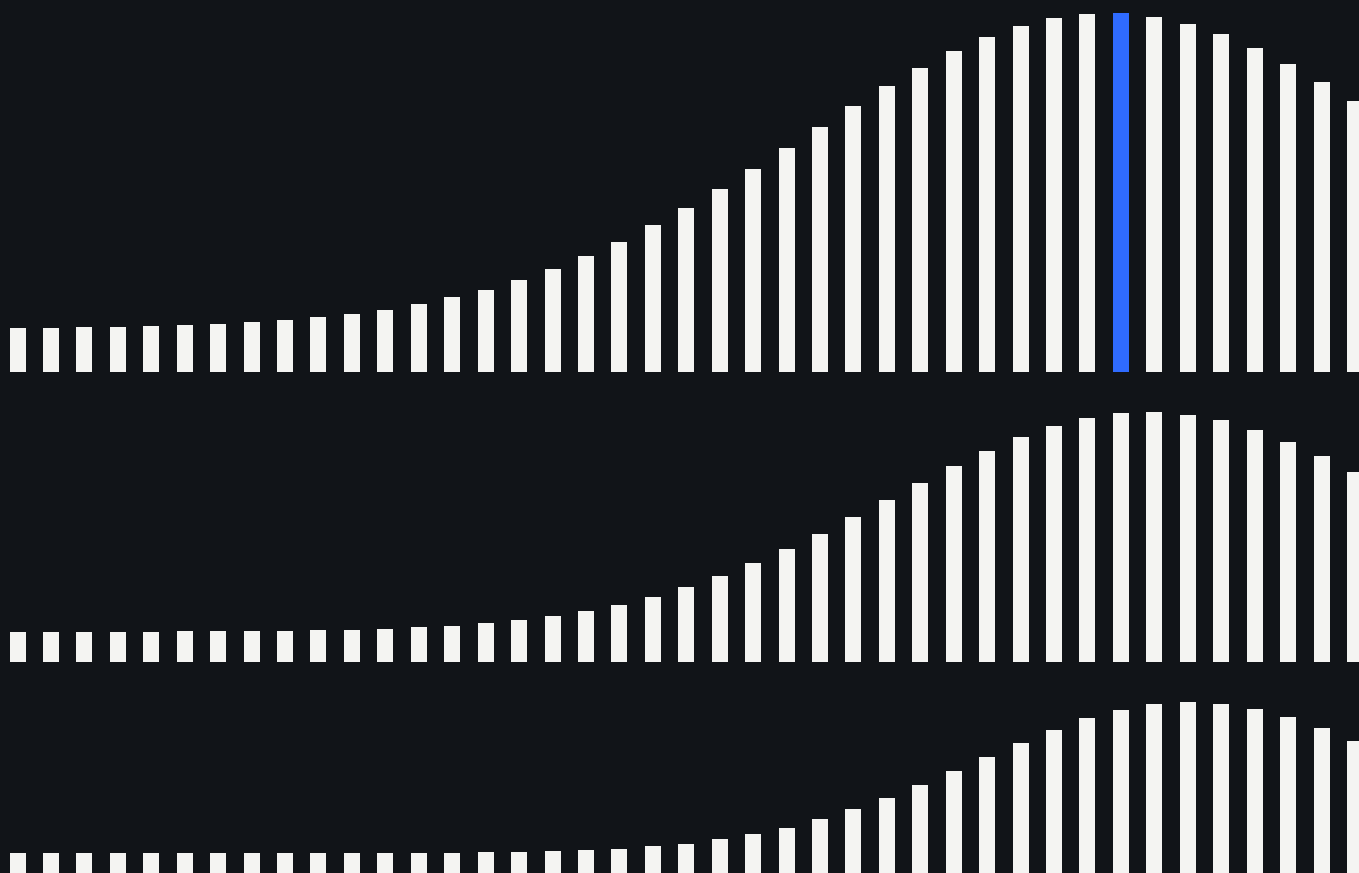
Why the two halves need each other

A platform with nothing to customize cannot become the tool a team needs. A generator with no platform under it builds apps that cannot pass a review.

Studio is both: the parts a regulated app needs, already joined, and the code surfaces where a sentence can become a feature.

MODELS

OpenAI, Anthropic, Google, Mistral, Azure OpenAI, AWS Bedrock, Ollama for open weights, and any OpenAI compatible server. The app does not change when the model does.



07

When to build it yourself

What Studio does not do, said plainly

The readers for whom an assembled stack is right

Three steps to test the difference on your data

TABLE 7.1

The trade-offs, said plainly

YOU WANT	STUDIO	ASSEMBLED STACK
Your own choice of each engine	☐ One engine per job	☒ Any part, swapped
Many writers across a cluster	☐ One writer, read replicas	☒ If you build it
Formal ontologies and reasoning	☒ A typed graph	☒ With a triple store
Code in any language	☒ C#, Python for connectors	☒ Any language
Permissions, deletes, audit joined	☒ Shipped	☒ Code you write
A first app in weeks	☒ On the connected graph	☐ After the platform

☒ YES ☒ IN PART ☐ NO

READ IT THIS WAY

If your team wants to own every engine, or already runs a search layer it trusts, building is a reasonable choice. Count the seams in the plan, not only the parts. If the point is the apps, not the platform, start from the platform.

7.2 Your code still fits

Choosing Studio is not choosing against open source. Your Python and C# reach the graph through the Library API, external agents through MCP and the OpenAI compatible API, and Curiosity's own UI and NLP libraries are open source.

LIBRARY API

The C# and Python package your own code uses to read and write the graph, under a scoped token.

Test the seams on your own data

01

Install the free developer version on one machine you control.

02

Connect one system with real permissions, then sign in as two users.

03

Describe one tool to Sudo, review its diff, and ship it.

BEFORE YOU START

- ☐ Which system holds the permissions your auditors ask about?
- ☐ Which tool has your backlog been asking for longest?
- ☐ Who would approve a change before it ships?
- ☐ What would your team have to build to match this?

AIRBUS

“By intelligently enhancing our search efficiency, Curiosity lets Airbus technical support quickly find information across millions of documents.”

SERVICES INNOVATION TEAM · AIRLINE SERVICES AT AIRBUS

THE PROBLEM

Forty years of technical data across CRM systems, network drives and engineering databases, in a vocabulary of acronyms and reference numbers that defeated ordinary search.

WHAT CHANGED

One entry point over those sources, with the engineering references extracted and joined in a graph, and filters by aircraft type, program and reference number.

Running on infrastructure our customers own.

Across Curiosity's production deployments, on premises and in private clouds.

30TB+

DATA CONNECTED IN PRODUCTION

20,000+

ACTIVE USERS ACROSS DEPLOYMENTS

15%+

EFFICIENCY GAIN IN PRODUCTION WORKFLOWS

70+

ENTERPRISE SYSTEMS CONNECTED

What to take from this paper

01

The parts are free. The seams are not.

Permissions, deletes and audit live between the parts.

02

One rule, one store, one process.

Studio ships the seams closed, so nothing drifts.

03

Generated apps need a platform under them.

Describe it, review the diff, ship it on your data.

Terms used in this paper

CONNECTOR

The part that copies mapped records and their access rules from a source system and keeps them in step.

HYBRID SEARCH

Keyword and vector retrieval combined into one ranked result.

PERMISSION-AWARE

Results filtered by what the user may see in the source system, checked per record at query time.

SEAM

Code a team writes so two systems agree on a record, a user or a change.

DEFINITIONS

Studio's configuration as files: schemas, endpoints, tools, agents and tasks, kept in git.

INCREMENTAL SYNC

Reading only what changed in a source since the last run.

READ ONLY REPLICA

A second server with a full copy of the data that serves reads and search.

VIBE CODING

Describing software in plain language and letting a model write it.

Describe it Monday. Ship it this week.

Install the free developer version and connect one system with real permissions. Or book a demo, and we build a tool your team asked for on data shaped like yours.

curiosity.ai/request-demo

DATA RESIDENCY

EU, on premises or private cloud

COMPLIANCE

GDPR

MEMBER

KI Bundesverband