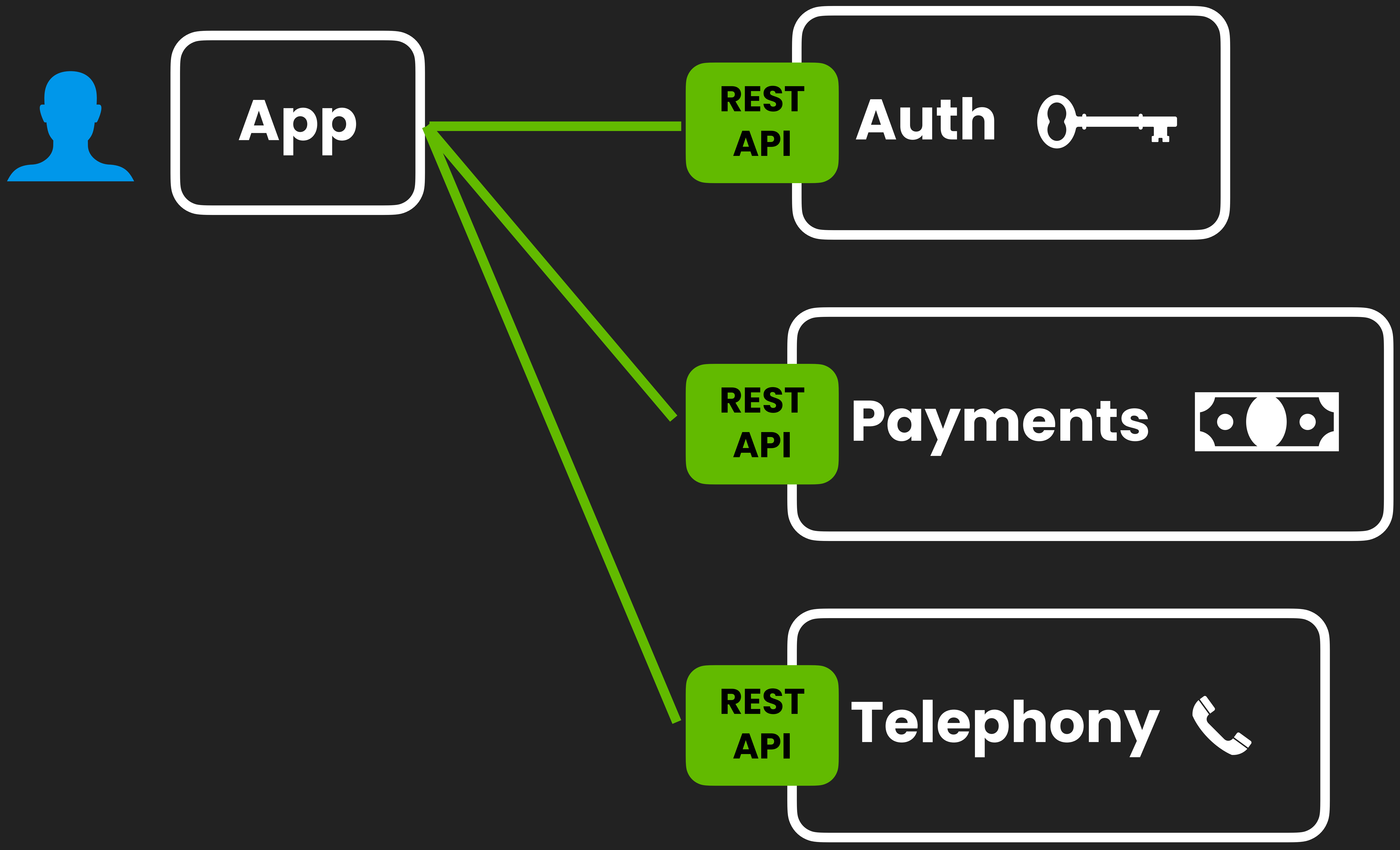
Leveraging SRE and Observability Techniques

for the Wild World of Building on LLMs

@cyen
@honeycombio

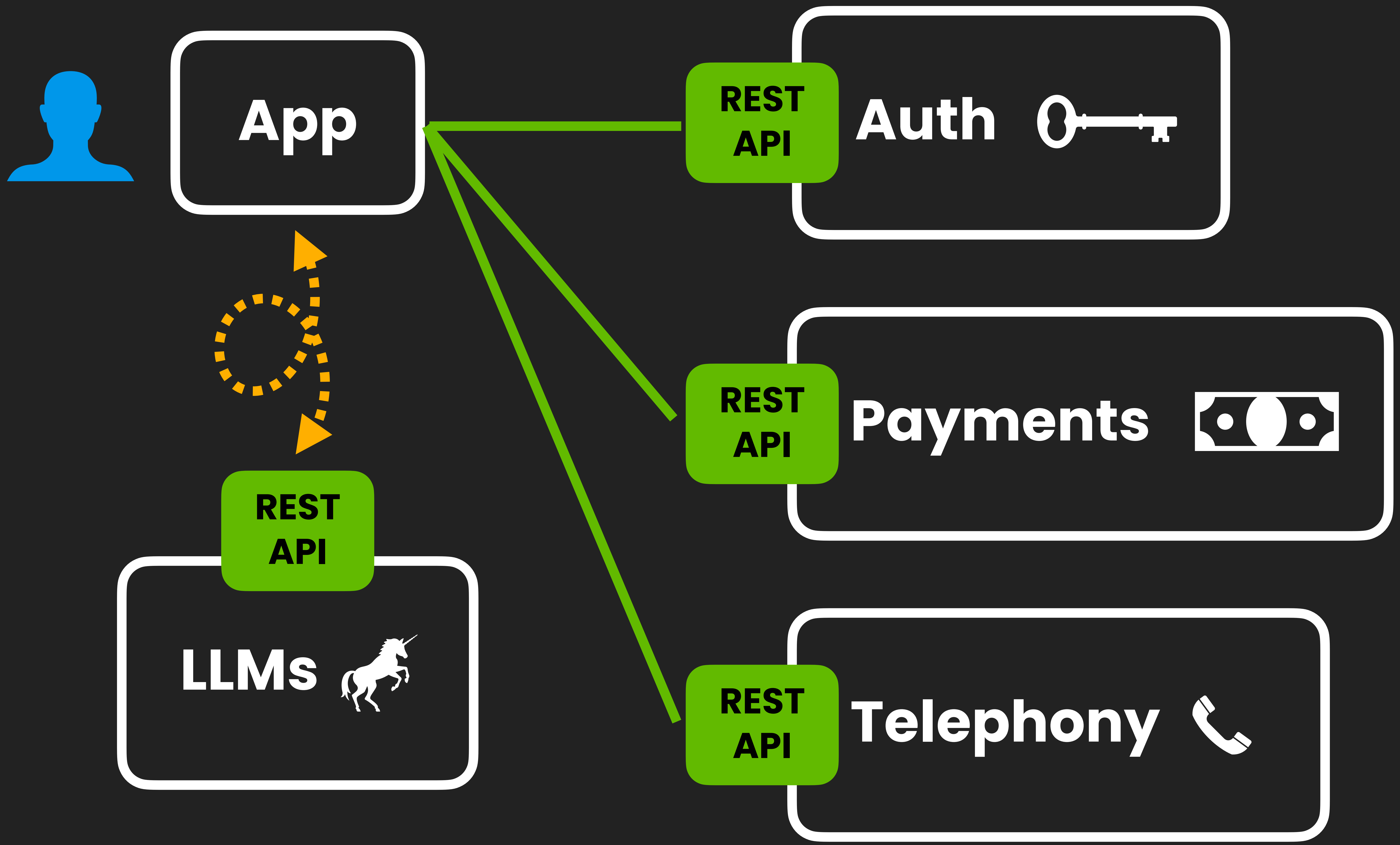
LLMs ≈ like APIs we know and 💛



- **Well-formed inputs** according to a spec
- **Cleaned-up user** inputs
- **Well-formed outputs** according to a spec
- **Standard** protocols (e.g. HTTP, SMTP)

= **testable mockable**

LLMs != like APIs we know and 💛



~~predictable~~

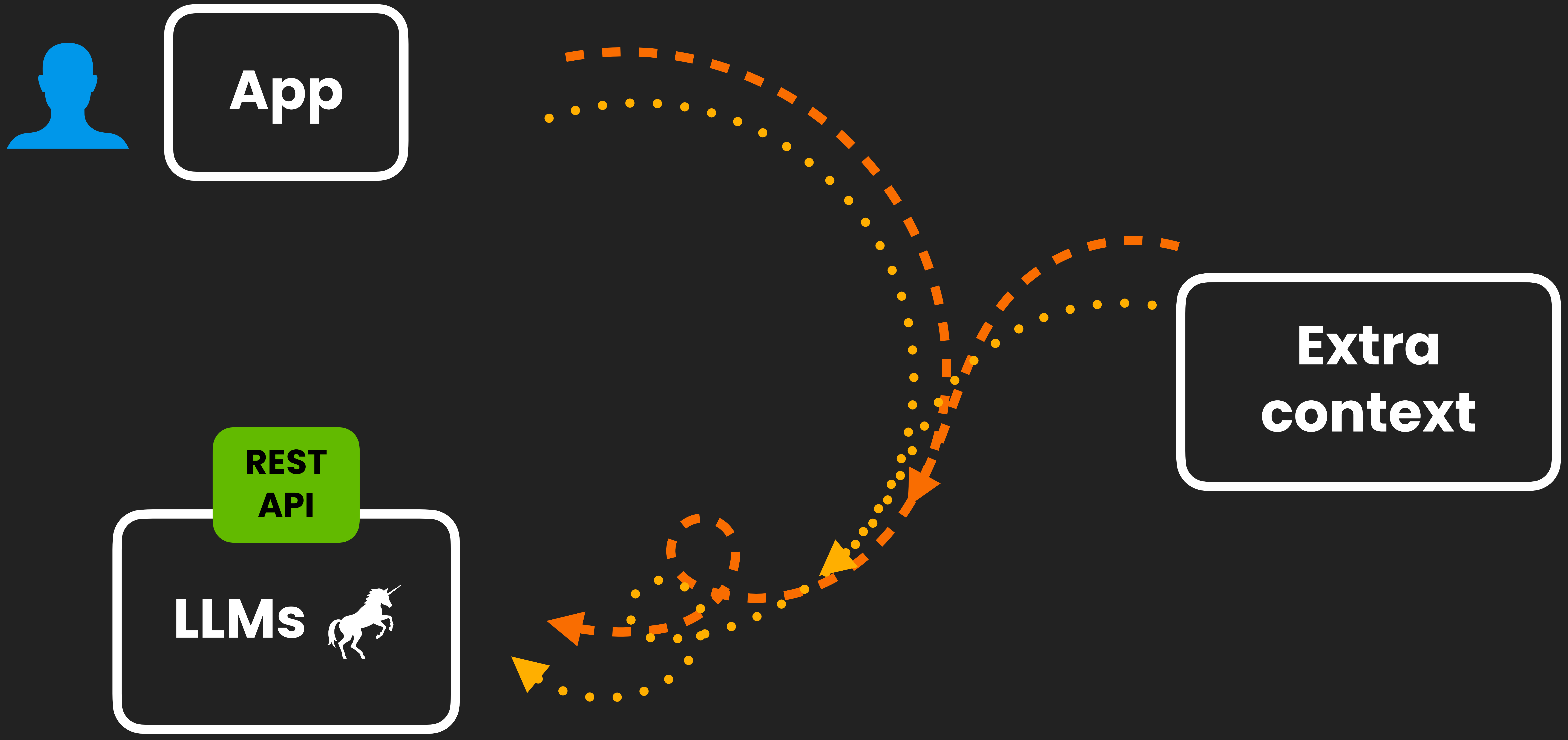
∴ ~~testable~~

∴ ~~mockable~~

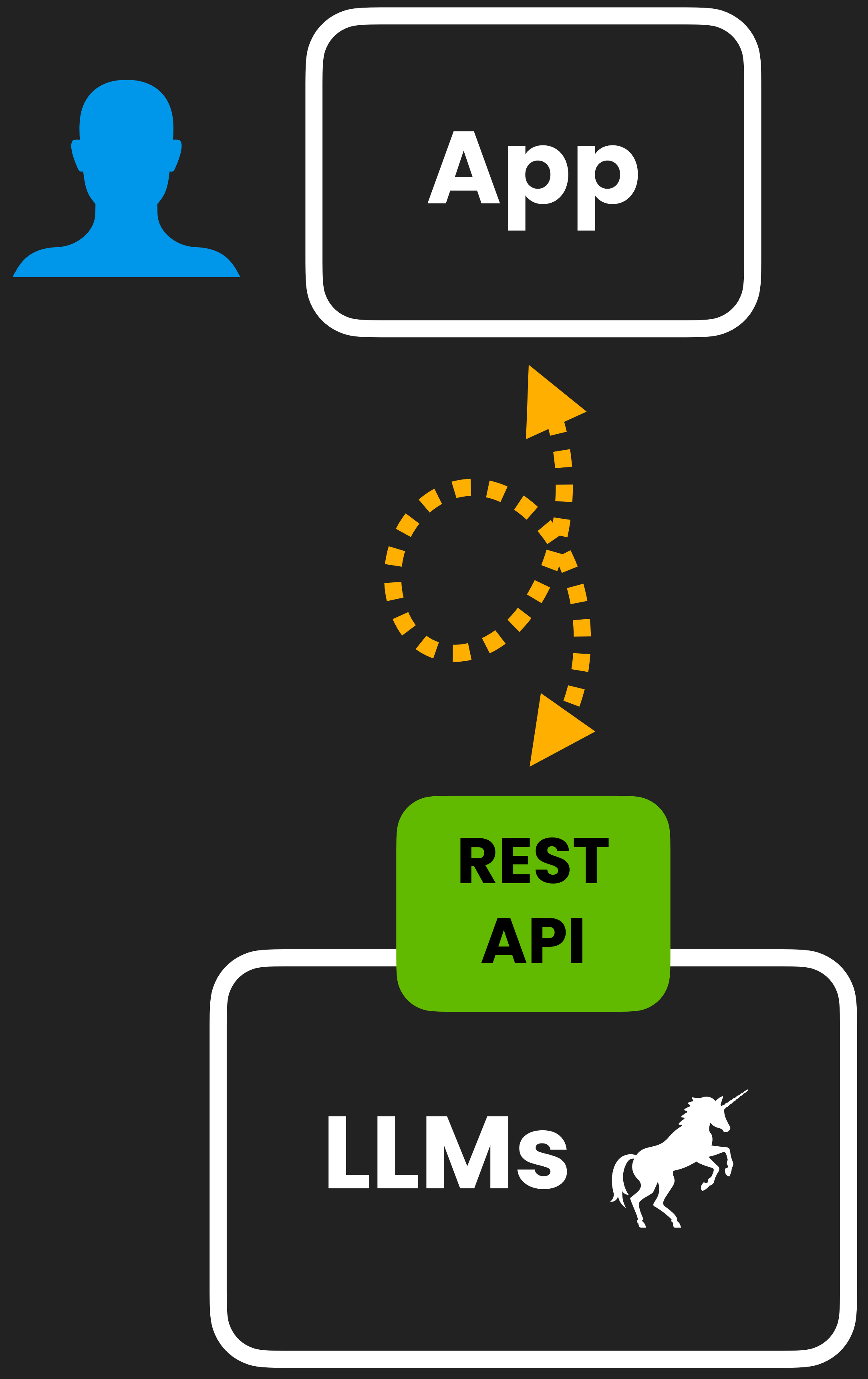
LLMs != like APIs we know and 💛

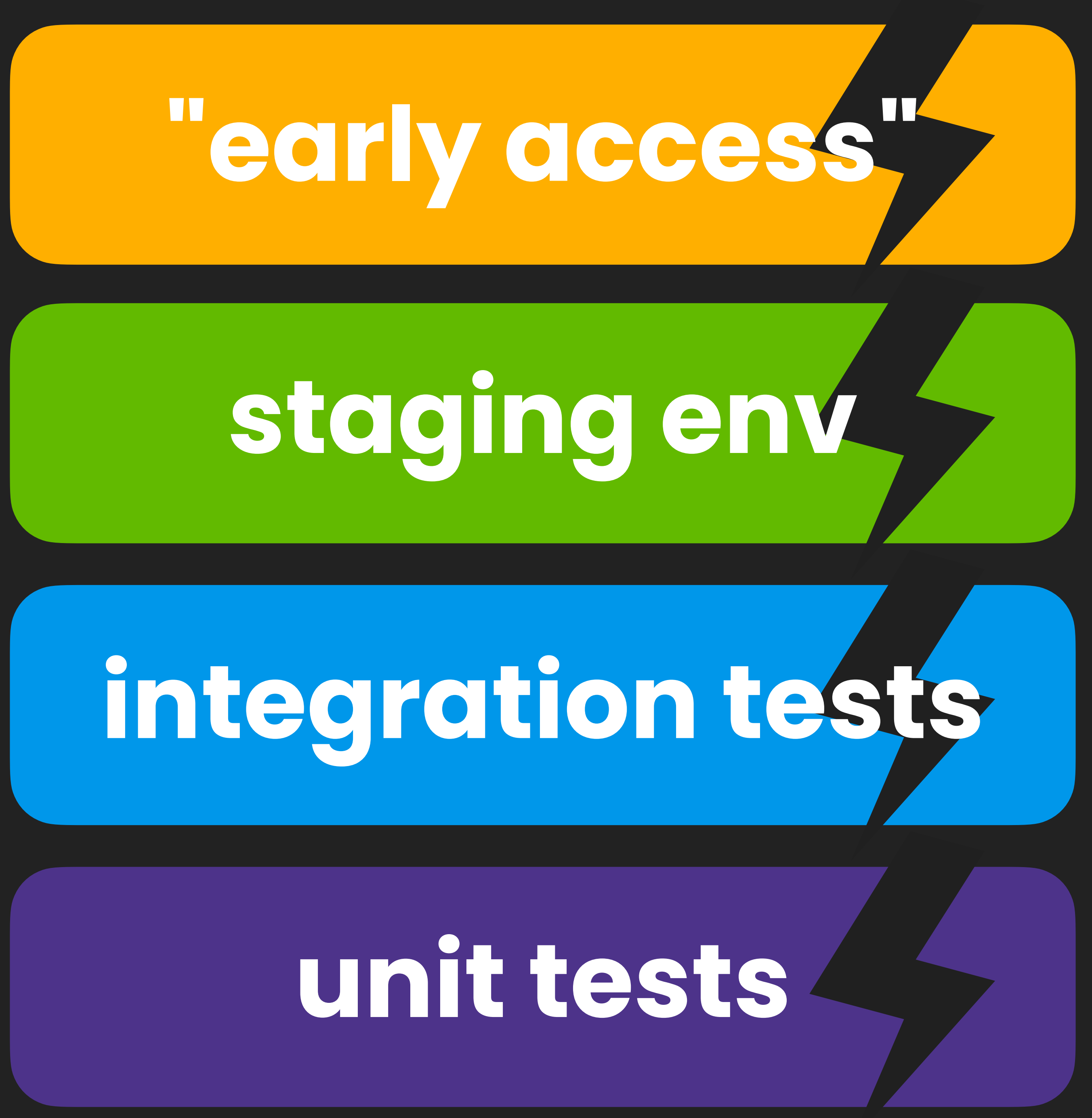
	Normal APIs	LLMs
`unit tests`	can conceivably scope the range of inputs	intentionally invites free-form, natural-language input from users
`reproducible (AKA mockable)`	deterministic + (ideally) idempotent	subject to change ("drift" in model behavior) via public API access
`explainable (AKA debuggable)`	based on spec, can understand how a change in input → change in output	prompting can yield very different responses through small, subtle changes to prompt

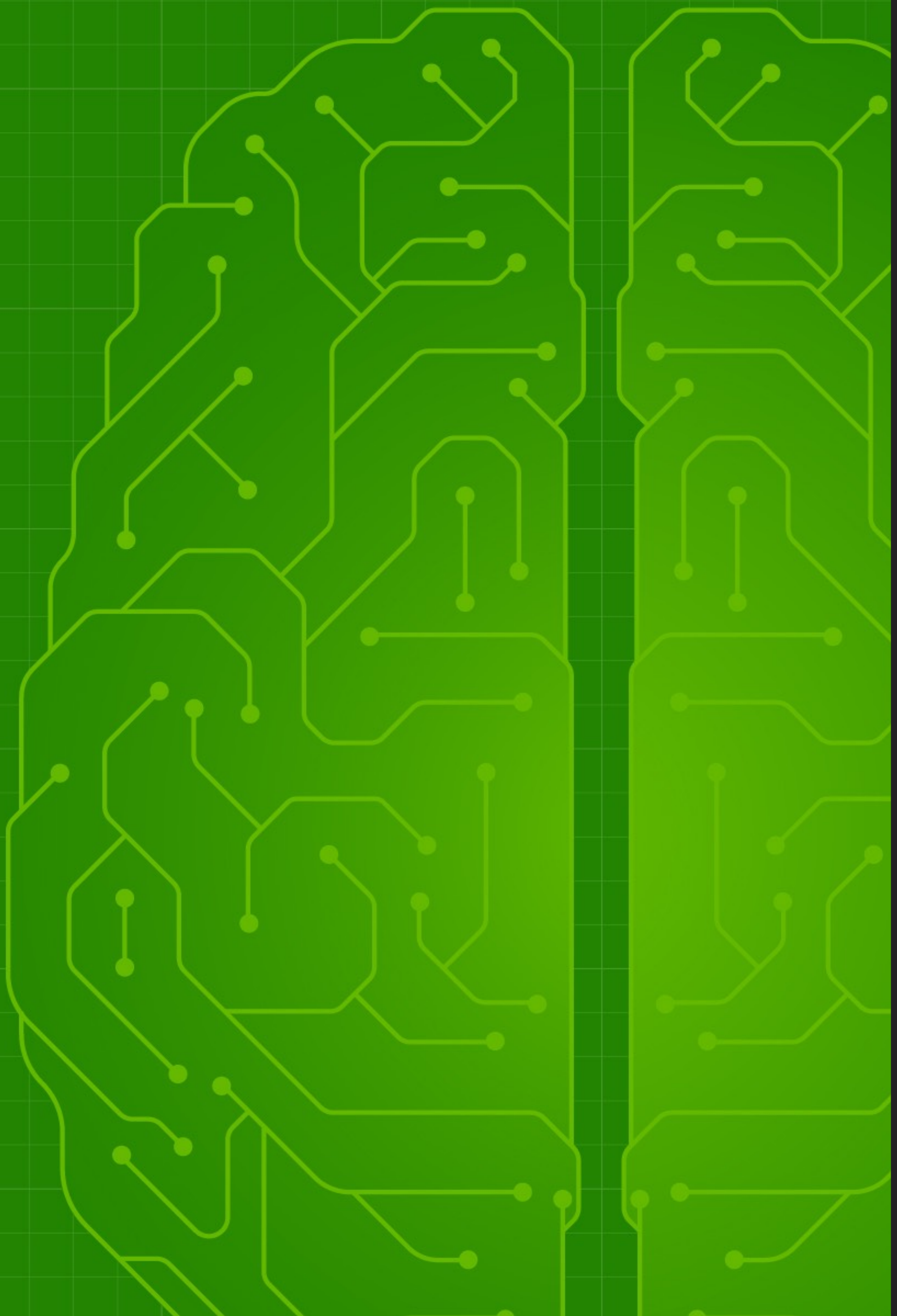
LLMs: even more unpredictability



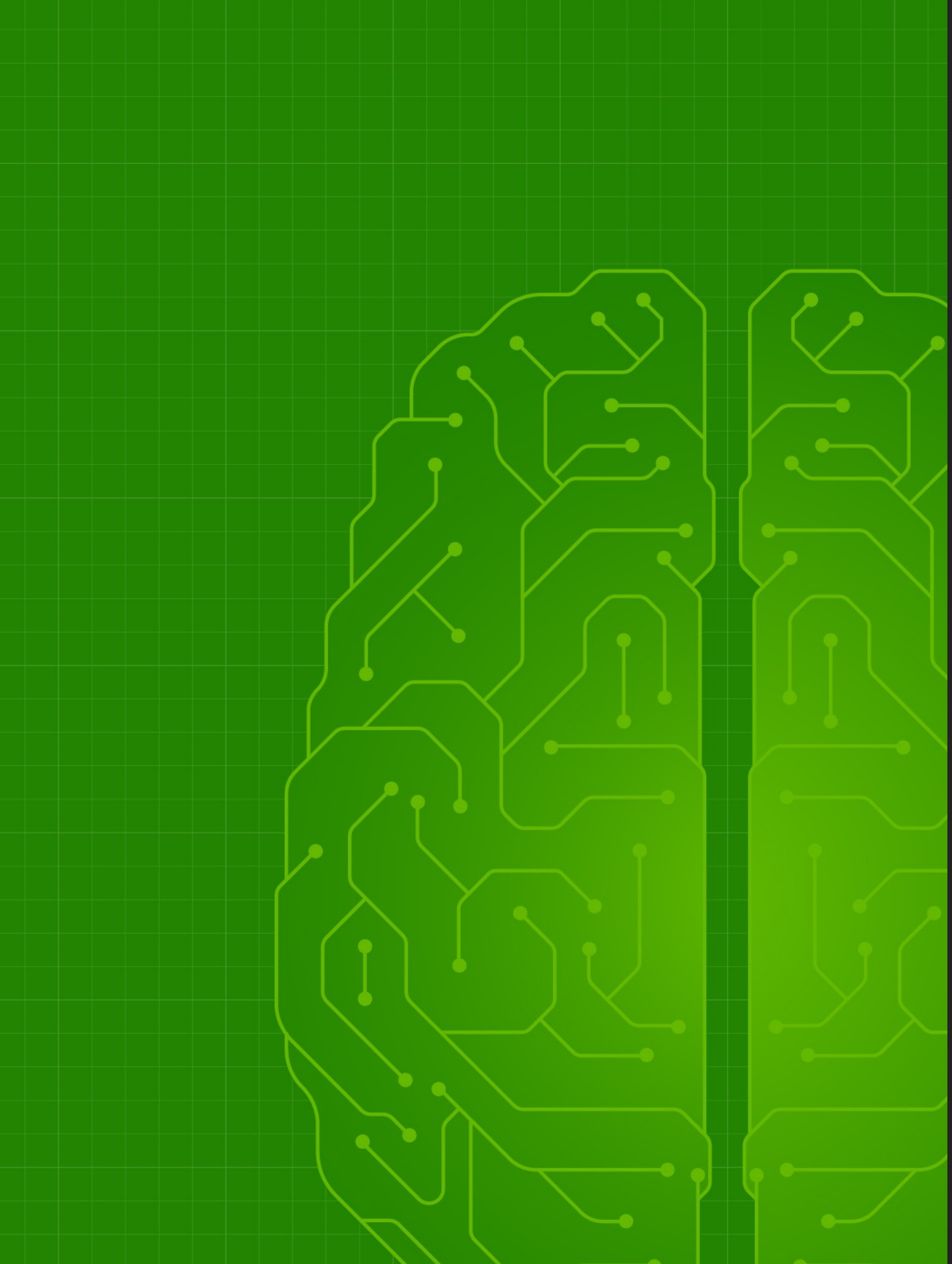
LLMs: how do we define "correct"?





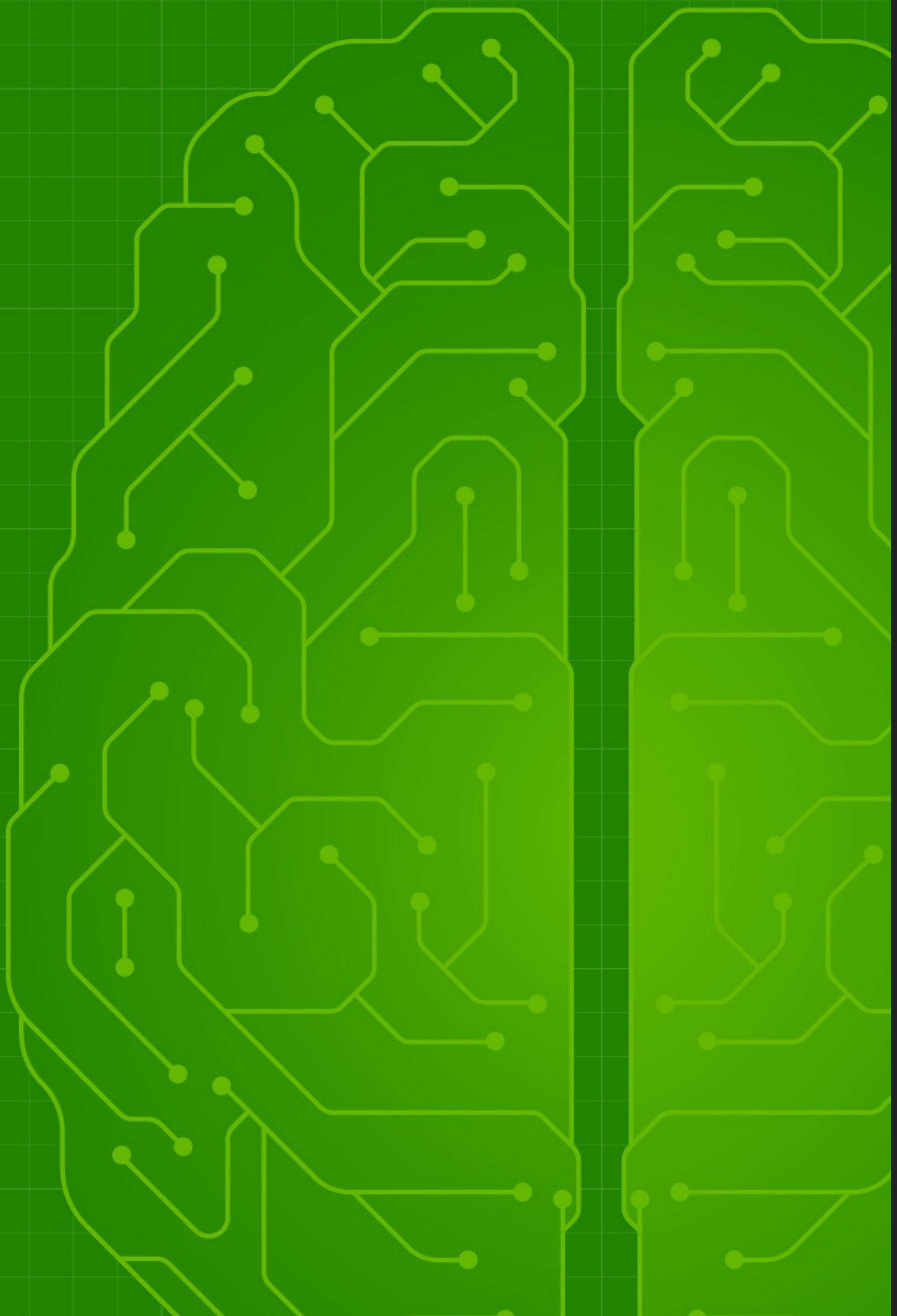


observability



AKA: an understanding the behavior of a system based on knowledge of its external outputs.

observability

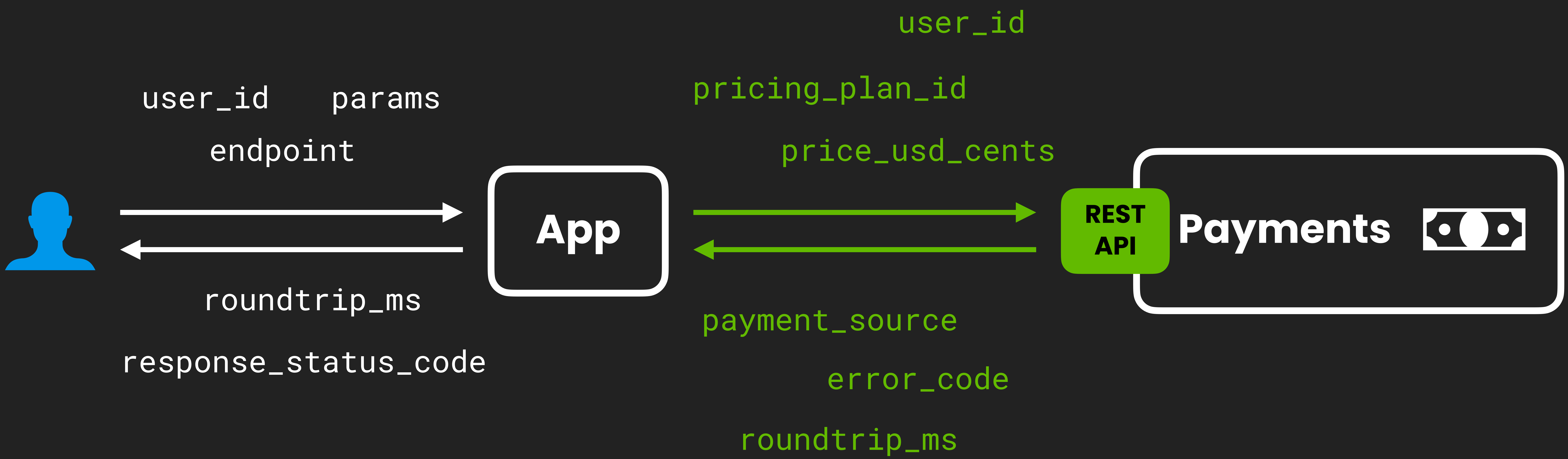


expected actual

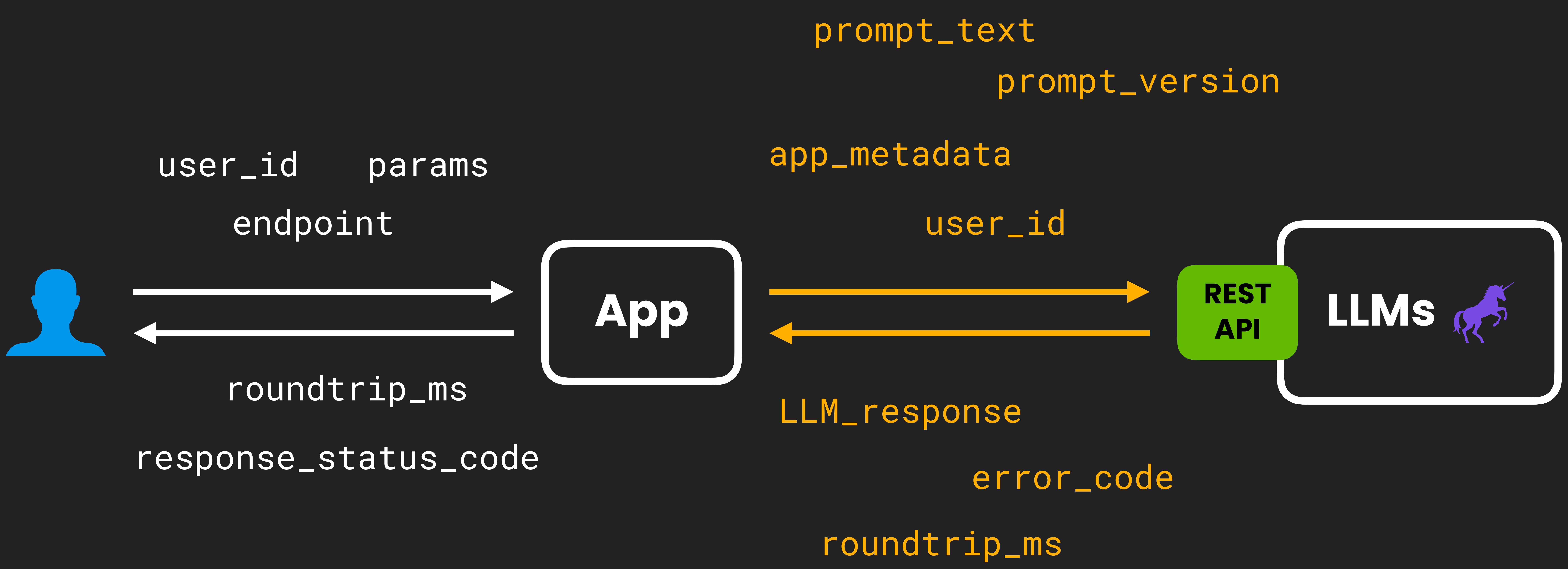
(especially in prod!)

observability

Observability: what's in the box?



Observability: what's in the box?

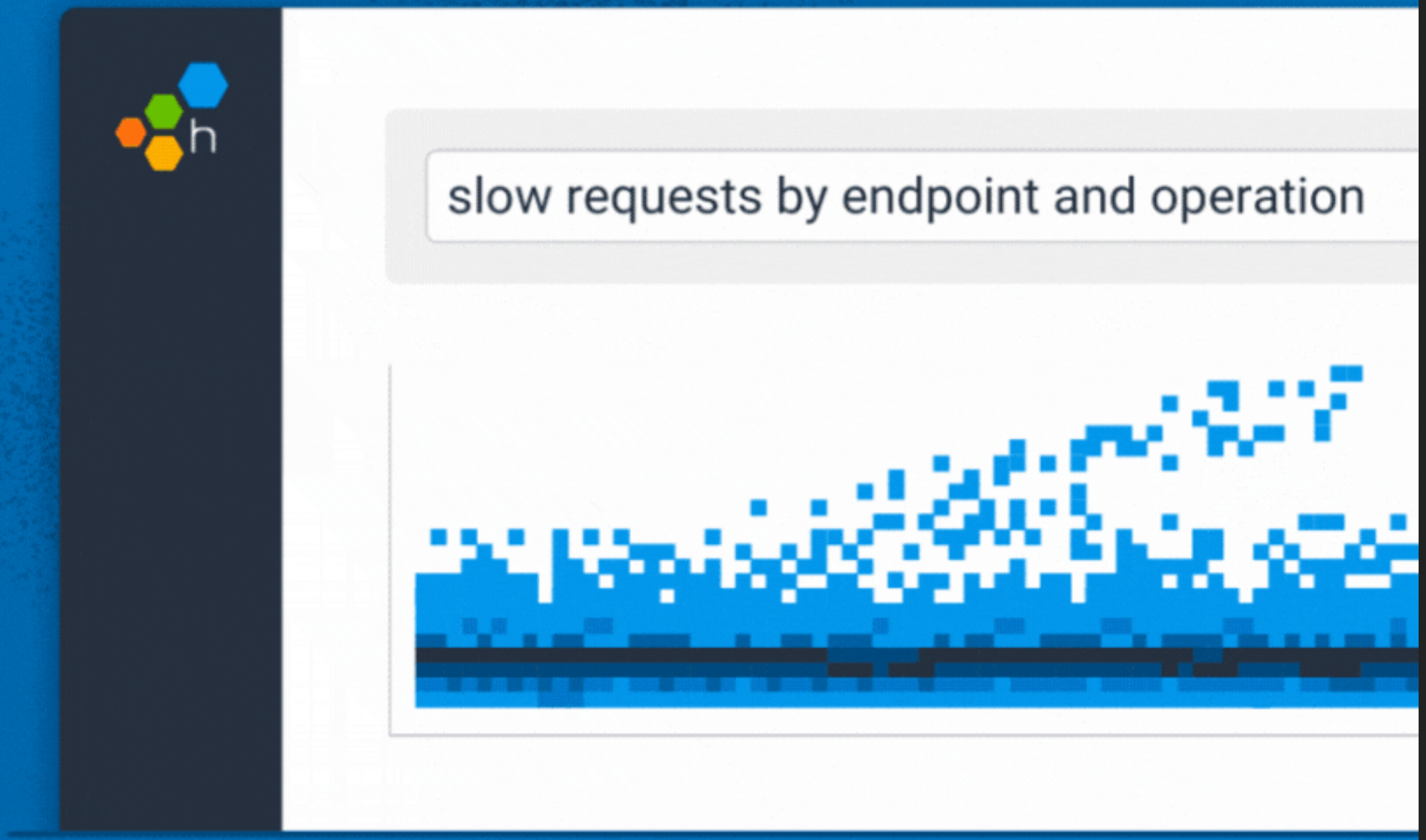


Observability: ∞ feedback loops

IDEATE → WRITE → TEST → RELEASE → OBSERVE

TEST

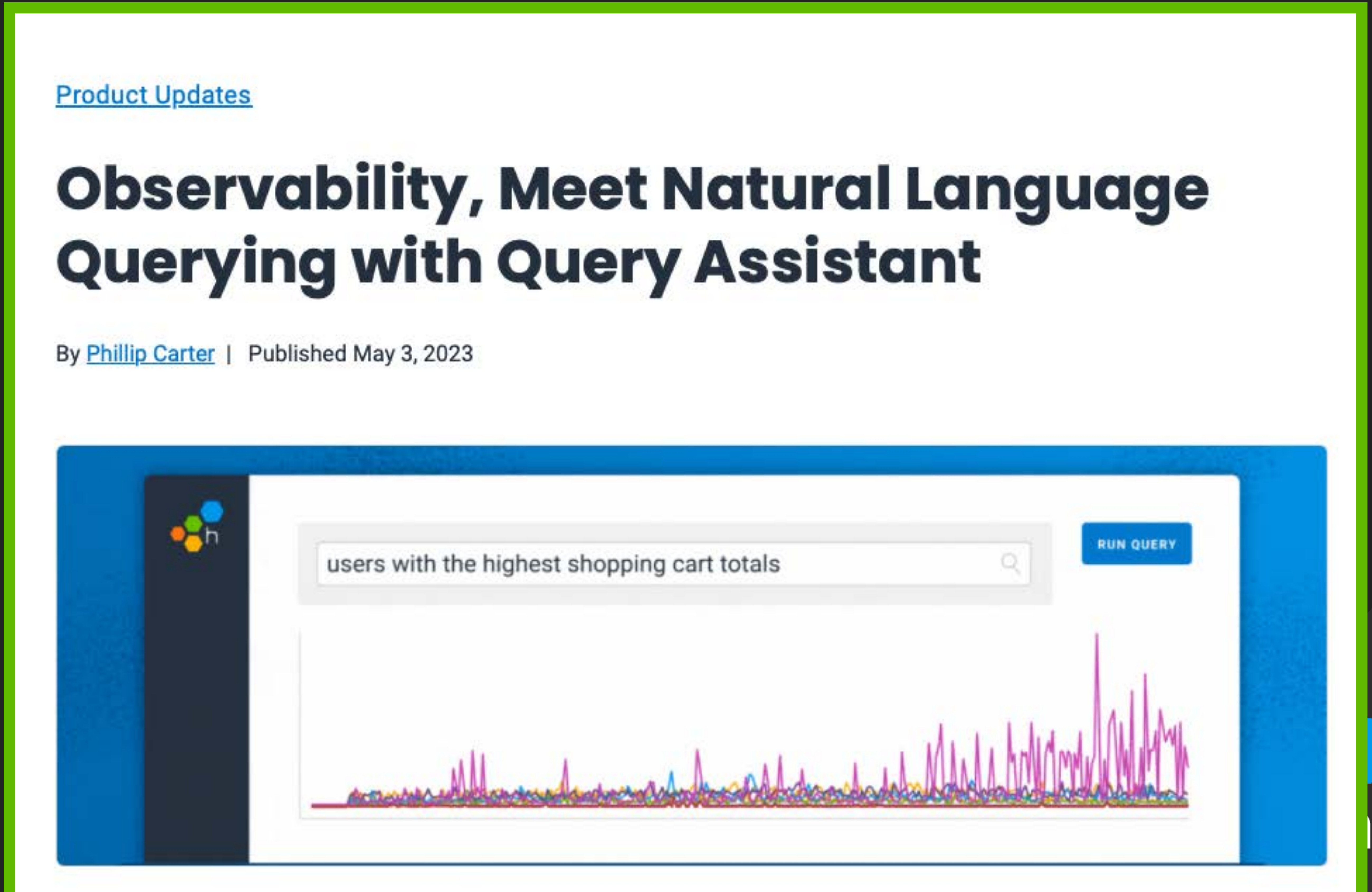
OBSERVE



Why believe me?

Query Assistant: timeline

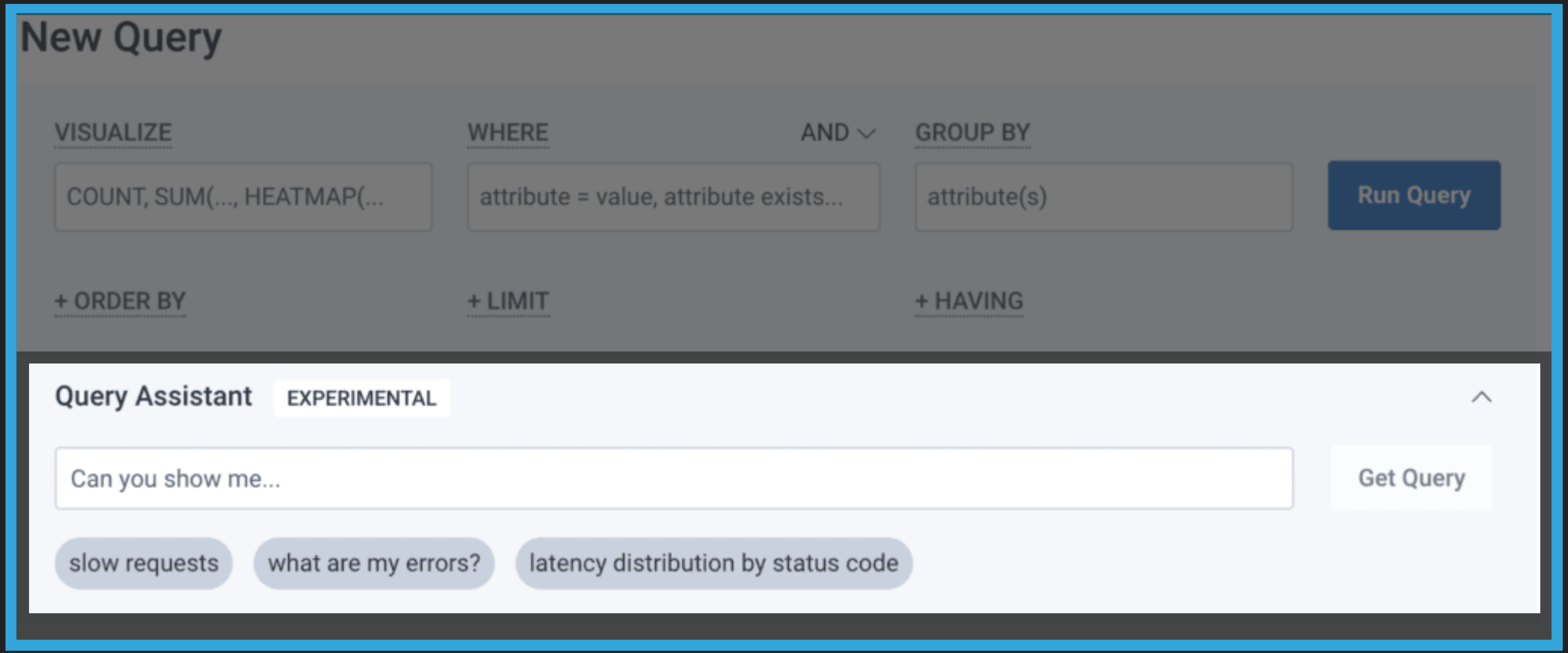
- 6 weeks of development
- May 2023
- 8 weeks of iteration

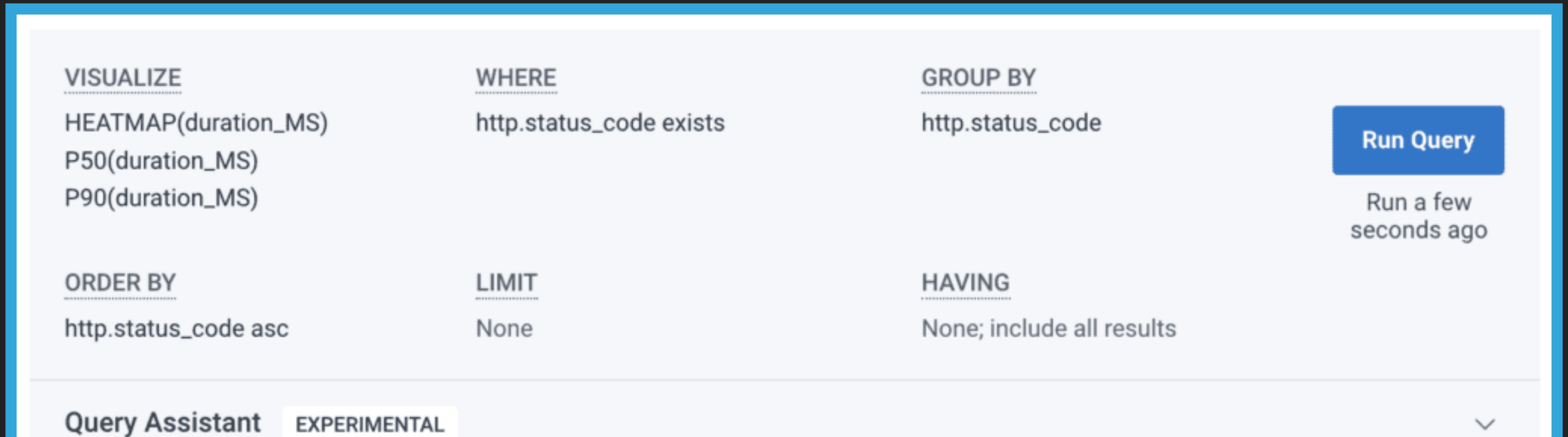

Product Updates

Observability, Meet Natural Language Querying with Query Assistant

By Phillip Carter | Published May 3, 2023

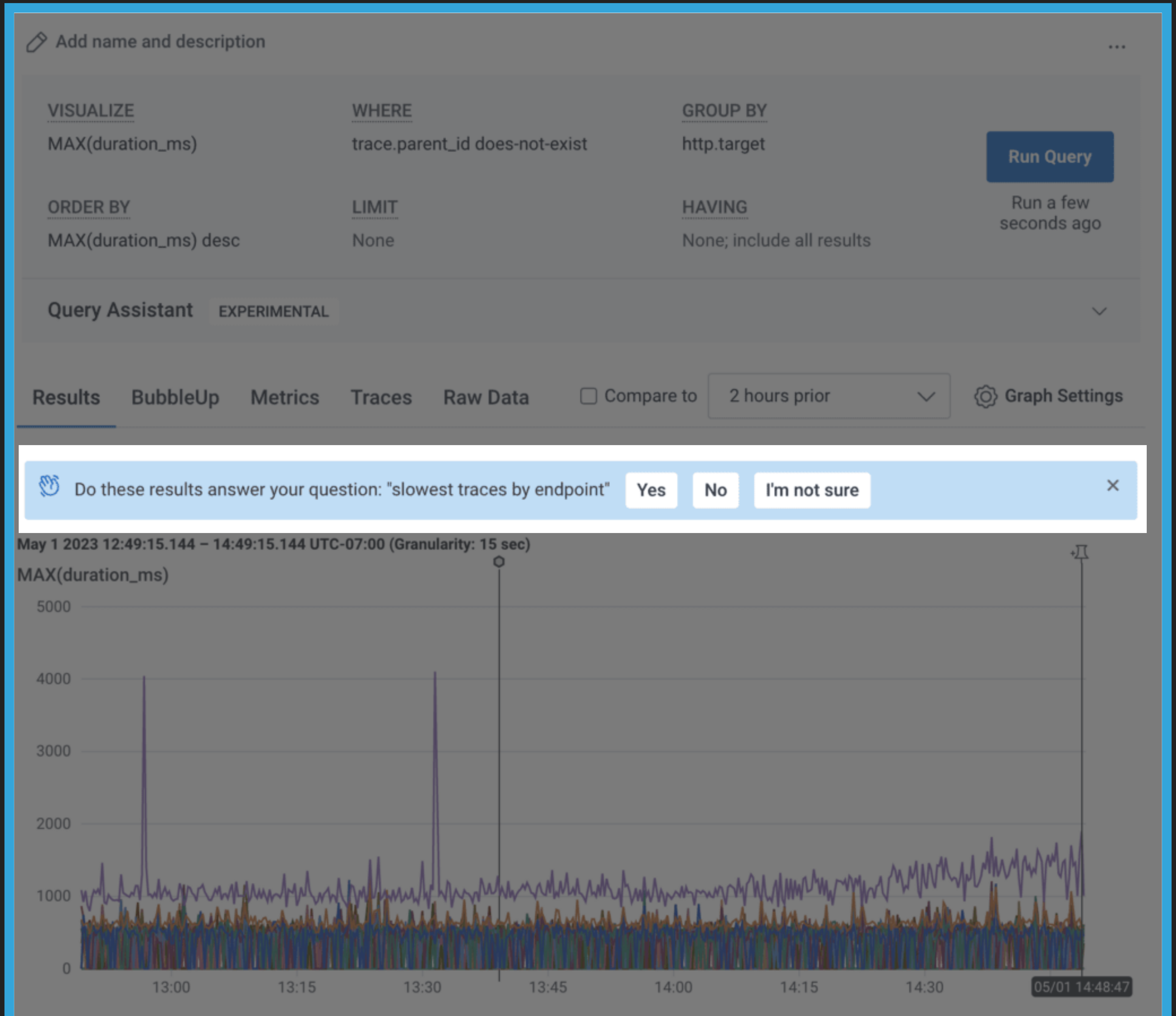
Query Assistant: goals

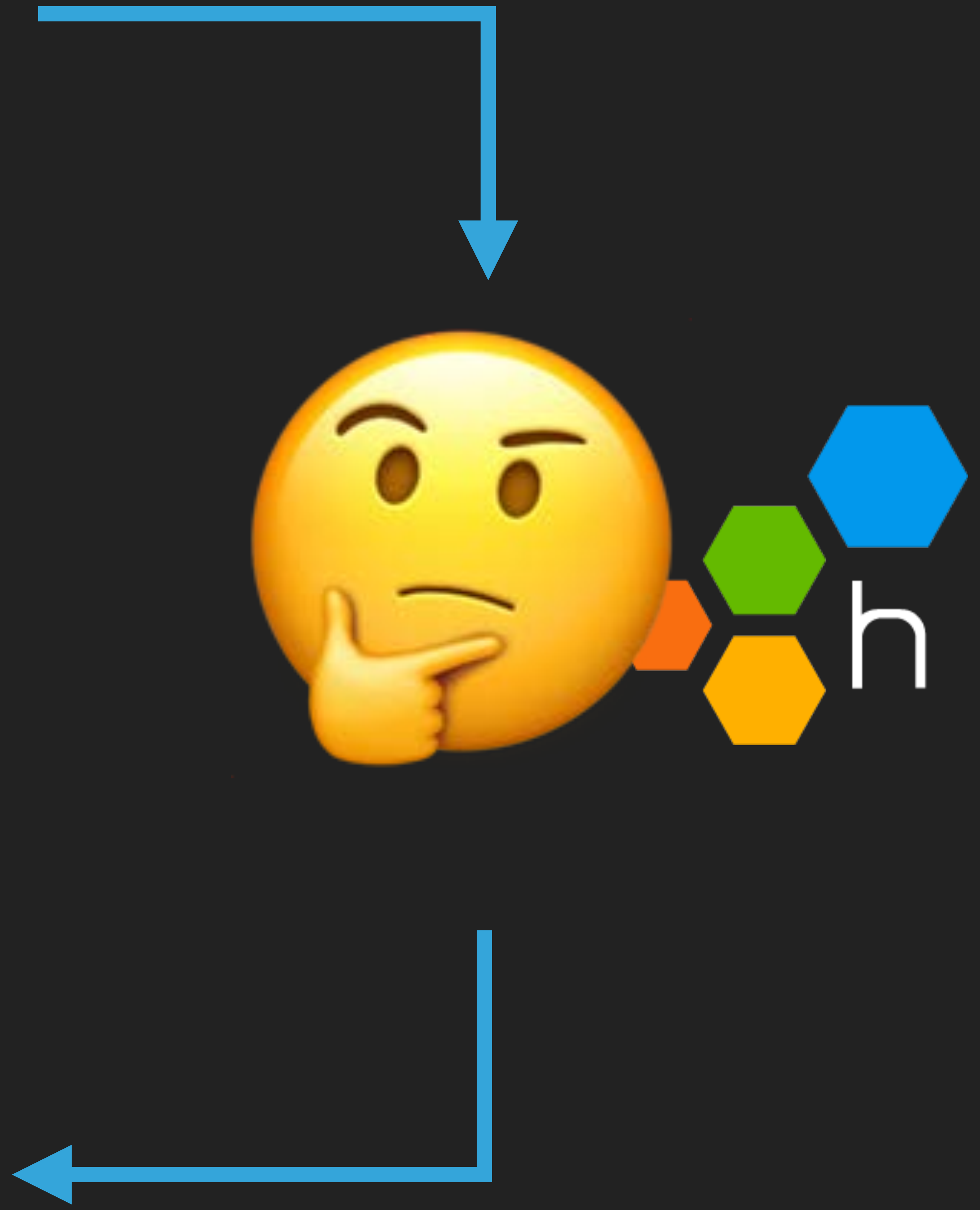




"What's the 95th percentile latency on the `/checkout` endpoint?"

Query Assistant: goals

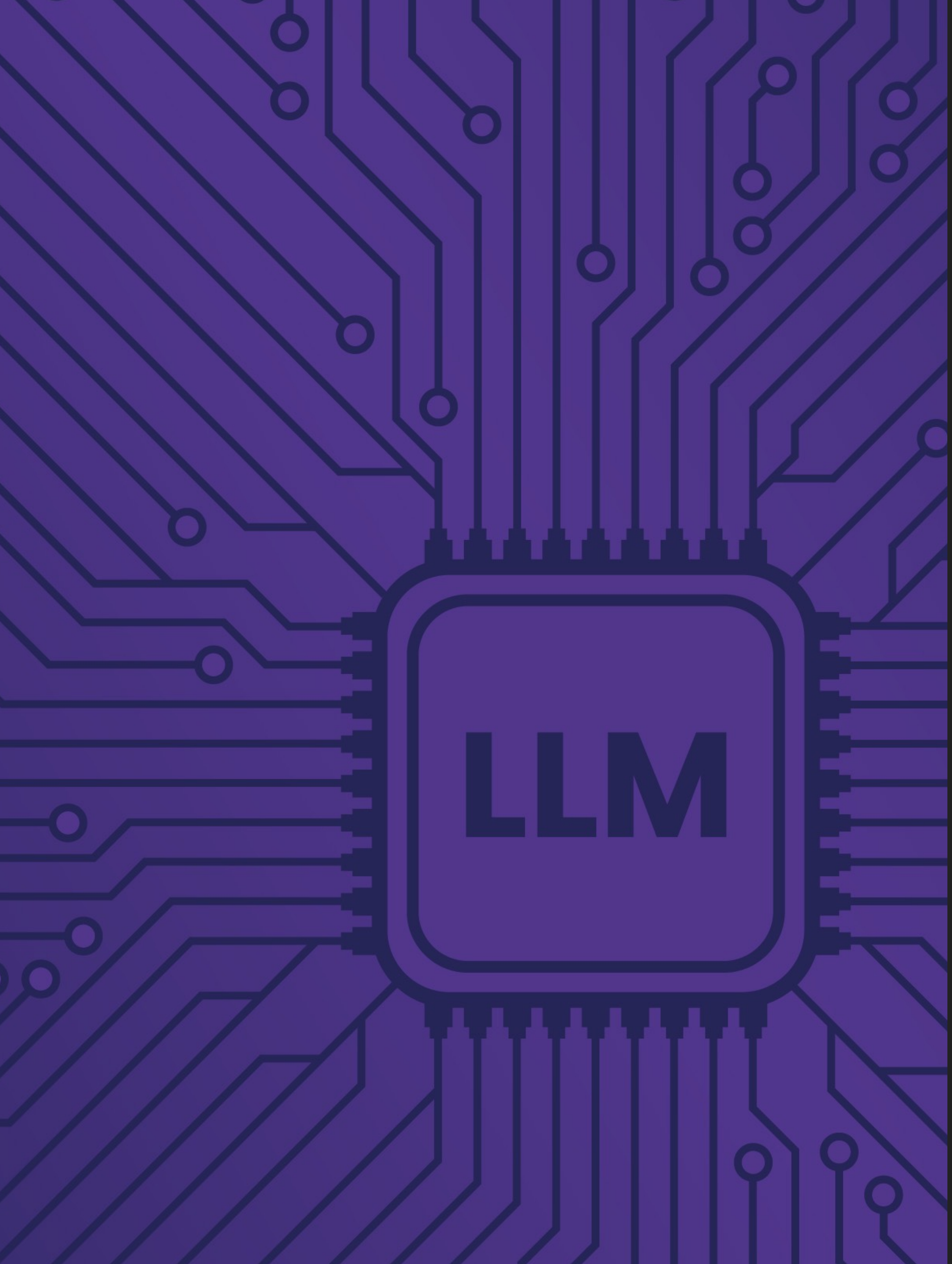




Laws of building on LLMs

- Failure will happen—it's a question of when, not if.
- Users will do things you can't possibly predict.
- You will ship a "bug fix" that breaks something else.
- You can't really write unit tests for this (nor practice TDD)
- Latency is often unpredictable
- Early access programs won't help you

https://honeycomb.io/blog/hard-stuff-nobody-talks-about-llm



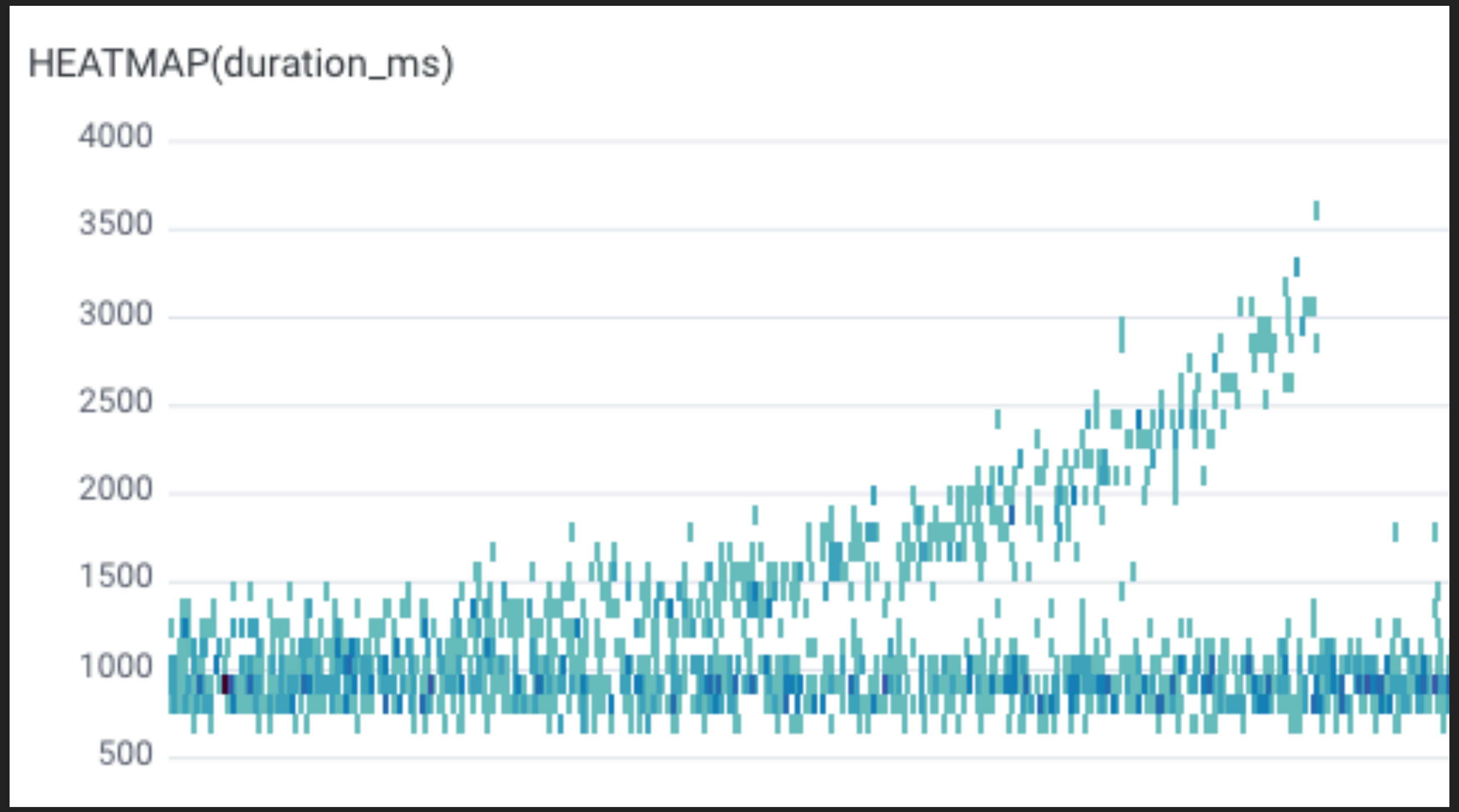
OK, so

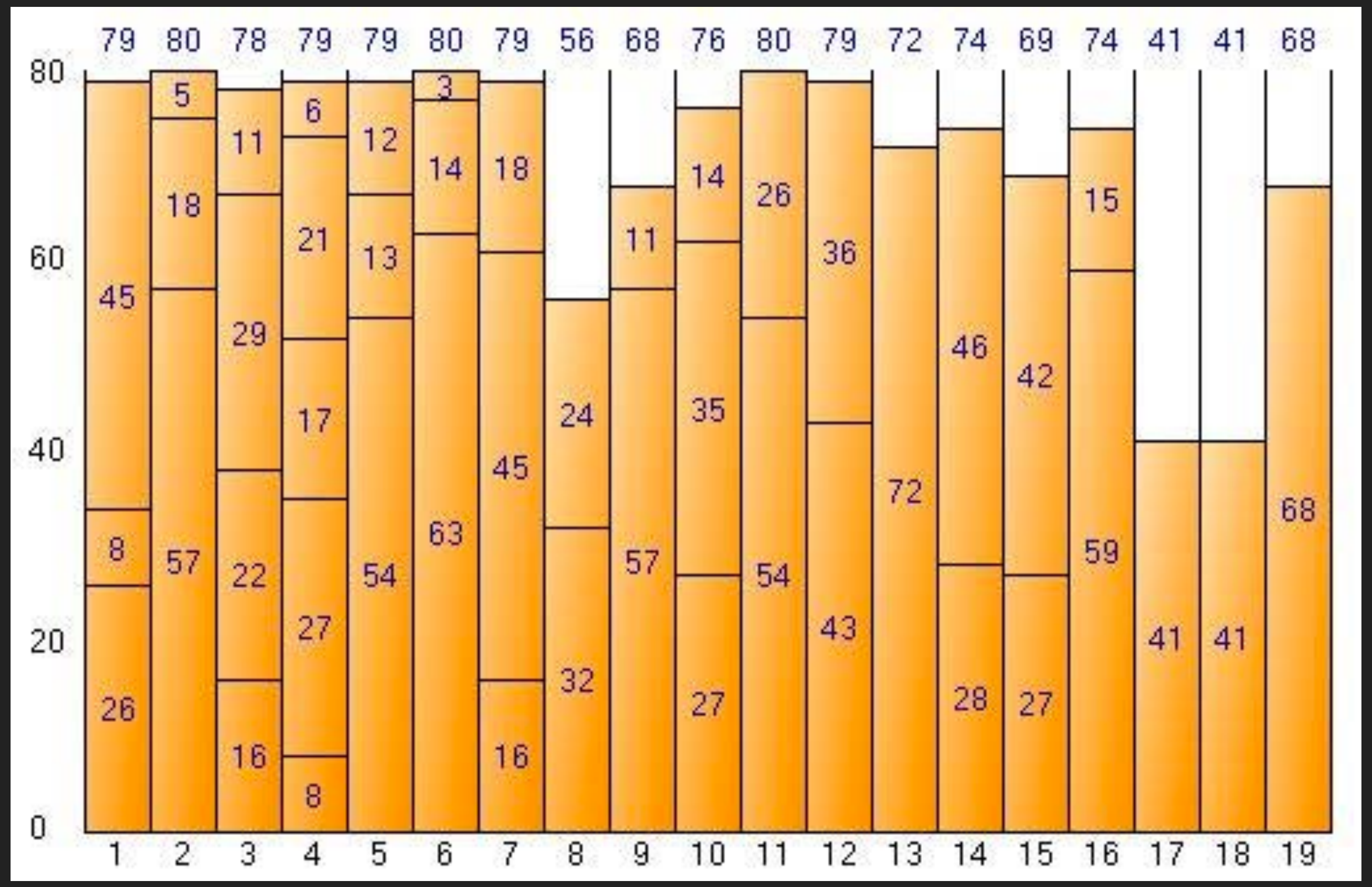
How *do* we go forward?

Instrumentation ~= docs and tests

capture data for your hypotheses

Instrumentation ~= docs and tests





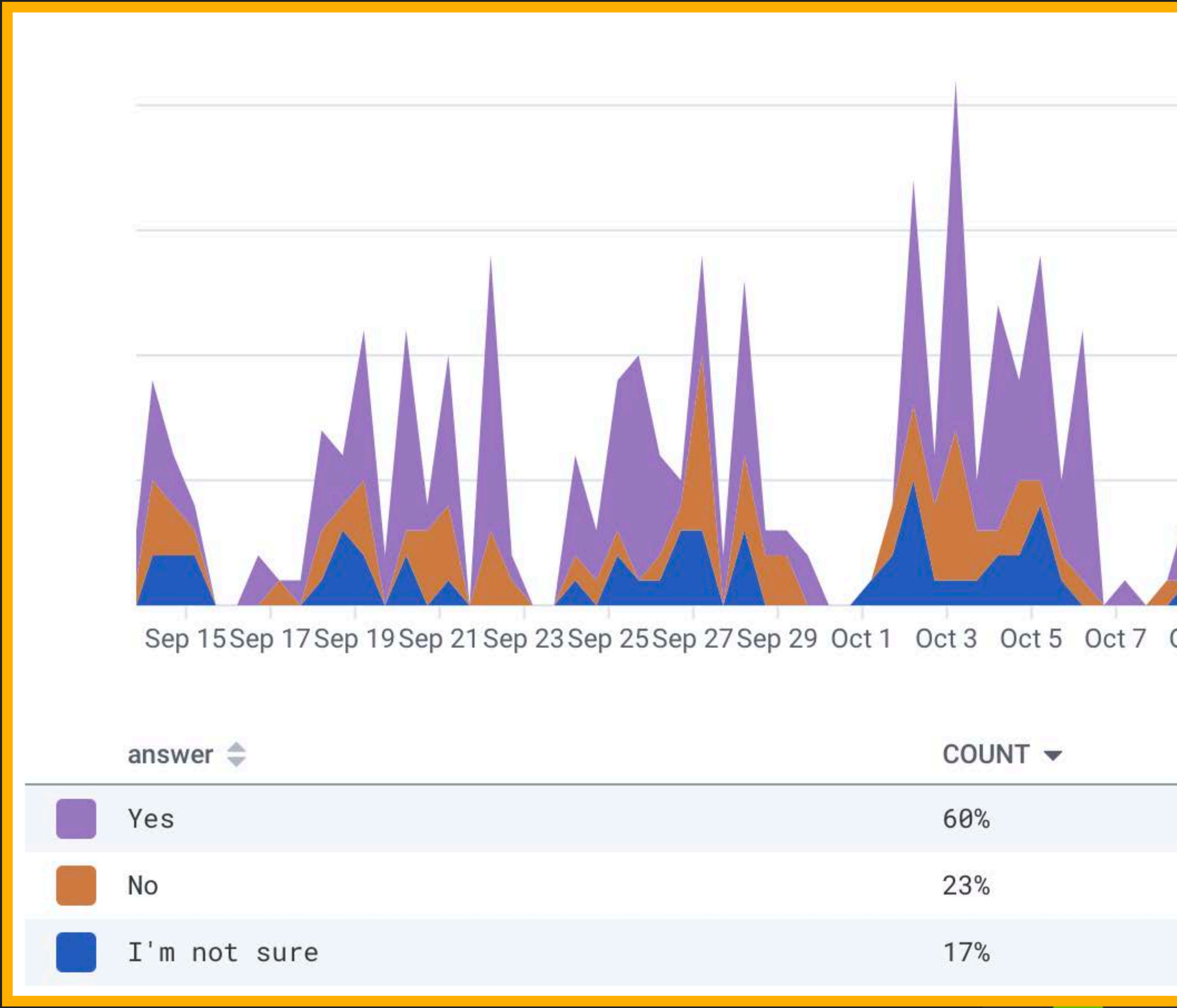
capture data for your hypotheses

Instrumentation for LLMs

- user/team IDs
- full user input string
- add'l product context for prompt
- token usage
- LLM latency
- full LLM response
- parse and/or validation errors
- user feedback

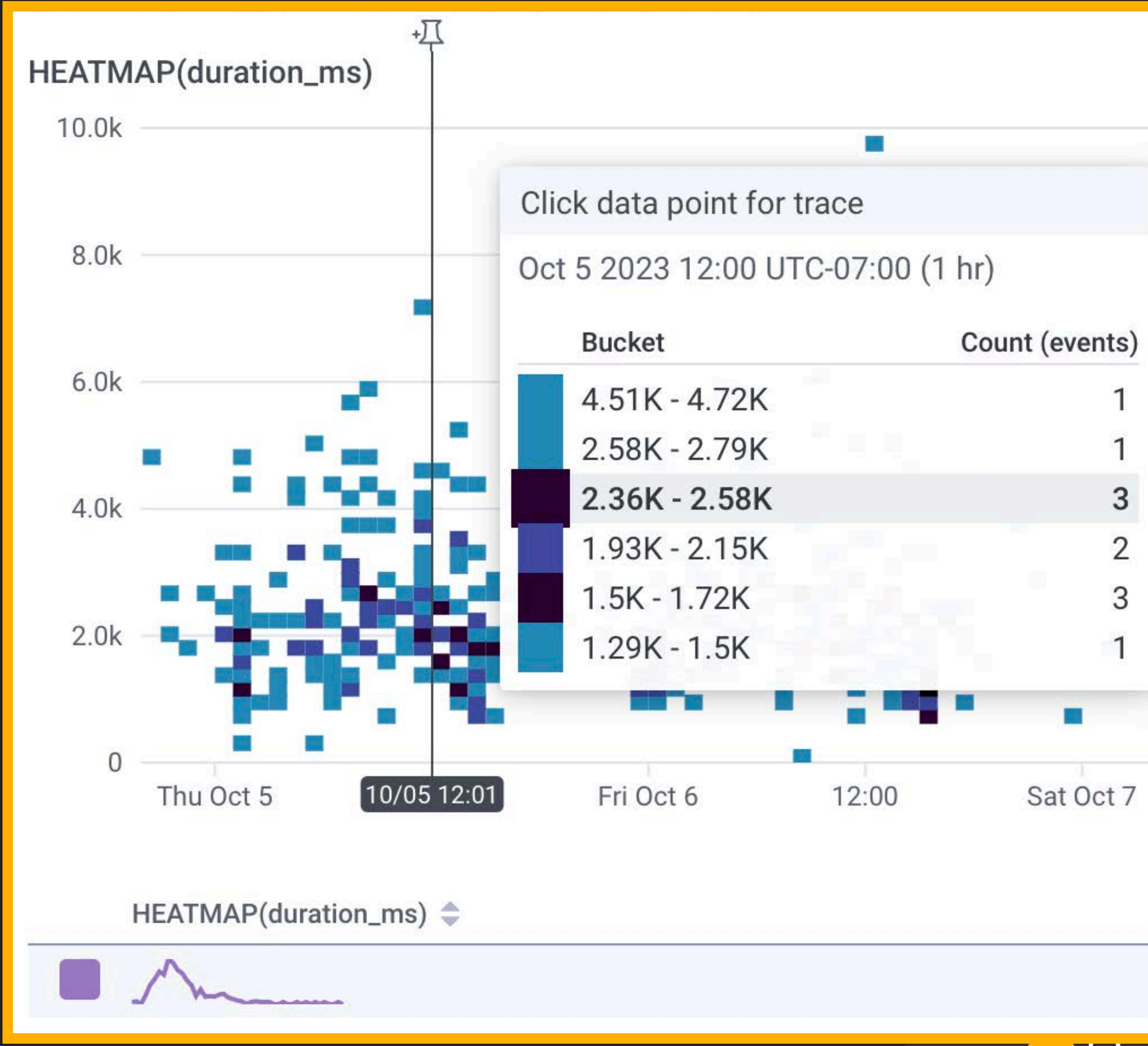
Instrumentation for LLMs

- user/team IDs
- full user input string
- add'l product context for prompt
- token usage
- LLM latency
- full LLM response
- parse and/or validation errors
- **user feedback**



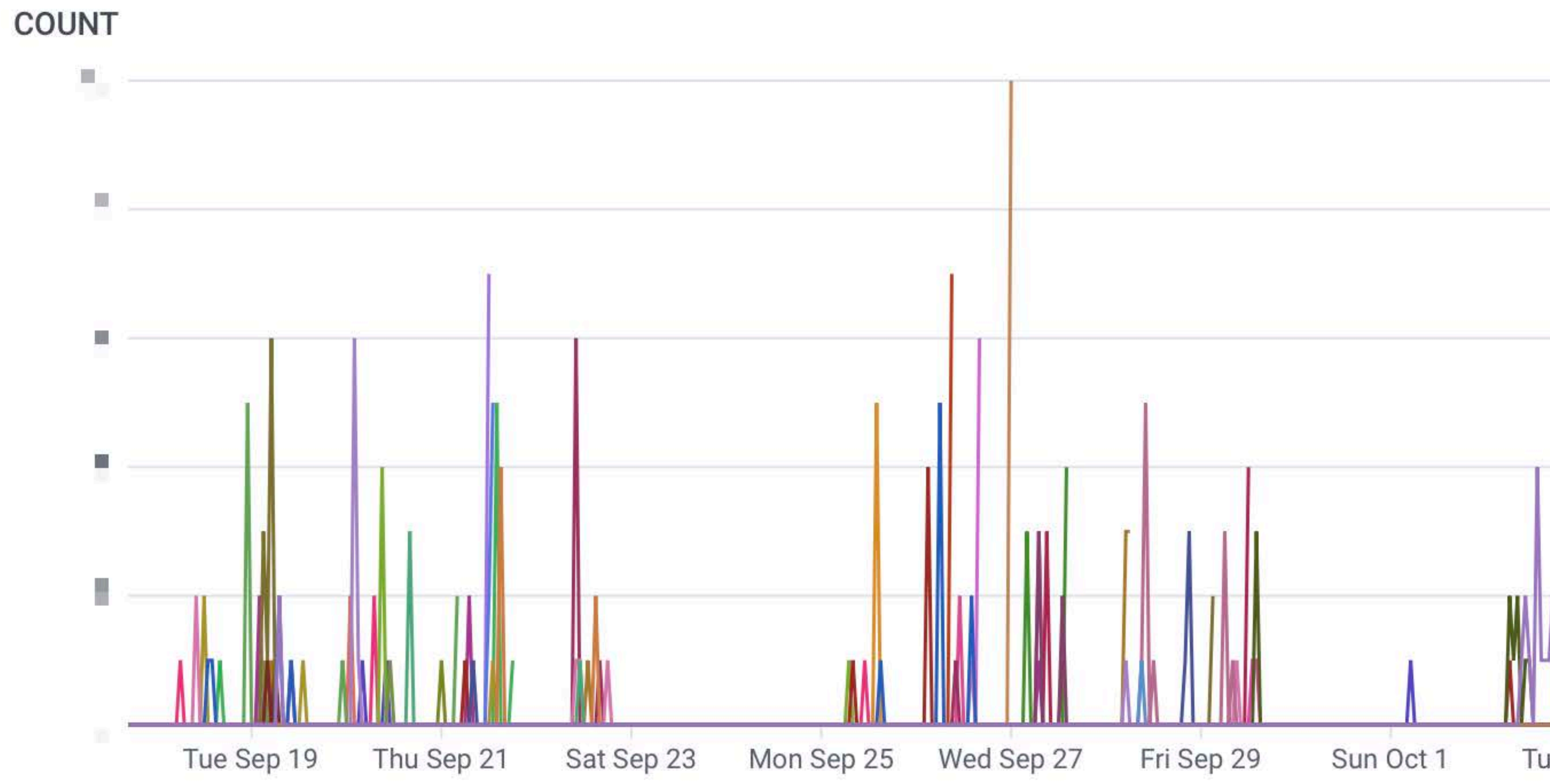
Instrumentation for LLMs

- user/team IDs
- full user input string
- add'l product context for prompt
- token usage
- **LLM latency**
- full LLM response
- parse and/or validation errors
- user feedback



Instrumentation for LLMs

- **user/team IDs**
- full user input string
- **add’l product context for prompt**
- **token usage**
- LLM latency
- full LLM response
- parse and/or validation errors
- user feedback



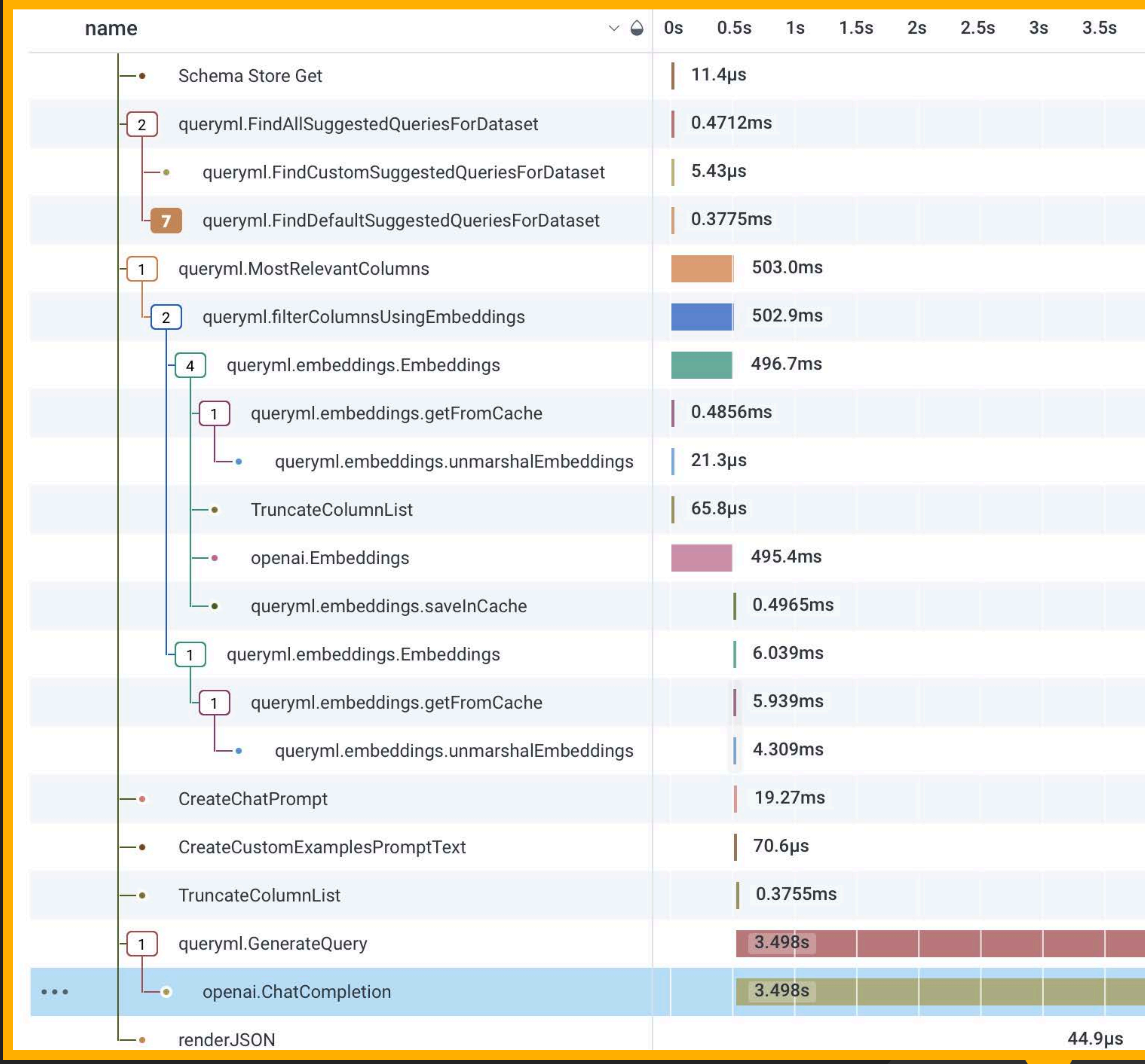
app.team.id	app.user.id	app.nlq.input_tokens_used.bucketed
13	85170	1500-1750
13	48009	1500-1750
62,519	152741	1750-2000
84,002	171917	1750-2000
13	43516	1500-1750
84,002	171918	1750-2000
48,775	142184	2000-3000

Instrumentation > EXCEPTIONS

- user/team IDs
- full user input string
- add’l product context for prompt
- token usage
- LLM latency
- full LLM response
- parse and/or validation **errors**
- user feedback

Instrumentation for LLMs

- user/team IDs
- full user input string
- add'l product context for prompt
- token usage
- LLM latency
- full LLM response
- parse and/or validation errors
- user feedback



Tapping into

Emerging behaviors

DEV 🔥

WRITE → TEST → COMMIT → WRITE → TEST → COMMIT
→ WRITE → TEST → COMMIT → WRITE → TEST → COMMIT
→ WRITE → TEST → COMMIT → WRITE → TEST → COMMIT
→ WRITE → TEST → COMMIT → WRITE → TEST → COMMIT

DEV

write lots of code
service ownership
developers on call
test in production

DEV

identify levers impacting logical branches in code (debuggability + reproducibility)

compare `expected` vs `actual`

fail fast / fail first;
embrace fast feedback loops

PROD

instrument code with intention

inspect results after changes go live; watch for deviations

ship to prod quickly (CI/CD);
expect to iterate

A truth in all software systems,
but never more true than with LLMs:

Software behaves in unpredictable, emergent ways, and the important part is observing your code **as it's running in production, while users are using it.**

Let’s zoom in on

Service Level Objectives

SLOs: a quick definition

Service Level Objectives codify what it means to **"deliver great service"**

- "Key user flows like cart checkout should complete quickly and reliably"
- "99.9% of shopping cart checkout attempts complete error-free in < Xs"

Laws of building on LLMs

- Failure will happen—it’s a question of when, not if.
- Users will do things you can’t possibly predict.
- You will ship a "bug fix" that breaks something else.
- You can’t really write unit tests for this (nor practice TDD)
- Latency is often unpredictable
- Early access programs won’t help you

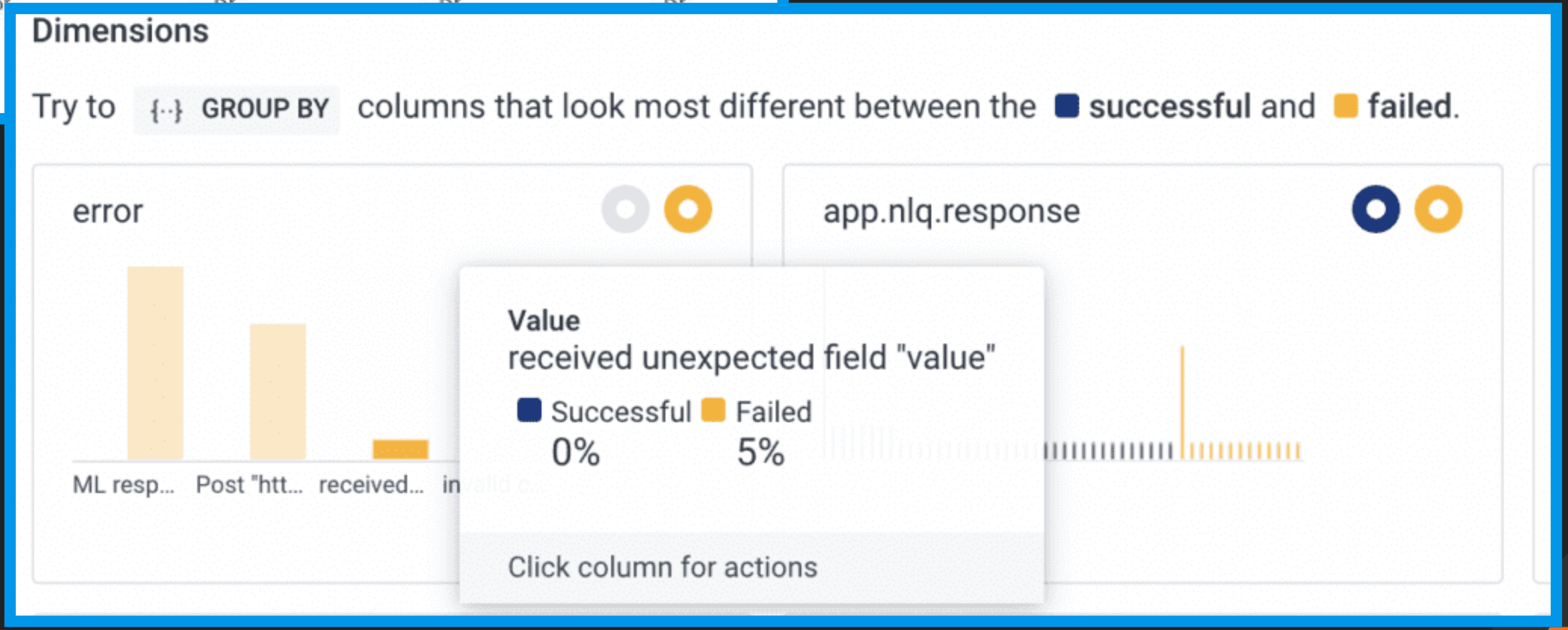
- **LINK TO: hard things about hard things blog post**

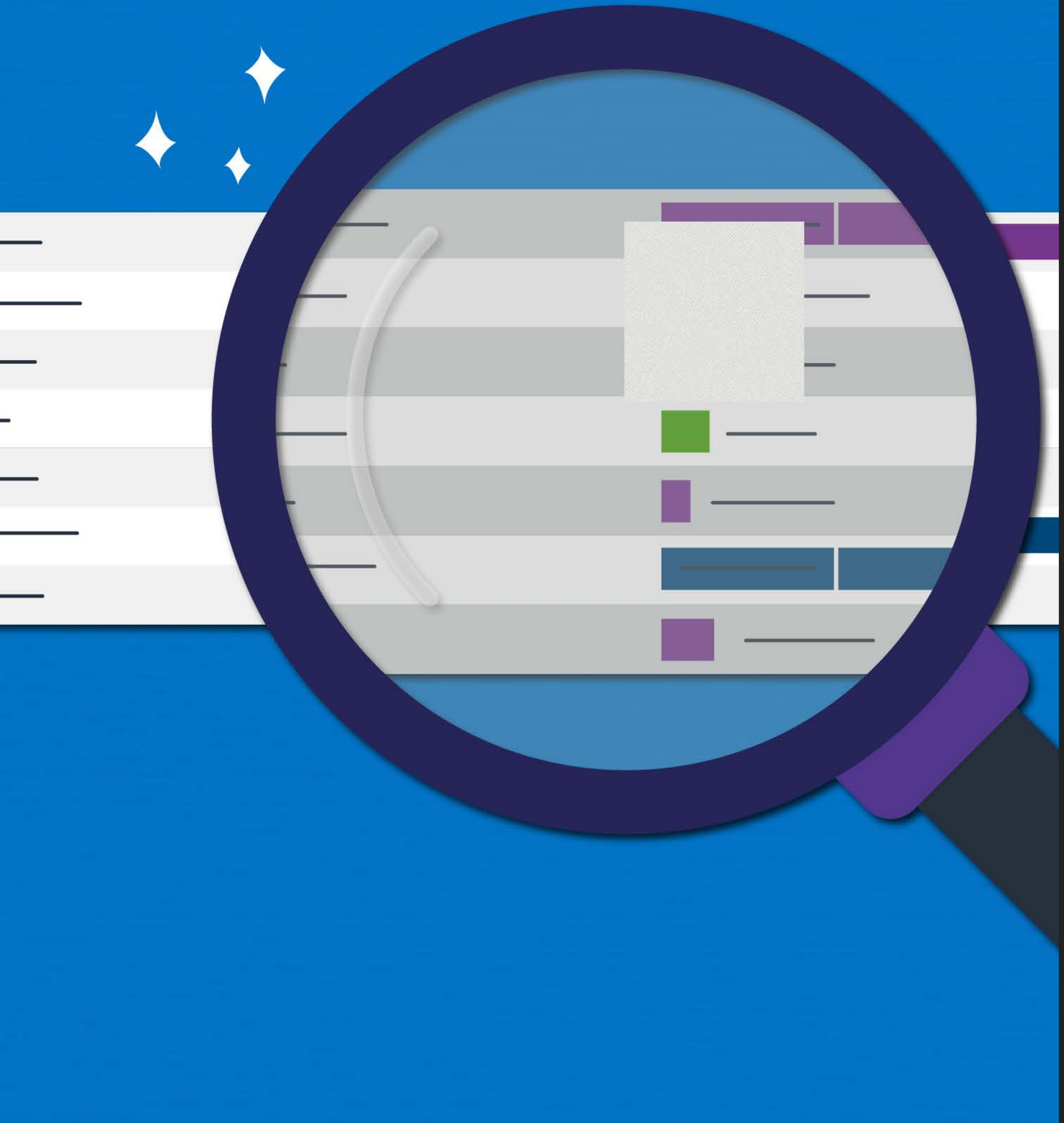
Remember this?

Degradation will happen.

SLOs can help.

SLOs for developing with LLMs

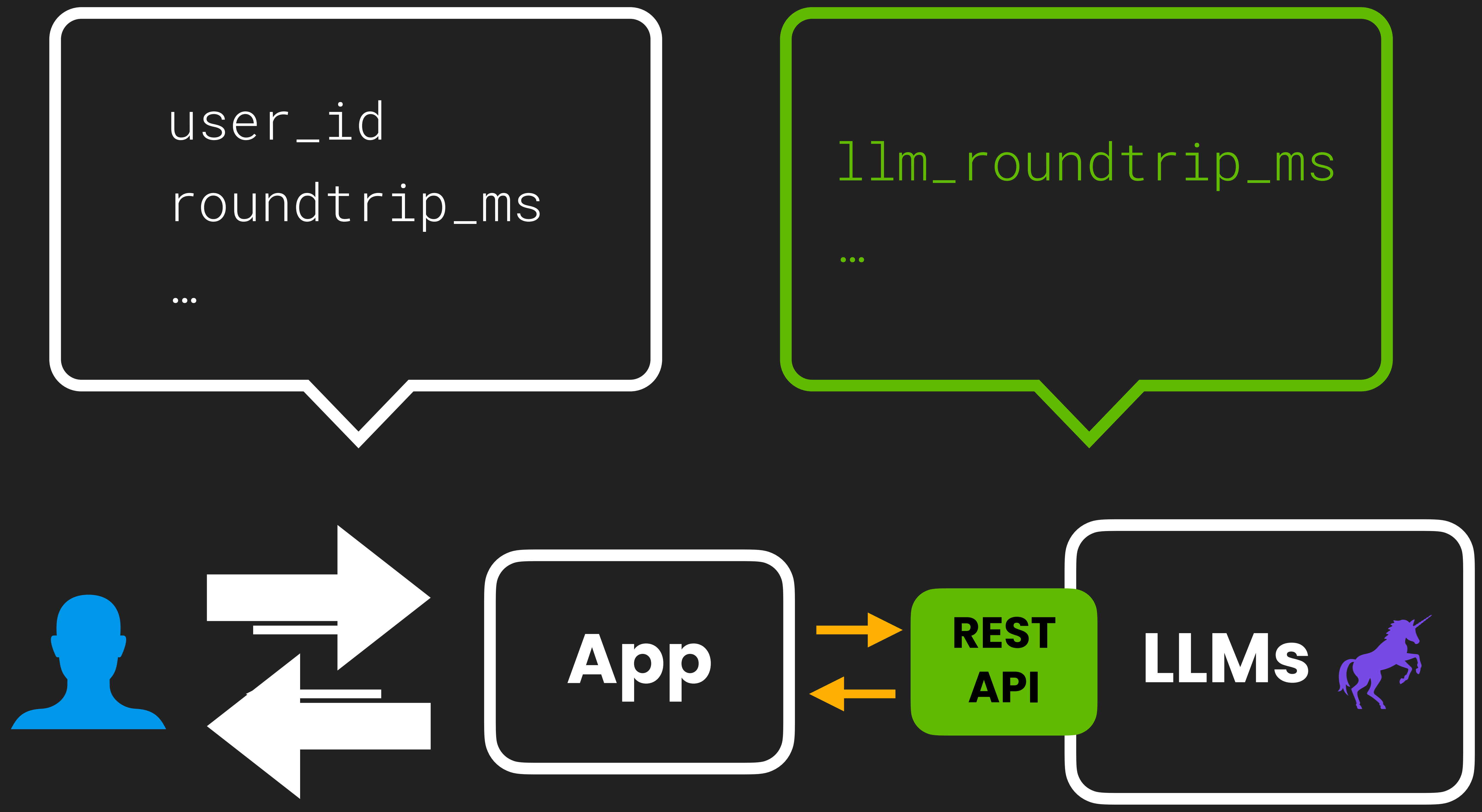




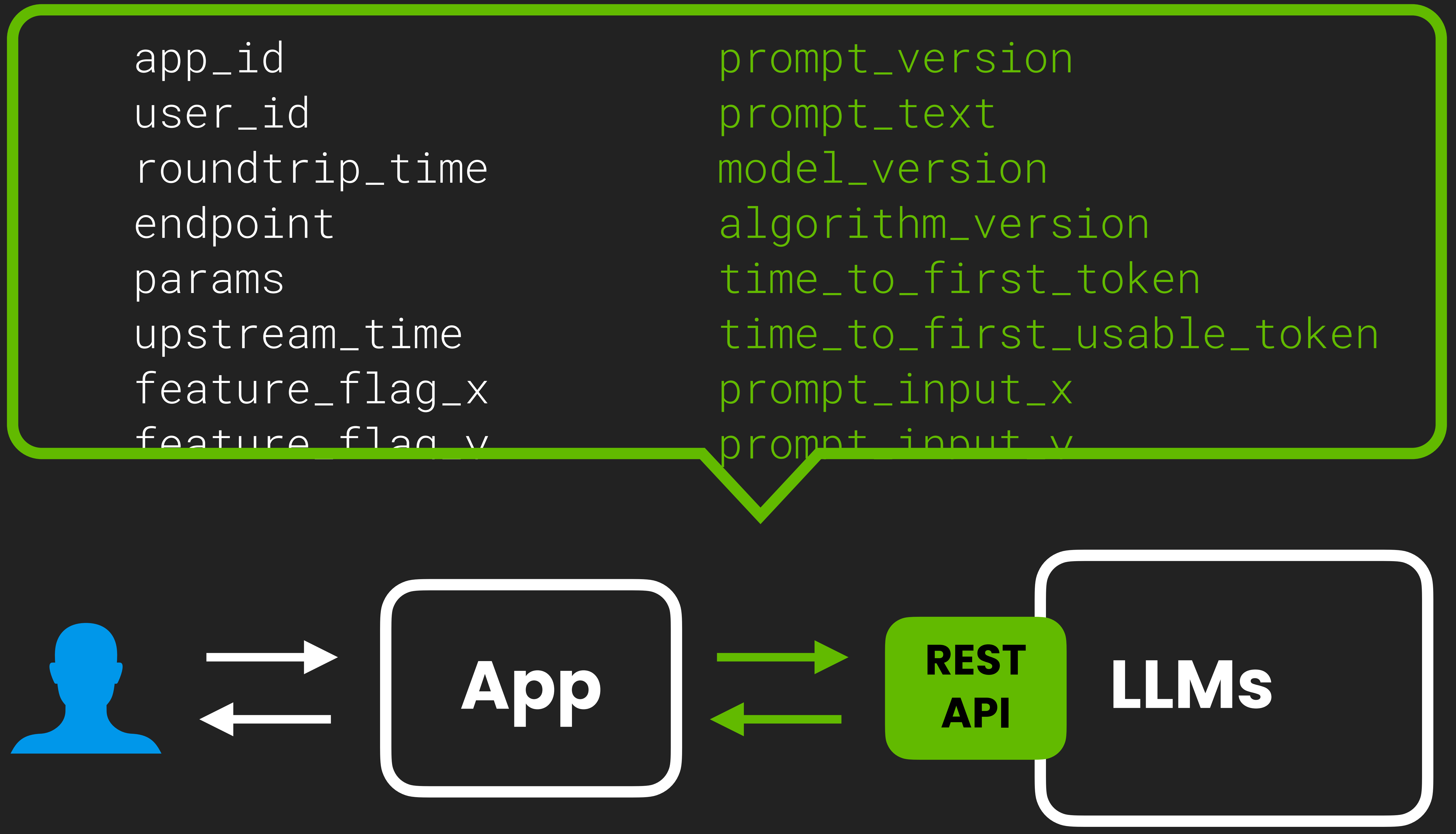
Some more stories

From others in the wild

duolingo



INTERCOM


app_id
user_id
roundtrip_time
endpoint
params
upstream_time
feature_flag_x
prompt_version
prompt_text
model_version
algorithm_version
time_to_first_token
time_to_first_usable_token
prompt_input_x
App
REST API
LLMs

So in the end:

- Incorporating LLMs breaks many of our existing tools for ensuring correctness + a good user experience
- Observability can help! Instrument + observe from the outside in
- Capture all the metadata to be able to debug and analyze unexpected behavior in LLMs
- Embrace the unpredictability of user input + LLMs: run in production and plan to iterate *fast*

More resources:

https://honeycomb.io/blog/hard-stuff-nobody-talks-about-llm

https://honeycomb.io/blog/improving-llms-production-observability

https://honeycomb.io/blog/llms-demand-observability-driven-development

https://honeycomb.io/blog/we-shipped-ai-product

thanks!

q?

@cyen

@honeycombio