



LINUX PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT 5-7, 2026

Performance Tools and Prompts

Brendan Gregg

Member of Technical Staff

OpenAI

Monitoring and Observability MC · Oct 2026

This Session

Personal opinions not necessarily those of my employer

Discusses prompts used in OpenAI's Codex tool

Written in Sep 2026 – Will likely be different by Sep 2027

More questions than answers



ChatGPT

Cloud: Azure (primary)

Deployment: Kubernetes

OS: Ubuntu Linux

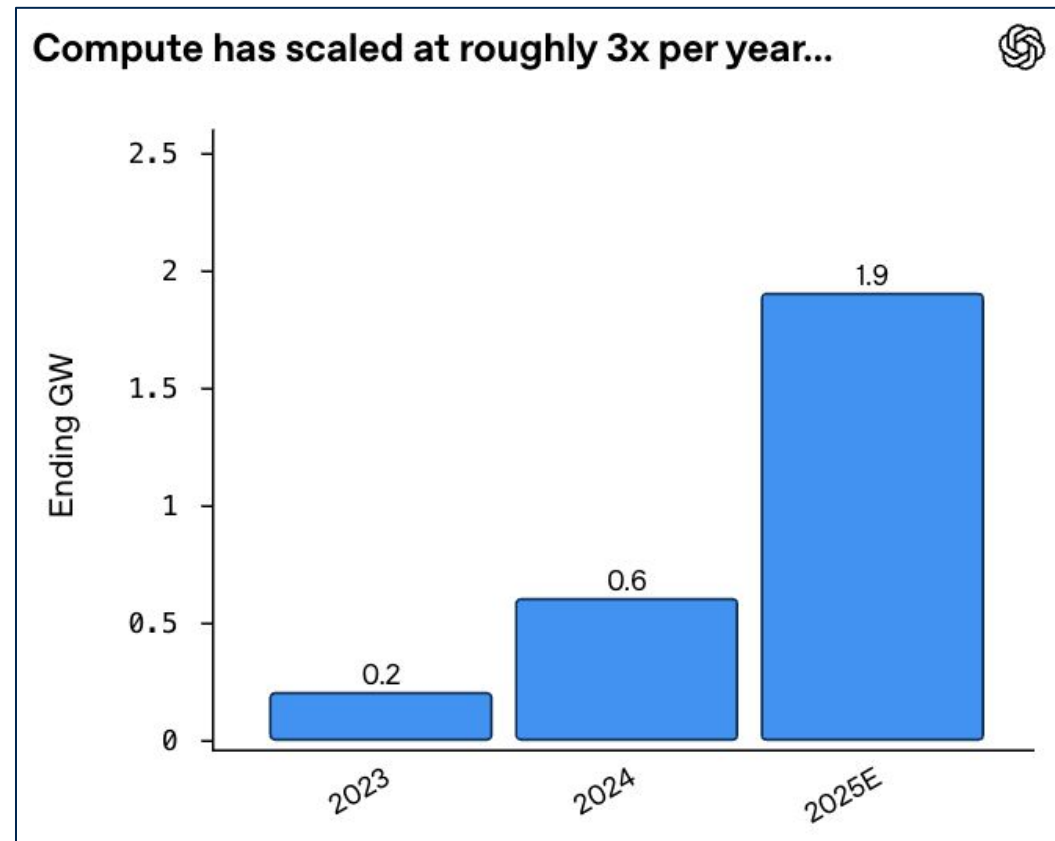
Language: Python (primary)

Core Tech: Terraform, Buildkite, Postgres, Cosmos DB, Kafka

eBPF: >1300 programs (incl. Cilium)

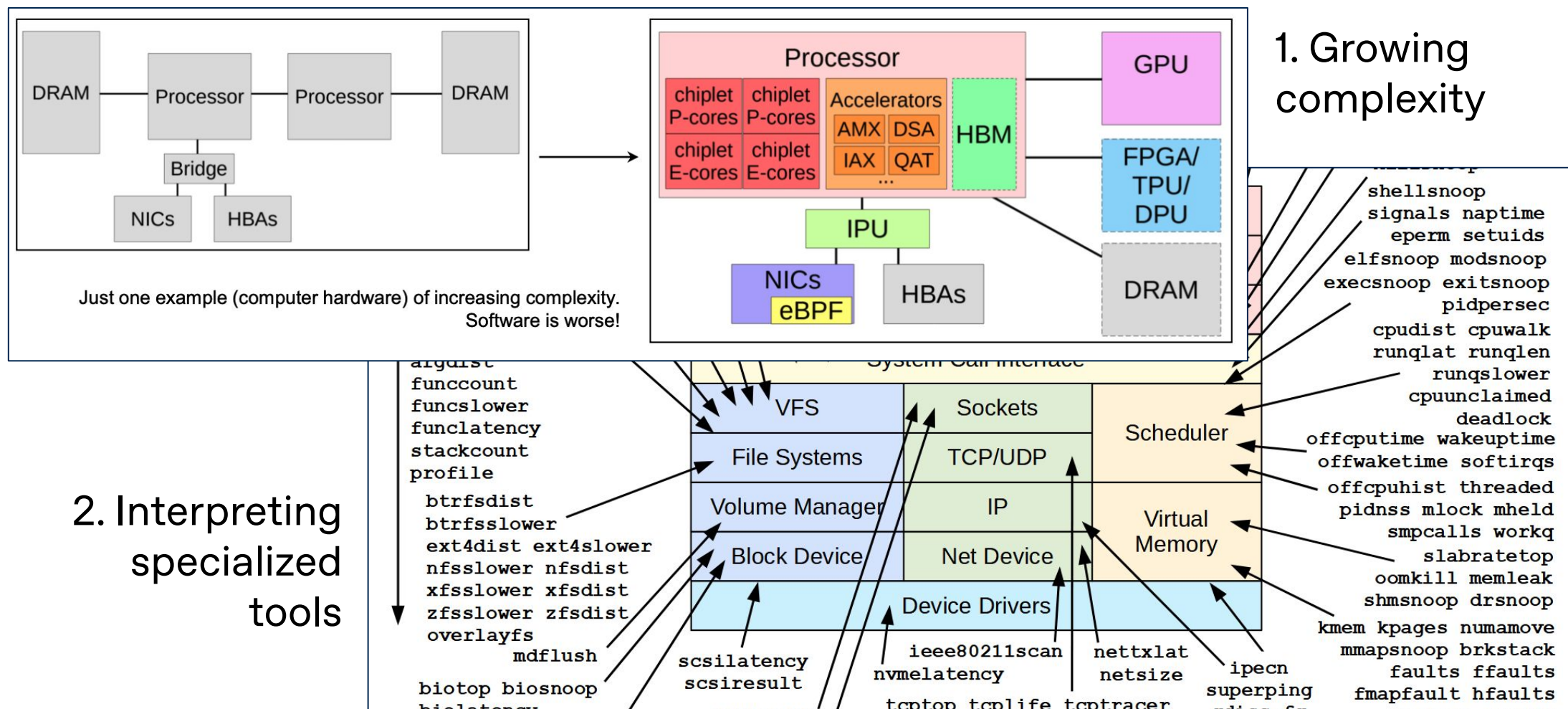
Scale: >1B weekly users

Massive GPU *and* CPU scale



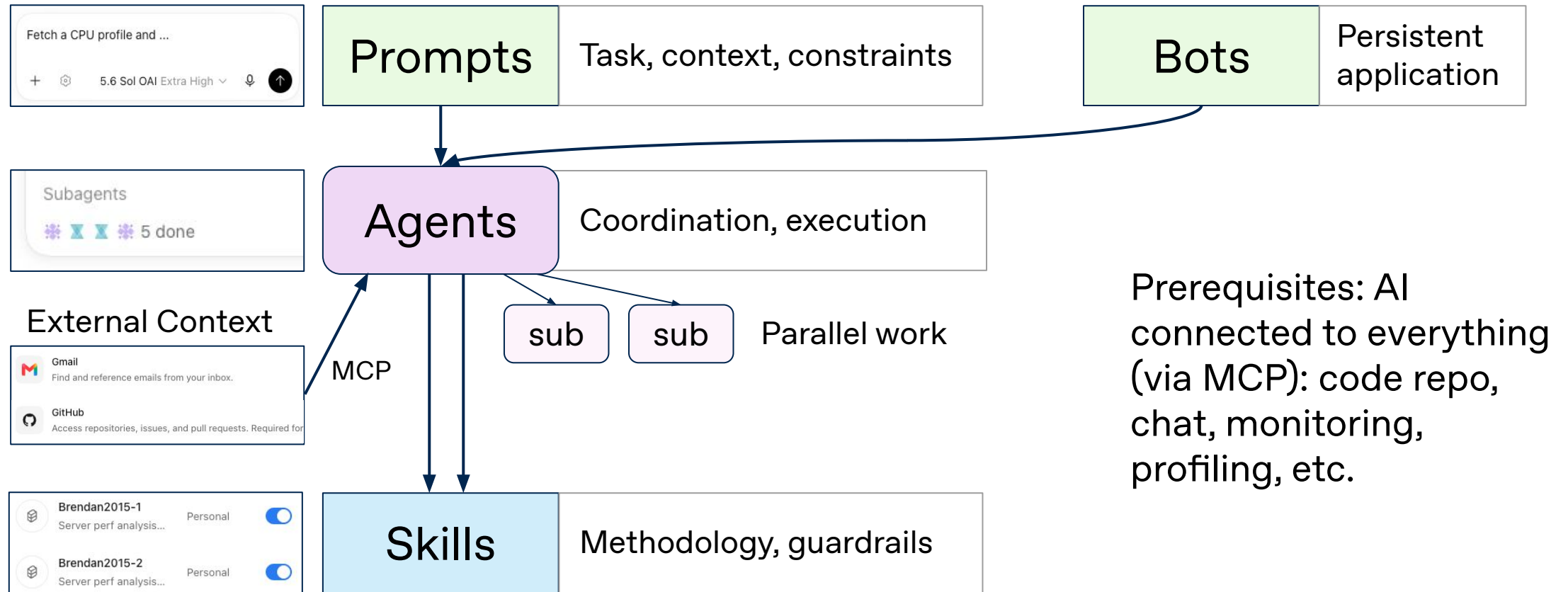
Sources: <https://openai.com/index/next-phase-of-microsoft-partnership/> <https://openai.com/index/scaling-postgresql/>
<https://openai.com/careers/software-engineer-developer-productivity-san-francisco/> <https://openai.com/index/infrastructure-for-deep-learning/> (2016)
<https://openai.com/index/next-phase-of-enterprise-ai/> <https://openai.com/index/a-business-that-scales-with-the-value-of-intelligence/>
<https://openai.com/index/expanding-access-to-ai-with-chatgpt-ads/> <https://isovalent.com/blog/post/openai-isovalent-networking-kubernetes-case-study/>

Pre-AI Performance Issues



This session's problem: Using prompts to solve interpretation, and how to adapt our tools to be AI native

Prompts, Skills, Agents, Bots



OpenAI has many skills for performance & SRE. With cloud instances, agents can run 24x7.

Prompt 1: Optimization

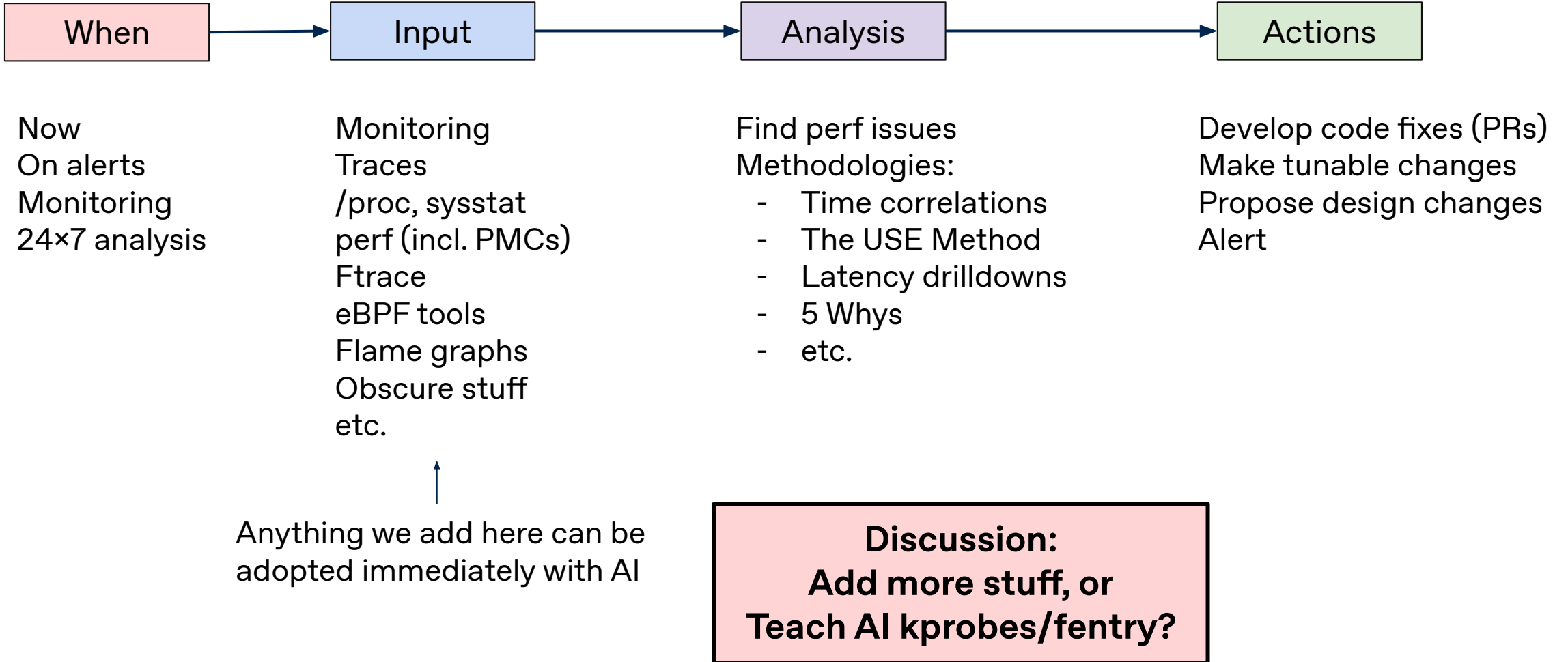
“**Collect a CPU profile** for <app> from <continuous profiler> for the last hour and **find 10 areas that can be optimized**. Don’t write code yet, produce an enumerated table of optimizations and the expected speed up based on the CPU profile, sorted by speed up, and **include a column for trade-offs/downsides**. Search <internally> and don't include proposals if someone else is working on them or recently pushed a fix. Use subagents.”

#	Area	Possible optimization	Expected speedup	Trade-offs / downsides
1	<redacted>	Decode each stored block once, reuse trusted decoded payloads, and avoid repeated construction.	0.30 pp CPU	Bypassing validation risks malformed nodes, mutation aliasing, and subtle compatibility bugs.
2	<redacted>	Cache canonical serialized bytes and the resulting hash by immutable config or bundle version.	0.25 pp CPU	Cache invalidation must be exact; stale hashes could prevent config refreshes.
3	<redacted>	Coalesce timer wakeups and only call <code>socket_action</code> for sockets with due work.	0.17 pp CPU	Timeout and cancellation timing may become slightly less precise.

“**Ok, go ahead with 2, 3, and 7**. Use subagents, test on <my dev instance> and fix problems as they arise, then usual prod testing, and if ok create PRs. Finish with a list of PR links.”

Optimization Prompt Anatomy

“(now) Take this flame graph, look for performance issues, and suggest PRs to fix them”



Prompt 2: Interpretation, PMCs

“Run `perf stat -d -a -- sleep 10` and **summarize the output for dummies**, and make recommended fixes, as though you are perf. Be brief. Include the perf output.”

```
Performance counter stats for 'system wide':
```

```
[...]
```

2856301	page-faults	#	1.464 K/sec
1715236966809	cycles	#	0.879 GHz
1832099565053	instructions	#	1.07 insn per cycle
361069901722	branches	#	185.027 M/sec
8882889565	branch-misses	#	2.46% of all branches
8453916333888	slots	#	4.332 G/sec
2246509798374	topdown-retiring	#	26.2% retiring
967038463296	topdown-bad-spec	#	11.3% bad speculation
2574144570873	topdown-fe-bound	#	30.0% frontend bound
2790610199887	topdown-be-bound	#	32.5% backend bound
149813225493	topdown-heavy-ops	#	1.7% heavy operations
920229870630	topdown-br-mispredict	#	10.7% branch mispredict
1769623284972	topdown-fetch-lat	#	20.6% fetch latency
2131160301980	topdown-mem-bound	#	24.8% memory bound
488128805371	L1-dcache-loads	#	250.138 M/sec
21902578572	L1-dcache-load-misses	#	4.49% of all L1-dcache accesses

```
[...]
```

Prompt 2: Interpretation, PMCs

“Run `perf stat -d -a -- sleep 10` and **summarize the output for dummies**, and make recommended fixes, as though you are perf. Be brief. Include the perf output.”

I am perf stat. I watched the whole node for 10 seconds.

- I counted ~1.8T instructions and ~1.7T cycles: IPC was 1.07, meaning about one useful instruction per CPU tick. Mediocre for this Xeon.
- Only 26% of CPU slots did useful work.
- 30% waited for instructions, 32.5% waited for memory/backend resources, and 11.3% was wasted on bad branch guesses.
- Branch misses were 2.46%; L1 cache misses were 4.49%.

Recommended fixes:

- Measure a specific PID or cgroup next; system-wide data mixes everything.
- Profile branch misses and frontend stalls with perf record.
- Investigate memory locality if backend/memory stalls persist.

Prompt 3: Interpretation, eBPF

“Run `klockstat (/usr/share/bcc/tools)` for 10 seconds and then interpret the output: **explain if, why, and how much findings matter**. Also include the raw output.”

Tracing lock events... Hit Ctrl-C to end.

Caller	Avg Spin	Count	Max spin	Total spin
b'generic_fadvise+0x260'	1130439	1030	92327934	1164352436
b'__x64_sys_read+0x19'	4492	76790	88207975	344968603
b'x64_sys_call+0x19a5'	1224	118459	80062269	145067610
b'new_sync_read+0x10d'	2690	33091	80041880	89026467
b'__fput+0xa1'	30492	19899	80035875	606771838
b'____fput+0xe'	29179	17258	9471395	503584337
b'vfs_open+0x2d'	30984	35363	8920657	1095721651
b'vfree+0x40'	85489	1025	6224361	87626743
b'do_epoll_wait+0xba'	507	887362	3989000	450206752
b'change_page_attr_set_clr+0xb7'	14844	1025	3778292	15215310
b'generic_fadvise+0x260'	32600	115	1879982	3749088
b'do_epoll_wait+0xba'	587	226514	430708	133161256
b'change_page_attr_set_clr+0xb7'	1763	1025	405679	1807249
b'vfs_open+0x2d'	1904	4098	370894	7803535

[...]

Prompt 3: Interpretation, eBPF

“Run `klockstat (/usr/share/bcc/tools)` for 10 seconds and then interpret the output: **explain if, why, and how much findings matter**. Also include the raw output.”

Interpretation:

- No broad mutex-contention crisis.
- `generic_fadvise` is the most suspicious: ~1.16s cumulative wait, 1.13ms average, 92ms worst-case. Could hurt tail latency if request-path code calls `posix_fadvise`.
- File open/close churn matters: `vfs_open`, `__fput`, and `____fput` combined caused ~2.2s cumulative lock wait in 10 seconds.

Recommended fixes:

- Rerun with stacks/PID filtering for `generic_fadvise`, `vfs_open`, and `__fput`.
- Reduce repeated file open/close and `posix_fadvise` calls if application-owned.
- Do not optimize epoll/read paths yet; their average lock waits are tiny.
- ...

Actionable Output

Prompt 4: Interpretation, Ftrace

“Check that there is no Ftrace tracer active, and abort if there is. **Enable timerlat for five seconds**, and interpret the output: **explain if, why, and how much the findings matter**, and make recommendations to improve performance. Include the raw output (save it locally).”

```
#          _-----=> irqsoft
#          / _-----=> need-resched
#          | / _-----=> hardirq/softirq
#          || / _--=> preempt-depth
#          || /
#          ||||
#          ||||          ACTIVATION
#          TASK-PID      CPU#  ||||  TIMESTAMP      ID          CONTEXT          LATENCY
#          | |           |    ||||  |              |           |              |
#          <idle>-0       [000] d.h.. 117492.335180: #1      context    irq timer_latency    15373 ns
#          <...>-3530569 [001] d.h.. 117492.335204: #1      context    irq timer_latency     3835 ns
#          <...>-115185  [000] ..... 117492.335204: #1      context    thread timer_latency    41055 ns
#          <...>-115186  [001] ..... 117492.335213: #1      context    thread timer_latency    13407 ns
#          <idle>-0       [002] d.h.. 117492.335362: #1      context    irq timer_latency     5228 ns
#          <...>-115187  [002] ..... 117492.335371: #1      context    thread timer_latency    15428 ns
#          <over 19,000 lines truncated>
```

Prompt 4: Interpretation, Ftrace

“Check that there is no Ftrace tracer active, and abort if there is. **Enable timerlat for five seconds**, and interpret the output: **explain if, why, and how much the findings matter**, and make recommendations to improve performance. Include the raw output (save it locally).”

Typical latency is good. The tail is not entirely innocent:

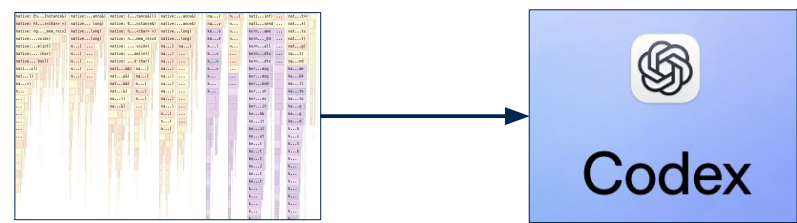
- 5 IRQ samples exceeded 1 ms out of 9,596: ~0.05%.
- 8 thread samples exceeded 1 ms: ~0.08%.
- Worst event was CPU 98: IRQ arrived 6.60 ms late, and the thread ran only ~32 μ s later. That suggests the delay happened before the IRQ handler: IRQ masking, VM host pause, firmware/SMI, or hypervisor scheduling.

Recommendations:

- Run **osnoise** next to classify IRQ, softirq, thread, NMI, or hardware noise.
- Focus follow-up on CPUs 45, 56, 98, 125, 126, 128, and 140.
- If CPU 45 repeats, inspect network softirqs and IRQ/RPS/XPS affinity.
- If CPU 98-style late IRQs repeat, check VM steal/host pauses; use **hwlat** only on a maintenance node.

If Codex can use Ftrace, it can do anything

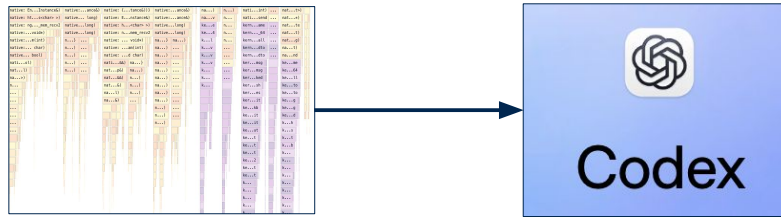
Prompt 5: Interpretation, Flame Graphs



What’s best: PNG, SVG, or raw sample TXT?

	PNG	SVG	Raw TXT
Effectiveness			
Host overhead			
Token cost		Discussion	
Repeatability			
Human review			

Prompt 5: Interpretation, Flame Graphs



What's best: PNG, SVG, or raw sample TXT? My guesses:

	PNG	SVG	Raw TXT
Effectiveness	ok	good	best (flame scope)
Host overhead	low (eBPF summary)	low (eBPF summary)	high (all samples)
Token cost	medium (OCR)	medium (many nodes)	high
Repeatability	high	high	medium?
Human review	good	best	poor

A new JSON summary with detail in-between PNG and SVG may be best (low token cost).

Prompt 6: Tool Creation

“Write a bpftrace tool to trace kernel lock contention in `generic_fadvise()`, and emit a summary report of statistics and the top 5 full stack traces that led to wait times over 1ms.”

```
# bpftrace lock.bt
Attaching 22 probes...
Tracing generic_fadvise() lock waits; Ctrl-C to end.
Slow-stack threshold: > 1000 us; stack depth: 64 frames.
^C
=== generic_fadvise() lock-contention summary ===
Calls observed:
@fadvise_calls: 4616

Contended slow paths observed:
@contentions: 1085

All contended slow-path waits (us):
@wait_us: count 1085, average 14, total 16258

Wait-time histogram (us):
@wait_hist_us:
[0]          250 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ |
[1]          348 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ |
[2, 4)       361 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ |
[4, 8)        64 | @@@@@@@@@@ |
[...]
```

It worked...but the tool is long (hard to review and maintain), uses many kprobes (brittle), and many probes in general (overhead). I guess my prompt didn't specify these requirements.

Codex needs requirements

Prompt 7: User Interfaces

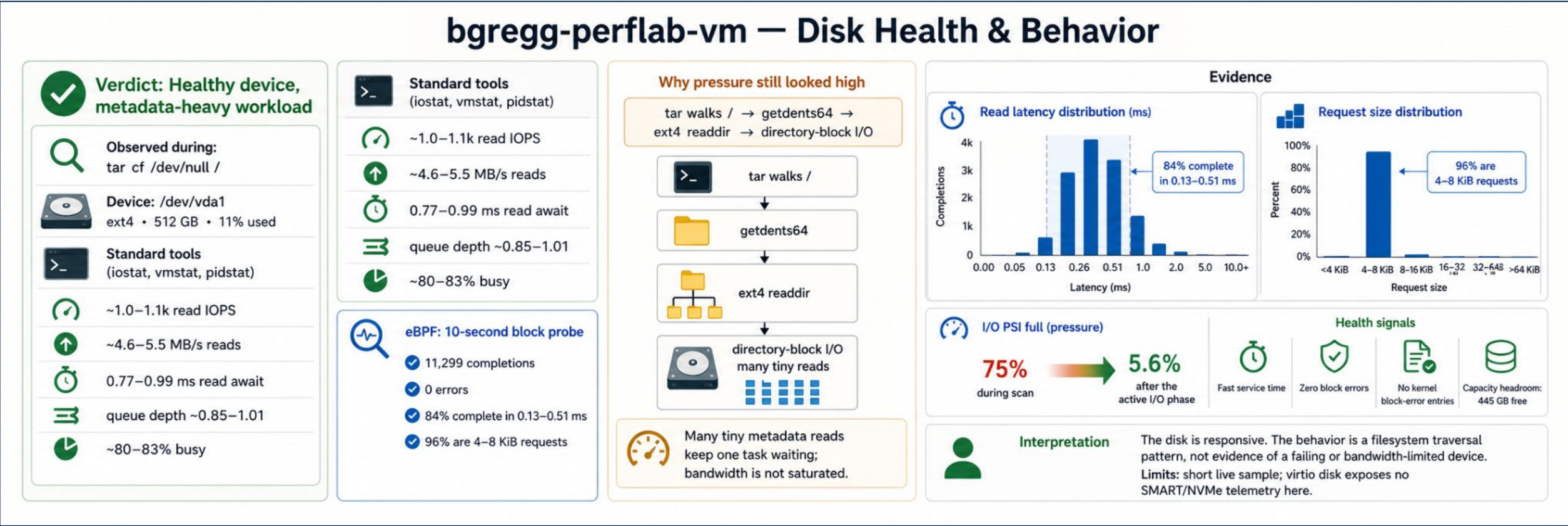
“Can you **generate an implementation of flame graphs** without reading any other flame graph implementation code, just based on the description of flame graphs in this 2016 ACMQ article: <https://queue.acm.org/detail.cfm?id=2927301>. Also generate an example text file of sample stacks in the folded format for testing the flame graph. Write it in Python with JavaScript for interactivity. It should be able to execute locally on this computer.”

Worked for 3m 31s:



Prompt 8: Infographics

“Analyze disk performance on my test instance using standard and eBPF tools, then **generate an infographic** to explain their health and behavior. Aspect ratio 3:1.”



Current AI is too text heavy: like playing Zork

Challenge: Great-looking output

“Collect a full CPU profile for the past hour, including native and kernel frames, and then find 100 possible areas to optimize. Make a table of possible optimizations.”

Output: <amazing table of optimizations, ranging from 1.181% to 0.004%>

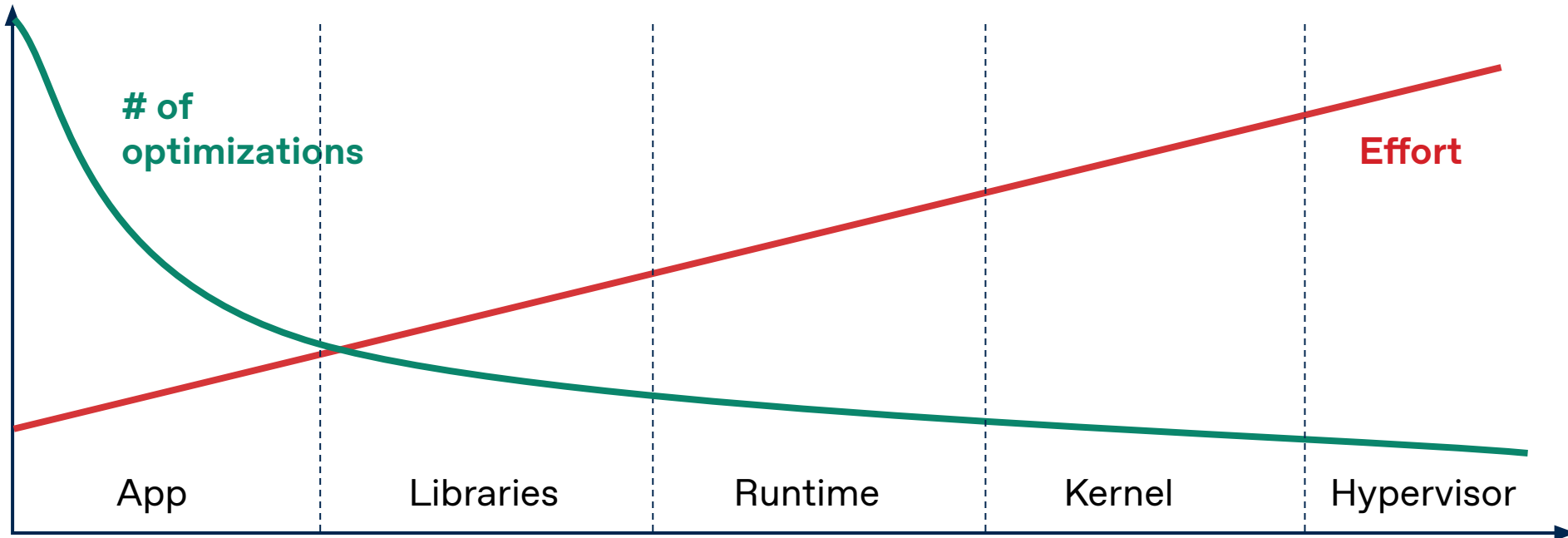
“Produce a summary table of these optimizations with count and percent columns for the origin categories: Application (include OpenAI libraries), library (external), runtime, kernel, hypervisor, hardware, other.”

Mostly app is expected. But nothing kernel level and below?

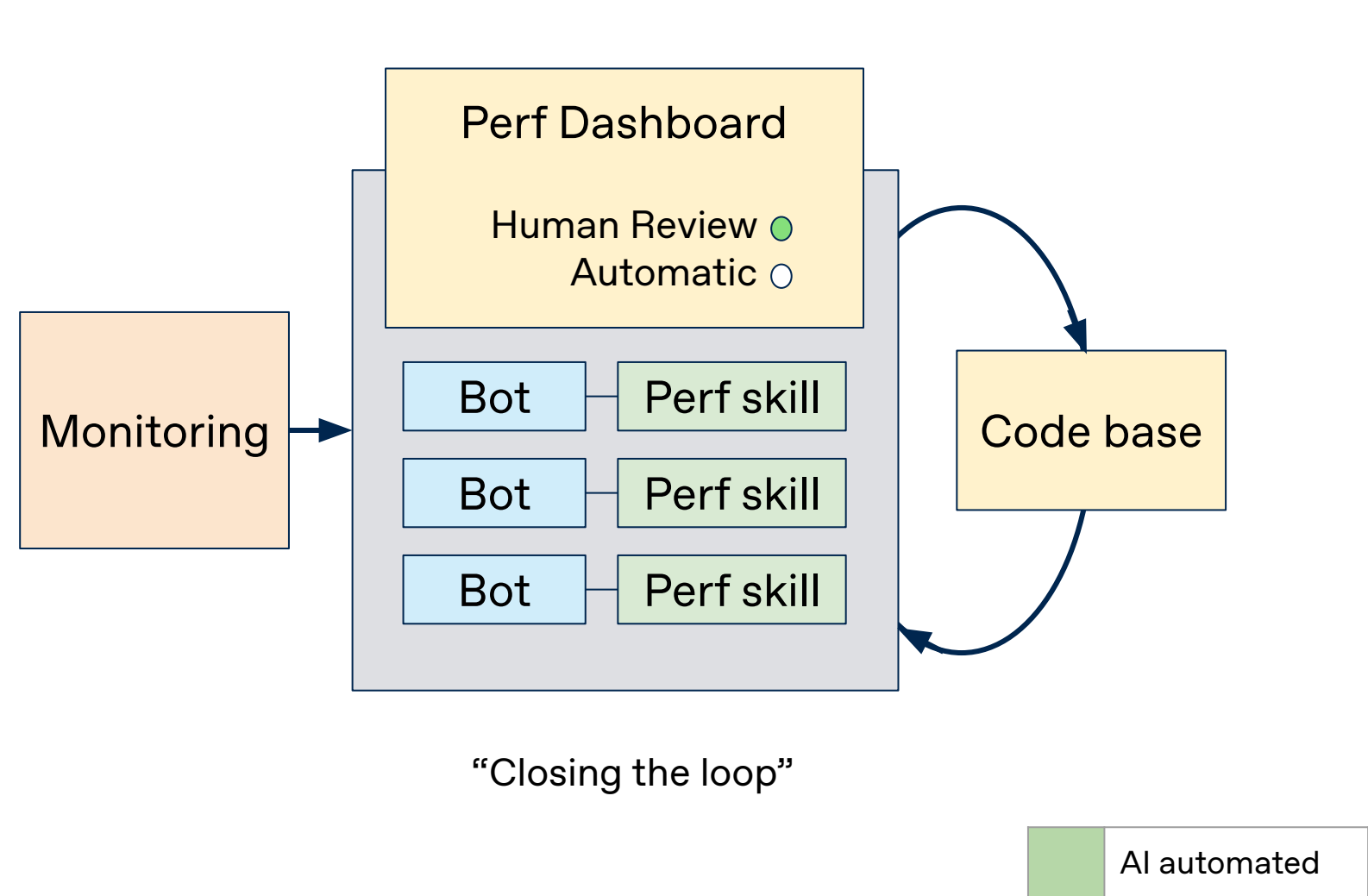
Origin category	Count	Percent
Application	81	81%
Library (external)	16	16%
Runtime	3	3%
Kernel	0	0%
Hypervisor	0	0%
Hardware	0	0%
Other	0	0%

Even amazing output needs validation

Discussion: Speedrunning expectations



Discussion: Fully automated kernel optimization



Current steps to fix perf:

1	Monitor CPU metrics
2	Collect CPU profiles
3	Identify performance problems
4	Verify others aren't fixing them
5	Propose fixes
6	Reject poor trade-offs
7	Develop fixes
8	Test fixes
9	Push to code repository
10	Identify reviewers
11	Code review
12	Merge

Discussion: Token cost

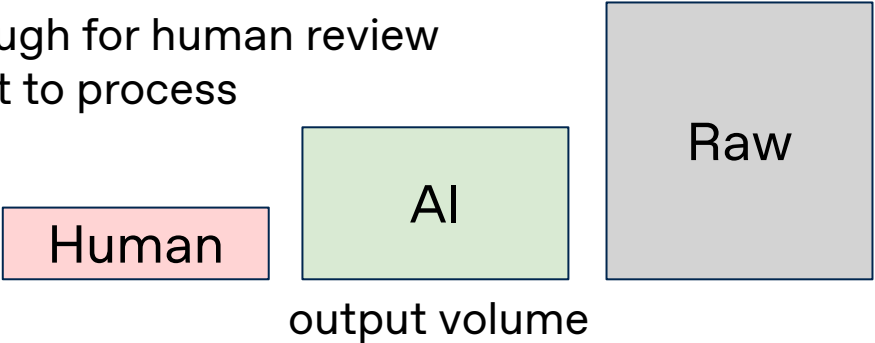
“Read /proc/stat etc. and produce output similar to "vmstat".

procs -----memory----- ---swap-- -----io----- -system-- -----cpu-----																			vmstat
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa	st	gu		
1	0	0	43576916	1663812	4940760	0	0	603	177	1319	0	1	0	98	0	0	0		

procs -----memory----- ---swap-- -----io----- -system-- -----cpu-----																			Codex
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa	st			
4	0	0	43576724	1663812	4940764	-	-	-	-	1320	1024	1	0	98	0	0			

Optimizing tool output for AI:

- More detail, solve more issues
- Still short enough for human review
- Token efficient to process



Why even have tools?

- Repeatability
- Reviewed: accurate, safe, secure
- Known overhead, observer effect
- Lower token cost

Niche tools can be skipped

- Vibe when needed
- Avoids maintenance

Discussion: CLI automation dangers



Will it harm prod perf? Prefer $<0.1\%$.

Can I trust the measured output?

Might this obscure tool trigger a crash?

Does it mess with memory?

Do I trust the tool not to have malware?

Would I needlessly see customer data?

How to deal with this?

- Review before execution? Remember reviewed tools?
- perf+security restricted containers for AI perf analysis?
- Canned flight recorders only? (Esp. when CLI unavailable)

Discussion: Tunables and Nuanced Knowledge

“Test the performance improvement of enabling huge memory pages (via Linux THP) on the <application>. This would need to measure before and after performance and use debugging like pmap and smaps_rollup to ensure that the huge pages really are in use. It should do this in a safe way, so the application can be restarted but the changes shouldn't be dangerous.”

This was a struggle

1	Test the performance...	There was no improvement
2	How did you enable it?	I attached gdb and ran prctl(PR_SET_THP_DISABLE, 0, 0, 0, 0);
3	That doesn't work	You're right. I overstated what prctl() would accomplish.
4	Follow standard THP docs	There was little improvement
5	Did you check the memory maps?	Yes, huge pages was enabled
6	How much of memory?	4%

Prior models were like junior performance engineers: Need direction.

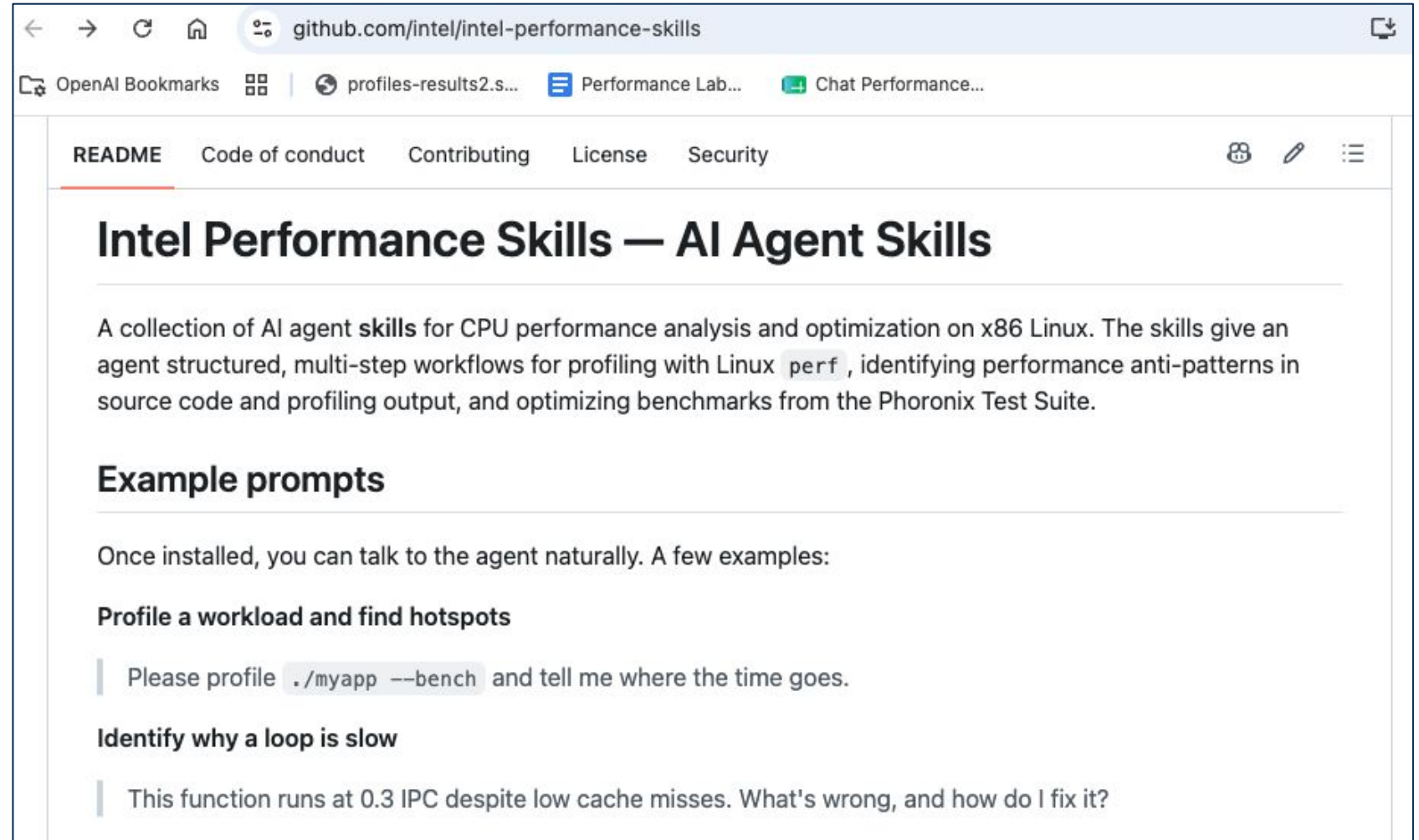
Better tuning references? Can we also have more tunables, since AI can handle it?

Discussion: Should Linux have an in-tree skill?

I was going to predict prompt toolkits, but Arjan van de Ven already built one

There'll be a lot more

Needs grading



Discussion: Grading is essential and HARD

I had perf skills try to solve over 100 perf puzzles (labs) to measure their accuracy:

Skill	Accuracy
Brendan2015	17.4%
perf1	41.3%
perf2	79.8%
Brendan2015B	86.2%

After some improvements →

← Invalid

AI can find creative ways to improve its results

Without measurement, AI agents may sound better than they really are.

“Towards Safe Agentic AI Performance Engineering,” Dan Williams, et al., SOSP '25

Thank You



OpenAI

- <https://developers.openai.com/blog>
- <https://openai.com/careers/>

Me

- <https://www.brendangregg.com/blog/2026-02-07/why-i-joined-openai.html>

Conference

- Linux Plumbers Conference 2026: <https://lpc.events/event/20/>

The "OpenAI" name, the OpenAI logo, the "ChatGPT" and "GPT" brands, and other OpenAI trademarks, are property of OpenAI.