

The app is the easy part. The foundation is not.

An app can be generated in an afternoon. What it stands on cannot. This paper collects documented incidents where apps leaked data, overreached, stopped scaling or went unmaintained, and shows the layer each one was missing.

TOPIC

Building apps without a foundation

FOR

CIOs and security leads

DATE

October 2026



The incidents happen in the foundation.

01	The app is the easy part What every app from scratch has to rebuild.	03
02	Where data leaks Open buckets, default passwords, keys in the browser.	06
03	Where assistants overreach Prompt injection, oversharing, agents in production.	09
04	Where it stops scaling Dual writes, stale permissions, throttled connectors.	12
05	Where maintenance never ends Framework churn, retired models, aging dependencies.	15
06	Where the supply chain gets in Hijacked packages, poisoned builds, invented names.	18
07	Build the foundation once What it must give every app, and how to test it.	21

IN ONE PARAGRAPH

None of the incidents in this paper were failures of a model or an interface. They were missing foundations: an access check, a credential, a storage setting, a sync, a patch. Every app built from scratch has to get each of those right on its own. Build the foundation once, check it once, and put every app on it.

01

The app is the easy part

Why generating an app got cheap, and its foundation did not

The five layers every app from scratch rebuilds

Why the risk grows with every new app

1.1

The cost moved below the interface

A screen can be generated in minutes. The layers that decide who sees what, where data lives and who patches it still have to be built, every time.

Generating an app from a prompt is now ordinary. So is a team assembling one from open source parts in a few weeks. The interface was never the hard part, and it is now the cheap part.

What the app stands on did not get cheaper. Servers and keys. Data connected from the systems that own it. Access checked per record. Code that someone patches.

An app built from scratch rebuilds each of those layers on its own. Ten apps rebuild them ten times, ten different ways, and each one has to be reviewed.

Veracode tested code from more than 100 language models in 2025. 45% of the samples failed security tests and introduced OWASP Top 10 flaws.

A Stanford study found that people using an AI assistant wrote less secure code than those without one, and were more likely to believe it was secure.

FOUNDATION

The layers under an app that it relies on but does not implement: hosting, data, access, models, change control.

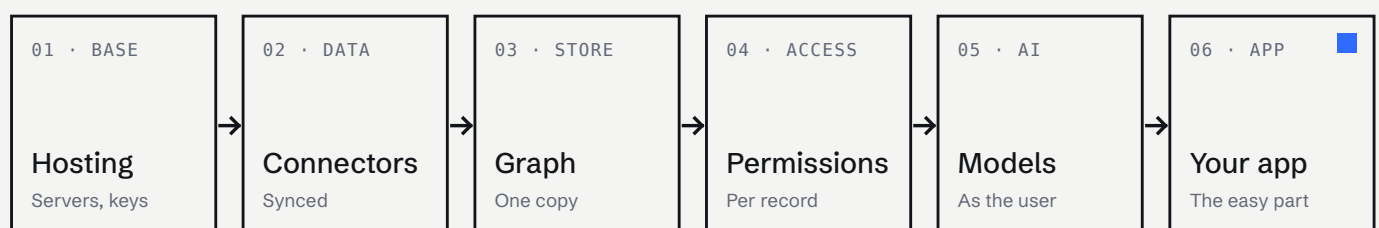


FIG. 1.1 · WHAT AN APP STANDS ON, FROM THE BASE UP. THIS PAPER FOLLOWS THE INCIDENTS INTO EACH LAYER.

TABLE 1.1

What an app from scratch rebuilds

LAYER	IT HAS TO GET RIGHT	CHAPTER
Credentials and storage	No default logins, no open buckets	02
Access per record	Every read checked, every copy	02, 03
Assistants and agents	Only what the user may see	03
Copies of the data	Index, vectors and source agree	04
Connectors	Throttling, deltas, permission sync	04
Upgrades	Frameworks and models that retire	05
Dependencies	Every package it pulls in	06

READ IT THIS WAY

Each row is a place where the incidents in this paper happened. None of them is the interface. An app on a shared foundation gets each row from that foundation. An app from scratch gets each row from whoever built it, that week.

1.2

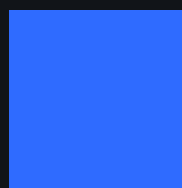
The risk grows with the number of apps

One app built from scratch is one review. Ten are ten reviews of ten different access checks, ten sets of credentials and ten dependency trees. Each one drifts on its own after launch.

That is the point where a security team stops being able to say yes quickly.

HOW TO USE THIS PAPER

Chapters 02 to 06 each start from documented incidents, then name the layer that was missing. Chapter 07 maps each layer to what a foundation has to give every app.



02

Where data leaks

Default logins, open storage and keys in the browser

The access check that was written backwards

Why a vector store is a copy of your data

TABLE 2.1

Six leaks, six missing layers

INCIDENT	WHAT WAS EXPOSED	MISSING LAYER
McHire (Paradox.ai), Jun 2025	Up to 64M applications	Admin login 123456
Tea app, Jul 2025	About 72,000 images, 13,000 IDs	Firebase bucket left open
Exam app on Lovable, Feb 2026	18,697 user records	Access check inverted
Microsoft AI research, 2020 to 2023	38TB incl. Teams messages	Token for a whole account
DeepSeek, Jan 2025	Over a million log lines	Database with no login
Escape scan, Oct 2025	400+ secrets in 5,600 apps	Service keys in the browser

READ IT THIS WAY

No row is a model failure. Each is a credential, a storage setting or an access check that the app or its builder had to get right alone. The sources page lists who reported each one.

2.1 The check that ran backwards

The exam app is the clearest case. A researcher, Taimur Khan, found that its generated access check blocked signed-in users and let anonymous visitors through. Teachers and students at universities were among the 18,697 records.

Lovable said it scans every app before publishing and that acting on the findings is up to the owner.

ROW-LEVEL SECURITY

A database rule that limits each row to the users allowed to see it. It protects nothing when the app holds a key that bypasses it.

2.2

The same pattern, three more ways

A developer who built a directory app in three days, with Lovable and Cursor on a Supabase database, wrote up two compromises in its first days online. A database view ran with its owner's privileges, so row-level security never applied to it.

A scan of 20,052 launched apps by SupaExplorer found database credentials in the browser code of 11% of them. A public key there is fine by design. The concern is the service key that bypasses every rule, which the scan found in some of them.

Data also leaks by ordinary use. In 2023 Samsung engineers pasted confidential source code and a recorded meeting into ChatGPT, three times in a few weeks. Samsung then restricted generative AI on company devices. The engineers had no sanctioned internal tool.

And a vector store is a copy of the text it indexes. Researchers at Cornell recovered 92% of 32-token inputs exactly from their embeddings, and full names from clinical notes. Embeddings need the same access rules and deletion as the documents.

SERVICE KEY

A server credential that bypasses every access rule. It belongs on a server, never in code sent to a browser.

IN STUDIO

The browser holds a session token for the person signed in, never a key to the store. Every read is checked per record, vectors included.

KEY POINT

A leak is a missing layer, not a bad model.

03

Where assistants overreach

Instructions hidden in a message, an email or a form

Assistants that see everything a user was ever shared

Agents that hold production credentials

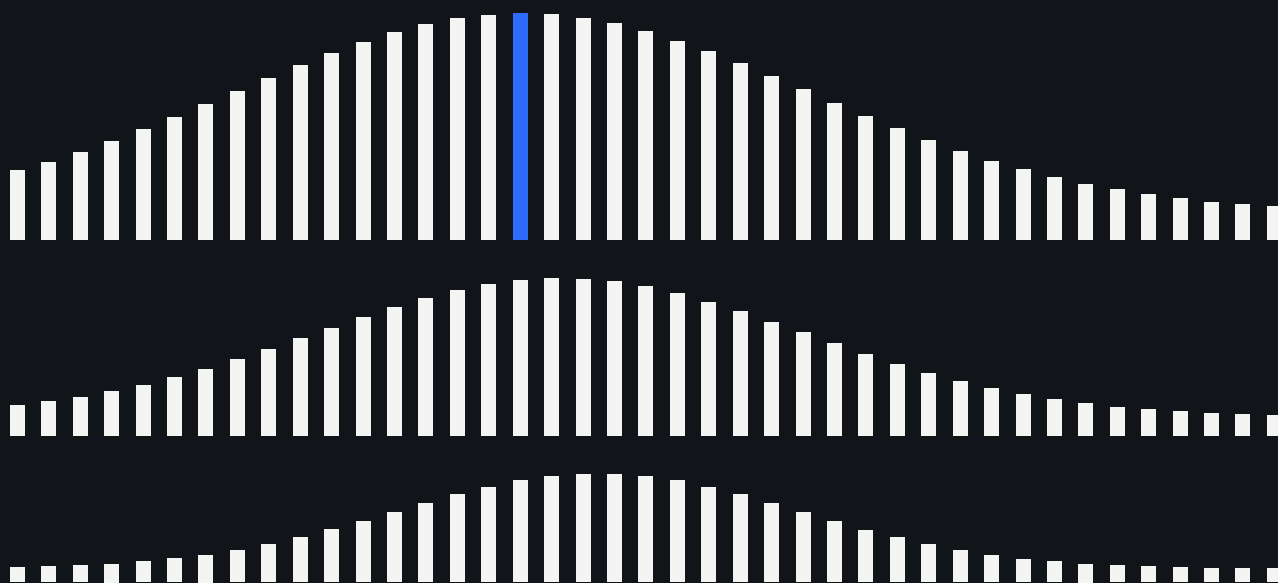


TABLE 3.1

Assistants that did what a stranger asked

INCIDENT	WHAT HAPPENED	MISSING LAYER
Slack AI, Aug 2024	Private channel data in a link	Untrusted text, a way out
EchoLeak, M365 Copilot, Jun 2025	One email, no click needed	Scope of what it reads
GitHub MCP, May 2025	Private repos posted publicly	Agent with all repos
ForcedLeak, Agentforce, Sep 2025	CRM data to an expired domain	Allow list left stale
Asana MCP, Jun 2025	Data visible to other tenants	Tenant check in a new path
Replit agent, Jul 2025	Production database deleted	Write access, no approval

READ IT THIS WAY

The first four are prompt injection: instructions planted in content the assistant later read. The last two are foundations an AI feature bypassed. EchoLeak (CVE-2025-32711) was rated 9.3 and fixed before any known use.

3.1

Oversharing, measured

An assistant grounded on a user's permissions finds everything that user was ever shared, including what was shared too widely years ago. Concentric AI reported that Copilot reached almost three million sensitive records per customer in the first half of 2025.

Microsoft's own deployment guidance for Copilot starts with remediating oversharing.

CAVEAT

The Concentric figures are a vendor's customer telemetry, not an independent study. The pattern does not depend on the exact number.

3.2

Limit what an assistant can reach

Simon Willison named the combination in June 2025: the lethal trifecta. An assistant with access to private data, exposure to untrusted content and a way to send data out can be tricked into leaking. Every row of Table 3.1 had all three.

No foundation makes prompt injection impossible today. What a foundation decides is how much an injected instruction can reach, and what it can change without a person.

In Curiosity Studio the built in tools search and read as the person asking, through the same per record check as search. An injected instruction reaches only what that person could already open.

Sudo, the assistant that builds and changes your setup, works in a sandbox with no network route out and no write access to your data. Nothing it drafts runs until an administrator approves the diff. An agent deleting a production database during a freeze is the case that approval exists for.

PROMPT INJECTION

Instructions hidden in content an assistant reads, which it then follows as if the user had asked.

LETHAL TRIFECTA

Private data, untrusted content and a way out, in one assistant. Remove one leg and the leak path closes.

KEY POINT

Limit what an assistant can reach. Approve what it changes.

04

Where it stops scaling

Copies of the same record that drift apart

Permissions that lag behind the content

Connectors that hit the source system's limits



TABLE 4.1

Seven ways an assembled stack drifts

PROBLEM	WHAT THE SOURCE SAYS	WHAT IT COSTS
Dual writes	Kleppmann, 2015	Index and store silently differ
New enemy	Google Zanzibar, 2019	Removed user sees new content
Permission sync	Elastic DLS docs	No ACL fields: open to all
Filtered vectors	pgvector README	10% filter: 4 of 40 rows
Index change	Elastic mapping docs	Full reindex to change a field
Delta tokens	Microsoft Graph docs	Expired token: start from scratch
Throttling	SharePoint Online docs	25 searches a second, per app

READ IT THIS WAY

None of these is a bug in the part. Each is documented behavior that the glue between parts has to handle. At pilot size none of them shows. At a hundred thousand documents and a thousand users, all of them do.

4.1

Drift you do not see

Martin Kleppmann put the dual write problem plainly: "you probably won't even notice that your database and your search indexes have gone out of sync." A deleted ticket keeps showing in search. A revoked user keeps finding new files.

NEW ENEMY PROBLEM

A permission check that uses an access list older than the content, so a person already removed can see what was added after.

4.2

One store, one write, one check

Copies that cannot drift are copies that do not exist. One store, fed by one commit, with access checked in one place.

In Studio the graph, the text index pages and the vectors live in one RocksDB store. A record and its change log are committed in the same write. The indexes are fed from a queue kept in that store, so after a crash they resume rather than rebuild.

Access is an edge in the graph, written by the connector with the record. A query checks it once, while the result set is built, before any count or facet. A revoked access applies on the next query, with no reindex.

Connectors follow each source's change feed and recover when a source resets its sync token. Read replicas take search load off the one writer.

DUAL WRITE

Writing the same change to two systems with no shared transaction, so one can succeed and the other fail.

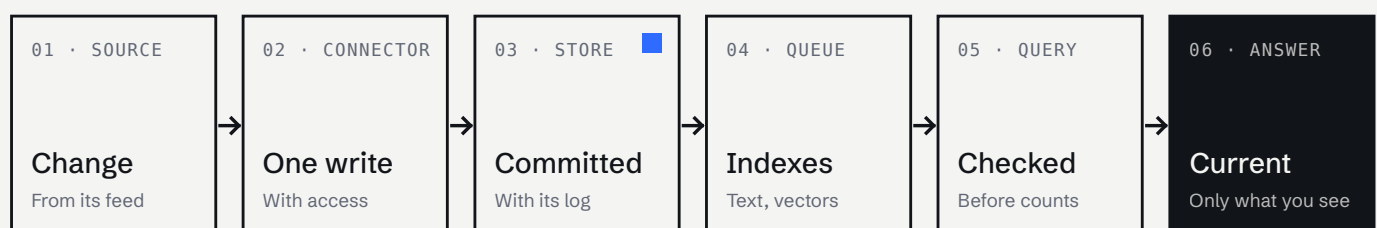


FIG. 4.1 · ONE WRITE IN STUDIO. THE MARKED STEP IS THE ONE AN ASSEMBLED STACK SPREADS ACROSS SYSTEMS.

THE APP IS THE EASY PART. THE FOUNDATION IS NOT.



05

Where maintenance never ends

Frameworks that rewrite their APIs every few months

Models and endpoints retired on the provider's schedule

Dependencies nobody has touched in years

TABLE 5.1

The schedule someone else sets

WHAT CHANGED	WHEN	WHAT IT MEANS FOR YOU
LangChain v0.2, v0.3, 1.0	May 2024 to Oct 2025	Three migrations in 18 months
OpenAI Assistants API	Notice Aug 2025, off Aug 2026	Rewrite to a new API
OpenAI gpt-4.5-preview	Gone three months after notice	Re-test on another model
Anthropic model retirements	At least 60 days notice	Old model calls fail
Old embedding models	OpenAI, retired Jan 2024	Re-embed the whole corpus
Inactive components	93% of audited codebases	Nobody upstream to patch

READ IT THIS WAY

Every row is a date someone else chose. An app from scratch absorbs each one alone. The last row is Black Duck's 2026 audit of 947 commercial codebases: 93% held components with no development for two years, 92% held components four or more years out of date.

5.1

Who maintains generated code

Code produced fast is still code to maintain. GitClear measured 211 million changed lines from 2020 to 2024: copied lines rose from 8.3% to 12.3%, and refactoring fell from 25% of changes to under 10%. It is a correlation, not a cause, and the direction is clear.

GLUE CODE

Sculley et al. at Google: a mature ML system can be at most 5% ML code and at least 95% glue, with "massive ongoing maintenance costs."

5.2

One upgrade instead of many

An app built from scratch carries its own framework versions, its own model calls and its own dependency tree. Each retirement notice lands on each app separately, and each app needs someone who still understands it.

On a foundation, most of that becomes one upgrade. Studio is one product, released and tested as one. Your extensions are kept as source and compiled by Studio, so an upgrade compiles them against the new version and a break shows as a compile error, not a failure in production.

The model is a setting. When a provider retires a model, you change the setting; the graph, the permissions and the tools stay. Changing the embedding model builds a new index beside the old one, and the graph is not touched.

What your teams build is kept as files in your git, with the history of who changed what. The next person to maintain it reads a diff, not an archaeology project.

RETIREMENT

The date after which a provider's model or API stops answering. Calls to it fail.

STILL YOURS

Your own endpoints and tools are your code. A foundation makes their upgrades visible, not optional.

KEY POINT

One product to upgrade, not one per app.

06

Where the supply chain gets in

Trusted maintainers, hijacked accounts, poisoned builds

Package names a model invents and an attacker registers

Why every dependency is a door

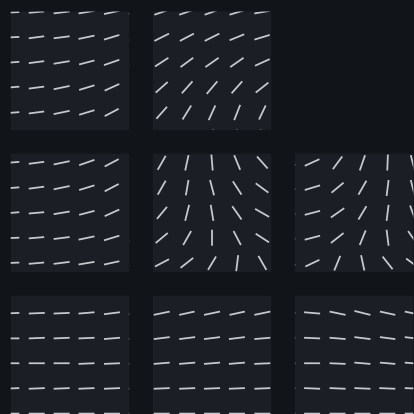


TABLE 6.1

Seven ways in, none through your code

INCIDENT	DATE	REACH
xz-utils backdoor	Mar 2024	Two years to earn trust
polyfill.io takeover	Jun 2024	100,000+ websites
Ultralytics on PyPI	Dec 2024	Poisoned build cache
tj-actions/changed-files	Mar 2025	Secrets in 23,000 repos' logs
chalk and debug on npm	Sep 2025	2 billion weekly downloads
Shai-Hulud worm	Sep 2025	500+ packages, self-spreading
ShadowRay, Ray	2023 to 2024	Hundreds of AI clusters

READ IT THIS WAY

Each came in through a dependency, a maintainer account or a build step, never through the app's own code. Sonatype counted more than 454,600 new malicious packages in 2025 alone.

6.1

Package names a model invents

Code generators recommend packages that do not exist. Spracklen et al. (USENIX Security 2025) generated 576,000 samples: 19.7% of recommended packages were invented, and 43% of the invented names came back on every one of ten tries. An attacker can register a name a model keeps inventing.

SLOPSQUATTING

Registering a package under a name that code generators invent, so generated code installs the attacker's package.

6.2

AI stacks add their own doors

The AI layer brings new dependencies with old habits. LangChain had two code execution flaws rated 9.8 in 2023 (CVE-2023-29374 and CVE-2023-36258). Ray's unauthenticated jobs API was exploited for seven months on hundreds of AI clusters. Wiz found more than 1,000 exposed Ollama servers. JFrog found about 100 Hugging Face models that run code when loaded.

A stack assembled from parts inherits every one of their dependency trees, and someone on your side has to watch all of them. The Cyber Safety Review Board expects vulnerable Log4j to stay in systems "perhaps a decade or longer."

A foundation does not remove dependencies. Studio has its own, and Curiosity tracks, patches and ships them as one release, with every third party component on its licenses page.

What changes is who watches, and how many trees there are. Sudo installs no packages. Adding one to your own code is an administrator's act, reviewed like any other change. The Windows installer refuses a download whose signature is not valid.

SUPPLY CHAIN

Everything your software is built from and with: packages, build tools, actions, registries, maintainers.

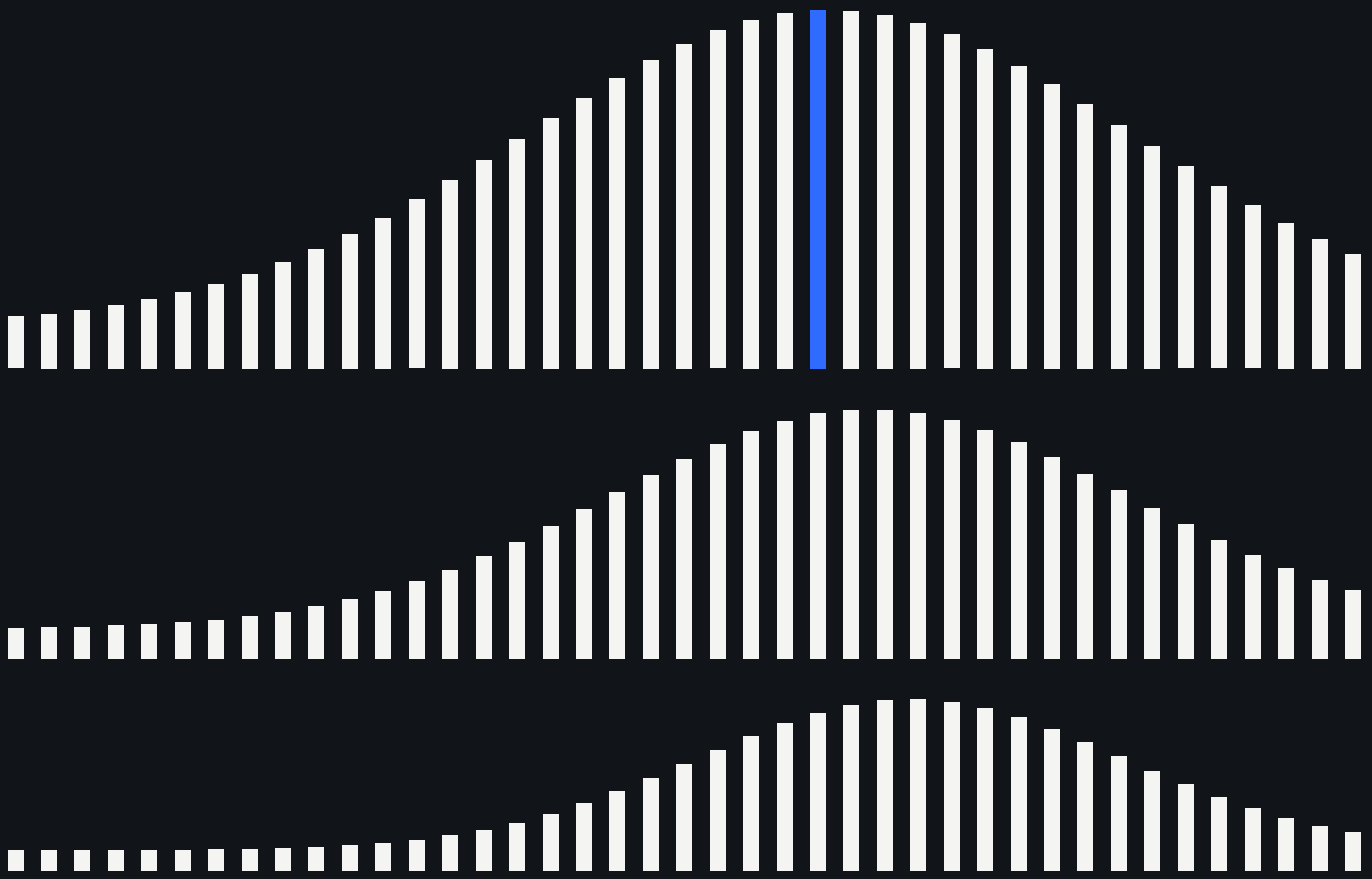
TRANSITIVE

A dependency of a dependency. Most of what a modern app runs, it never chose directly.

KEY POINT

Every dependency is a door. Count them before you open more.

THE APP IS THE EASY PART. THE FOUNDATION IS NOT.



07

Build the foundation once

What a foundation has to give every app

How Curiosity Studio provides each layer

Questions to ask before the next app ships

TABLE 7.1

Each failure, the layer that prevents it

FAILURE	THE FOUNDATION MUST	IN CURIOSITY STUDIO
Default logins, open storage	Own identity and storage	Your IdP, your servers, your key
Inverted or missing checks	Check every read itself	Per record, before counts
Assistants that overreach	Scope tools to the user	Tools search as the user
Agents changing production	Require a person's approval	Sudo: sandbox, diff, approval
Copies that drift	Keep one copy	One store, one commit
Retired models, frameworks	Absorb upgrades once	One release, model a setting
Hijacked dependencies	Track and patch centrally	Curiosity ships, Sudo adds none

READ IT THIS WAY

The middle column is a test for any foundation, ours included. The right column is how Studio answers it. An app on the foundation inherits every row instead of rebuilding it.

7.1

Generate the app, not the foundation

None of this argues against generating apps. It argues for generating them on top of something that already holds your data, your permissions and your change control. Then the generated part is small, and the review is of that part only.

DESCRIBE IT, SHIP IT

In Studio, Sudo drafts the endpoint, the tool or the screen. A person approves the diff. The app reads through the same checks as everything else.

Before the next app ships

01

List what each of your apps rebuilt: logins, checks, keys, connectors.

02

Pick one foundation and move one app onto it.

03

Make the next app a small piece of code on top.

ASK OF EVERY NEW APP

☐

Does the browser ever hold a key to the data?

☐

Who checks access on each read, and on each copy?

☐

What can its assistant reach if a document lies to it?

☐

Who patches its dependencies next year?

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 incidents happen in the foundation.

Keys, checks, storage, syncs and patches, not models.

02

Every app from scratch rebuilds it.

Ten apps, ten reviews, ten things that drift.

03

Build the foundation once.

Then generate the app on top, and review only that.

Who reported what

TOPIC	SOURCE	DATE
McHire	Ian Carroll and Sam Curry, ian.sh	Jul 2025
Tea app	The Record	Jul 2025
Exam app	The Register	Feb 2026
Microsoft, DeepSeek	Wiz Research	Sep 2023, Jan 2025
App scans	Escape; SupaExplorer	Oct 2025; Jan 2026
Samsung	Engadget; TechCrunch	Apr, May 2023
Embedding inversion	Morris et al., EMNLP	Oct 2023
AI code security	Veracode; Perry et al., CCS	Jul 2025; 2023
Slack AI	PromptArmor	Aug 2024
EchoLeak	Aim Security, via The Hacker News	Jun 2025
GitHub MCP	Invariant Labs	May 2025
ForcedLeak	Noma Labs	Sep 2025
Asana MCP	BleepingComputer; The Register	Jun 2025
Agent deletes database	Fortune	Jul 2025
Oversharing	Concentric AI; Microsoft Learn	2025; 2026
Lethal trifecta	Simon Willison	Jun 2025

Who reported what, continued

TOPIC	SOURCE	DATE
Dual writes	Martin Kleppmann, Confluent	May 2015
New enemy	Pang et al., USENIX ATC	2019
Search permissions, mappings	Elastic documentation	Current
Filtered vectors	pgvector README; Patel et al.	Current; 2024
Delta, throttling	Microsoft Learn	2025, 2026
LangChain releases	LangChain blog	2024, 2025
Model retirements	OpenAI; Anthropic docs	Current
Codebase audit	Black Duck OSSRA	Mar 2026
Code quality	GitClear	2025
Glue code	Sculley et al., NeurIPS	2015
xz-utils	Andres Freund; Russ Cox	Mar, Apr 2024
tj-actions, npm worm	CISA; StepSecurity; Wiz	Mar, Sep 2025
chalk and debug, polyfill.io	Aikido; Sansec	Sep 2025; Jun 2024
Ultralytics	PyPI blog	Dec 2024
Package hallucination	Spracklen et al., USENIX	2025
AI CVEs, Log4j	NVD; Oligo; Wiz; JFrog; CSRB	2022 to 2024
Malicious packages	Sonatype	2026

Terms used in this paper

DUAL WRITE

Writing one change to two systems with no shared transaction, so one can succeed and the other fail.

LETHAL TRIFECTA

Private data, untrusted content and a way out, in one assistant.

PROMPT INJECTION

Instructions hidden in content an assistant reads, which it then follows.

SERVICE KEY

A server credential that bypasses every access rule. Never in a browser.

FOUNDATION

The layers under an app it relies on but does not build: hosting, data, access, models, change control.

NEW ENEMY PROBLEM

A permission check using an access list older than the content it protects.

ROW-LEVEL SECURITY

A database rule limiting each row to the users allowed to see it.

SLOPSQUATTING

Registering a package under a name code generators invent.

Build the foundation once. Ship every app on it.

Install the free developer version and put one app on it. Or book a demo, and we show the checks, the approval and the audit trail on data shaped like yours.

curiosity.ai/request-demo

DATA RESIDENCY

EU, on premises or private cloud

COMPLIANCE

GDPR

MEMBER

KI Bundesverband