



The Augmented Auditor

Leveraging CodeQL and OpenCode for Precision Bug Hunting

08/10/2026

About us



Kévin Tellier

@kevintell



Hugo Vincent

@hugow

- **Pentester at Synacktiv**
- **Active Directory security**
- **White Box research**
- **LLM Automation**

You are handed the source code.
Where is the **bug** ?



READ IT

Thorough, but nobody reads a whole codebase



SEARCH IT

Fast, but a pattern does not understand the code

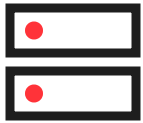


AUGMENT IT

Tools that understand code, and an AI that reasons about it

Three constraints

How do we automate vulnerability research in white-box?



LOCAL MODELS FIRST

Open-weight model behind our own gateway, client code never leaves



PRICE / VULNERABILITY RATIO

CodeQL narrows the code down, the LLM reads only the candidate paths



AUTONOMOUS BUT REUSABLE

The agent runs alone, the auditor gets back the database, the packs and the SARIF

The auditor's method

We want to turn it into a recipe the AI can follow

1

PREPARE

Install the target
Collect the code
Filter out known
libraries

2

MAP

Index the code
Map the attack
surface
Check nothing is
missing

3

ENTRY POINTS

Where does attacker
input come in?
Model what is not
covered

4

DANGEROUS CALLS

Which dependencies
are used?
Model the risky APIs

5

TRACE

Auth
Filters
Routing
How a request
travels

6

ANALYZE

Run the full analysis,
Manual / SAST
assisted
Verify each path
Learn from false
positives, loop

The best of both worlds

LLMS

STRENGTHS



Reasons about code

Qualifies a path, writes QL, writes a PoC



Autonomous work

Chains steps together without continuous supervision

LIMITATIONS



Limited context

~3.5 MB of raw code for 1M tokens



Methodology must be supplied

Without clear instructions, the agent goes in every direction

The best of both worlds

- Apache Brooklyn

```
34 public class BrooklynUserWithRandomPasswordSecurityProvider extends AbstractSecurityProvider implements SecurityProvider {
35
36     public static final Logger LOG = LoggerFactory.getLogger(BrooklynUserWithRandomPasswordSecurityProvider.class);
37     private static final String USER = "brooklyn";
38     private final String password;
39
40     public BrooklynUserWithRandomPasswordSecurityProvider() {
41         this.password = Identifiers.makeRandomId(10);
42         LOG.info("Allowing access to web console from localhost or with {}:{}", USER, password);
43     }
44
45     public BrooklynUserWithRandomPasswordSecurityProvider(ManagementContext mgmt) {
46         this();
47     }
48
49     @Override
50     public boolean authenticate(HttpServletRequest request, Supplier<HttpSession> sessionSupplierOnSuccess, String user, String pass) throws SecurityProviderDeniedAuthentication {
51         if ((USER.equals(user) && this.password.equals(password)) || isRemoteAddressLocalhost(request)) {
52             return allow(sessionSupplierOnSuccess.get(), user);
53         } else {
```

- A **semantic code analysis engine**
- The source is extracted into a **relational database**
- Questions are written in **QL**, a declarative query language

The best of both worlds

CODEQL

STRENGTHS

 **Search at scale**
Code becomes a queryable database

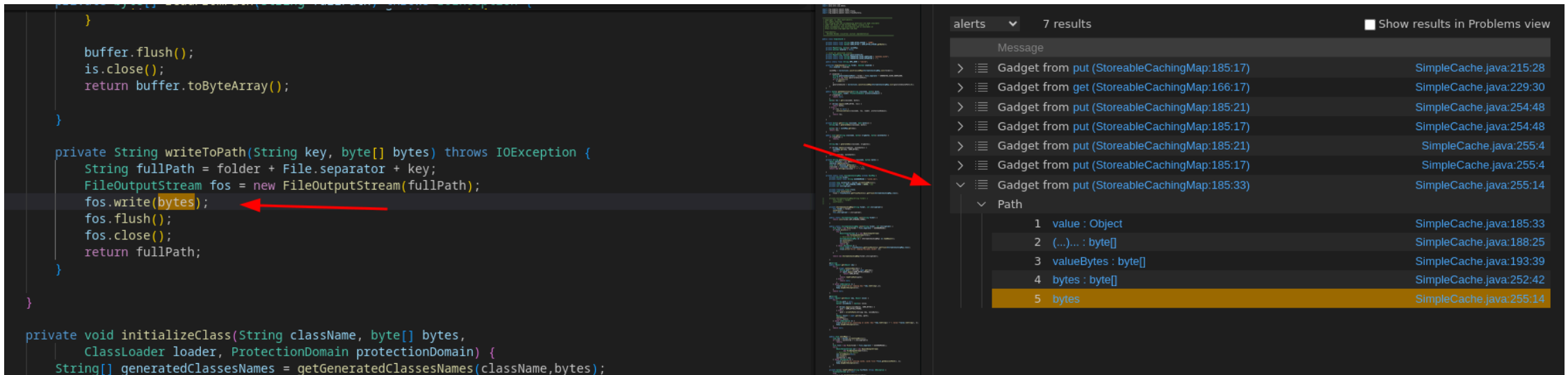
 **Deterministic approach**
Two runs, the same result twice

 **Community collaboration**
Shared query and model packs

LIMITATIONS

 **Steep learning curve**
QL is a full-fledged language

 **Tedious manual triage**
Sometimes hundreds of paths to qualify by hand



The screenshot displays the CodeQL interface with three main components:

- Code Editor:** Contains Java code. The line `fos.write(bytes);` is highlighted, and a red arrow points from it to the results panel.
- Query Editor:** Contains a SQL-like query: `FROM FileWrites fw WHERE fw.bytes IS NOT NULL`.
- Results Panel:** Shows 7 results under the 'alerts' tab. The 'Path' section is expanded, showing a list of values and their corresponding file locations in `SimpleCache.java`.

Path	
1 value : Object	SimpleCache.java:185:33
2 (...)... : byte[]	SimpleCache.java:188:25
3 valueBytes : byte[]	SimpleCache.java:193:39
4 bytes : byte[]	SimpleCache.java:252:42
5 bytes	SimpleCache.java:255:14

GREP "EXEC.COMMAND"

```
// exec.Command(name) runs a binary
log("exec.Command(" + name + ")")
exec.Command(name)
```

EVERY LINE THAT CONTAINS THE TEXT

CODEQL QUERY

```
// exec.Command(name) runs a binary
log("exec.Command(" + name + ")")
exec.Command(name)
```

ONLY THE ACTUAL CALL

Can you remember?

Can you name every **deserialization sink**
in Java ?

`ObjectInputStream.readObject`

`XMLDecoder.readObject`

...

Can you remember?

...in every library ?

`ObjectInputStream.readObject`

`XMLDecoder.readObject`

`XStream.fromXML`

`Yaml.load`

`...`

Can you remember?

...and every **other format** : binary, XML, YAML, JSON ?

`ObjectInputStream.readObject`

`XMLDecoder.readObject`

`XStream.fromXML`

`Yaml.load`

`Kryo.readObject`

`JSON.parseObject`

...

Can you remember?

...in **every language** ?

`ObjectInputStream.readObject`

`XMLDecoder.readObject`

`XStream.fromXML`

`Yaml.load`

`Kryo.readObject`

`JSON.parseObject`

`pickle.loads`

`BinaryFormatter.Deserialize`

`unserialize`

`Marshal.load`

...

```
extensions:
- addTo:
  pack: codeql/java-all
  extensible: sinkModel
data:
- ["com.caucho.hessian.io", "AbstractHessianInput", True, "readObject", "", "", "Argument[this]", "unsafe-deserialization", "manual"]
- ["com.caucho.hessian.io", "Hessian2StreamingInput", True, "readObject", "", "", "Argument[this]", "unsafe-deserialization", "manual"]
```

A ROW IN YAML

Package, class, method, tainted argument, vulnerability kind

PLUGS INTO EVERY QUERY

Existing taint tracking picks the new sink up, no QL to write

A FILE YOU CAN REVIEW

Versioned, diffed and shared like any other pack

Why not just ask the AI to remember ?

FROM THE MODEL'S MEMORY

- Recalled again in every session
- Might be incomplete
- Nothing to review or diff

FROM A MAD MODEL

- Written once, used every time
- Deterministic
- Easy to add new sinks/sources

SOURCE

Attacker-controlled entry point: an HTTP parameter, a form field, a header

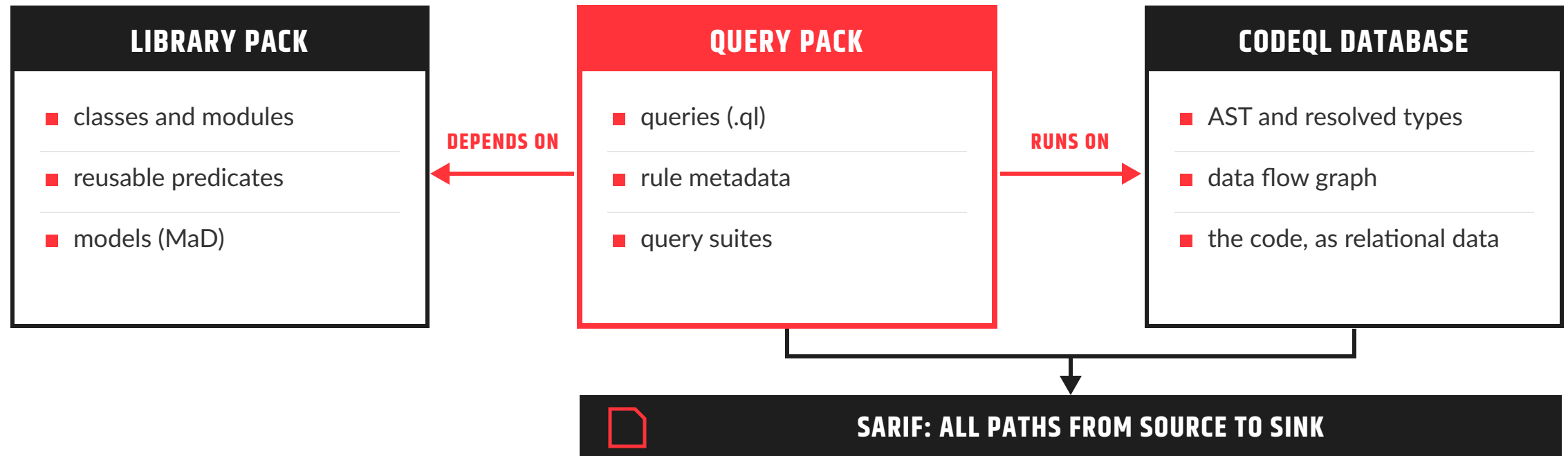
TAINT TRACKING

Following the **tainted** data from the source to the sink, across functions

SINK

Sensitive execution point: a SQL query, an `exec()`, a file read





codeql-vuln-research/

■ **SKILL.md**

The prompt: one specific skill

■ **scripts/**

Helper scripts (parsing, deterministic tasks)

■ **references/**

Documentation, templates, cheatsheets

AT REST

SYSTEM

FREE CONTEXT

The folder sits on disk, it costs no tokens

SKILL NEEDED

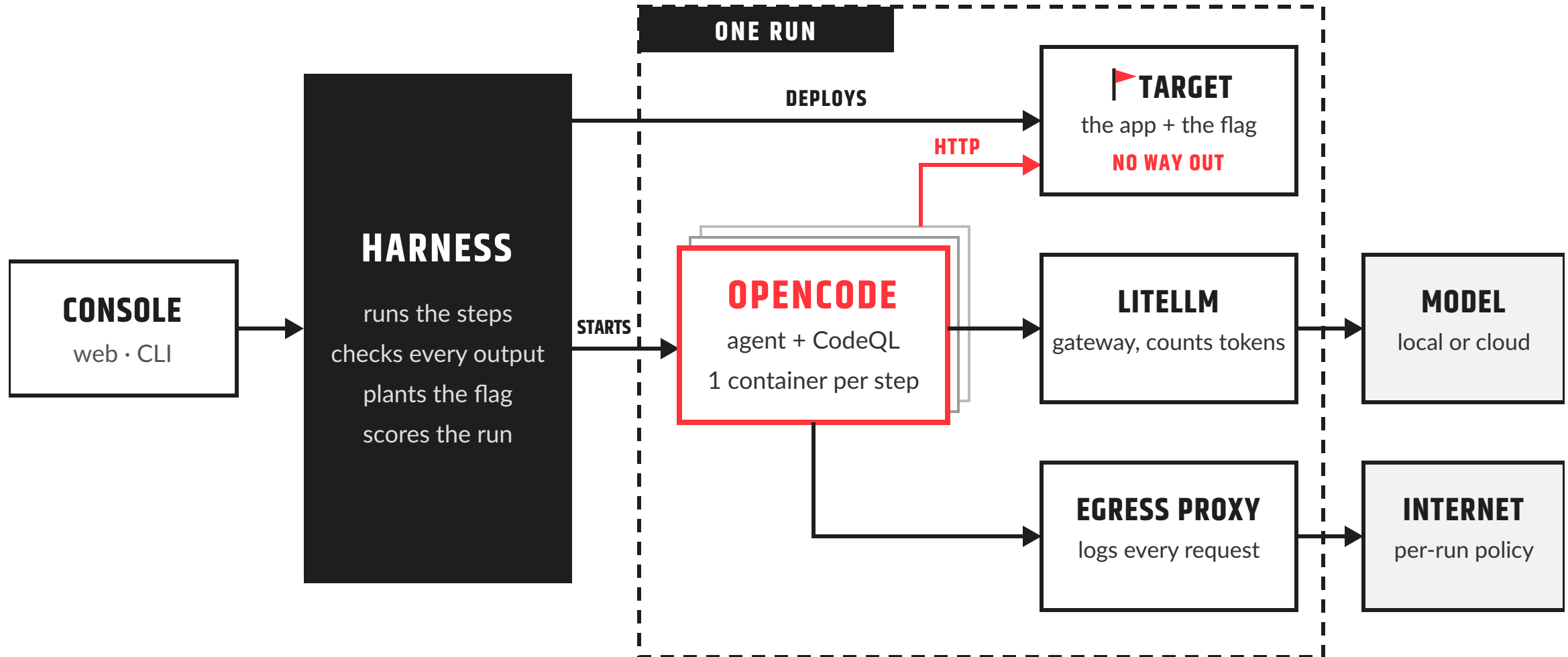
SYSTEM

SKILL

FREE CONTEXT

The harness injects it, along with its scripts and docs

Architecture



The agent that **executes** the recipe



AN OPEN-SOURCE AGENT

Reads code, runs commands, edits files, in a terminal



NOT TIED TO ONE MODEL

Local or cloud, whichever sits behind our gateway



SKILLS ARE PLAIN FOLDERS

The recipe stays in files we own, reusable elsewhere

Two kinds of target

BENCHMARK · 1-DAY

- A known vulnerability, at a pinned version
- No internet access, recent vulnerabilities
- A flag planted in the running app

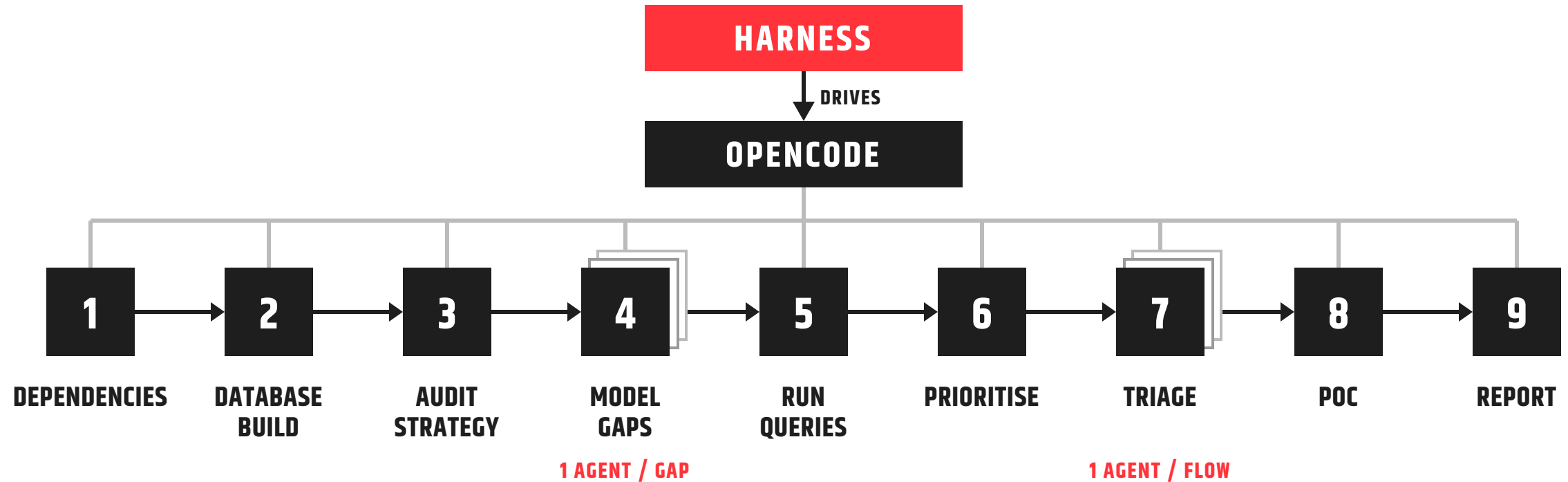
BENCHMARK & TRAINING

AUDIT · 0-DAY

- No known vulnerability
- Internet access (optional), CodeQL packs
- Up to date target (git repo or zip)

0-DAY RESEARCH

Benchmark run



WHAT ONE STEP RECEIVES	
ITS SKILL	the method for this step only
THE BRIEF	class + attacker privilege
WHAT EARLIER STEPS ESTABLISHED	summarised by the harness
LANGUAGE NOTES	only the frameworks the target uses
EXPECTED OUTPUT	files + their format

NEVER IN IT

- ✗ The earlier steps' conversation
- ✗ The flag value
- ✗ The CVE, the fix, the answer
- ✗ Git history
- ✗ The token budget

ONE SESSION, EVERY PATH

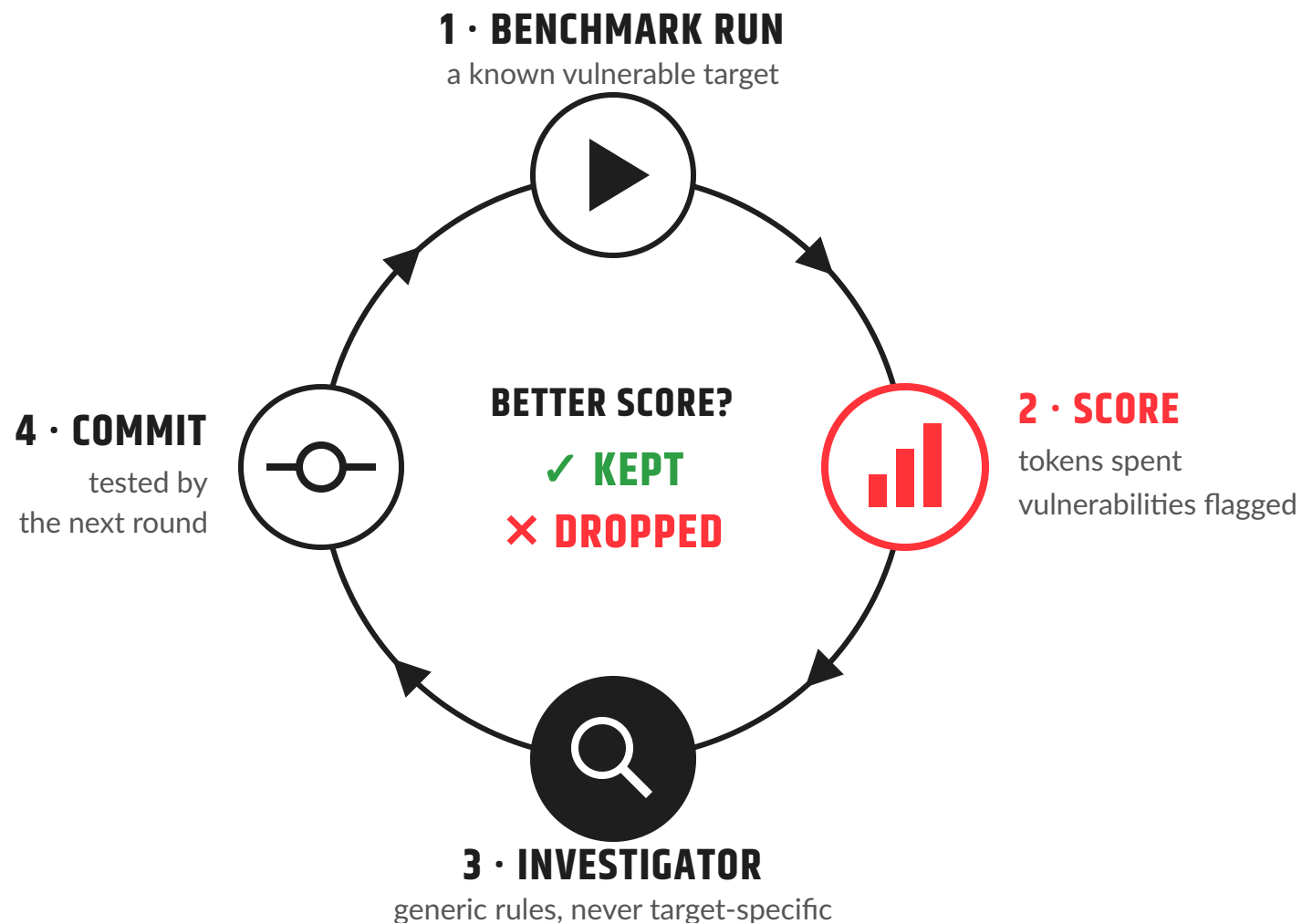
<i>BRIEF</i>	<i>PATH 1</i>	<i>FREE</i>		
<i>BRIEF</i>	<i>PATH 1</i>	<i>2</i>	<i>3</i>	<i>FREE</i>
<i>BRIEF</i>	<i>PATH 1</i>	<i>2</i>	<i>3</i>	<i>...</i>
			<i>N</i>	

ONE SUB-AGENT PER PATH

<i>BRIEF</i>	<i>PATH</i>	<i>CODE ±25</i>	<i>FREE</i>
<i>BRIEF</i>	<i>PATH</i>	<i>CODE ±25</i>	<i>FREE</i>
<i>BRIEF</i>	<i>PATH</i>	<i>CODE ±25</i>	<i>FREE</i>

Review

Autoresearch principle



THE SKILLS FIX

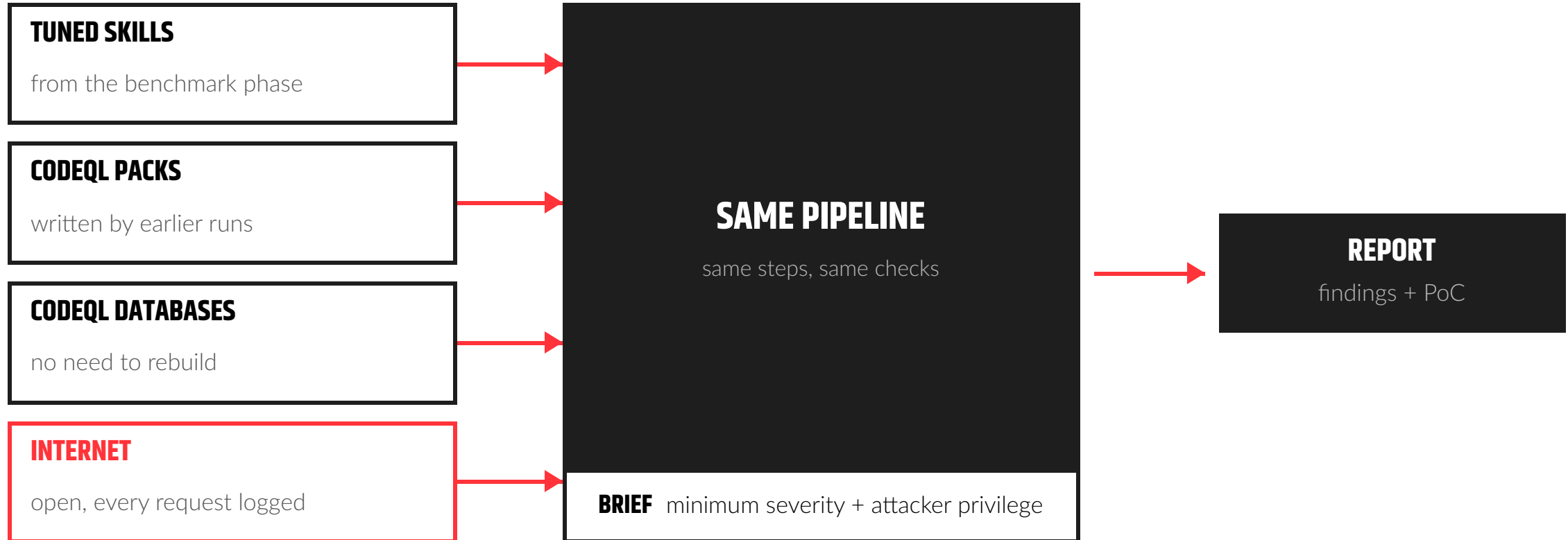
 **REASONING**

 **KNOWLEDGE**

 **METHODOLOGY**

→ **AUDIT MODE**
skills + packs reused

Audit run



What the auditor takes home

 Downloadable from the console

- | | |
|--------------------------|--|
| ■ CODEQL DATABASE | No rebuild on your side |
| ■ CODEQL PACKS | The models and queries the agent wrote |
| ■ SARIF RESULTS | Open them in your own CodeQL tooling |
| ■ TRIAGE VERDICTS | One per path, with the reasons |
| ■ POC AND REPORT | Replayable exploit, the deliverable |

Caveats

What does not work yet

OPENCODE IS A RIGID FIT

Resuming a failed run or riding out an inference outage is fought around the runtime. Its plus: the skills stay plain folders, reusable elsewhere

TARGETS THAT RESIST SETUP

Some code will not build or stand up. No container, no PoC: the step is dropped

A FIXED PIPELINE

The same steps in the same order. A target that needs another route does not get one

OUTSIDE CODEQL'S REACH

A flaw with no source-to-sink path never becomes a candidate

NEXT PHASE

AUTONOMOUS RESEARCH?

An agent beside the pipeline, for what the queries cannot see

LANGGRAPH INSTEAD OF OPENCODE?

A runtime built for resumable graphs

JEV-LIKE DECISION GATES?

A typed model answers yes/no, with a probability, between steps

Results

Local, cheap, and already finding 0-days

27B

LOCAL MODEL

Qwen 3.6 / 3.8 finds 1-days and 0-days

~2-15 \$

PER FULL RUN

30M to 100M tokens

RCE

PRE-AUTH, ARGOCD

Found end to end by the pipeline

0-DAYS

MULTIPLE PRODUCTS

Still not fixed

Demo

Orchestration in action



<https://www.linkedin.com/company/synacktiv>



<https://x.com/synacktiv>



<https://bsky.app/profile/synacktiv.com>



<https://synacktiv.com>

An incident? Contact csirt@synacktiv.com