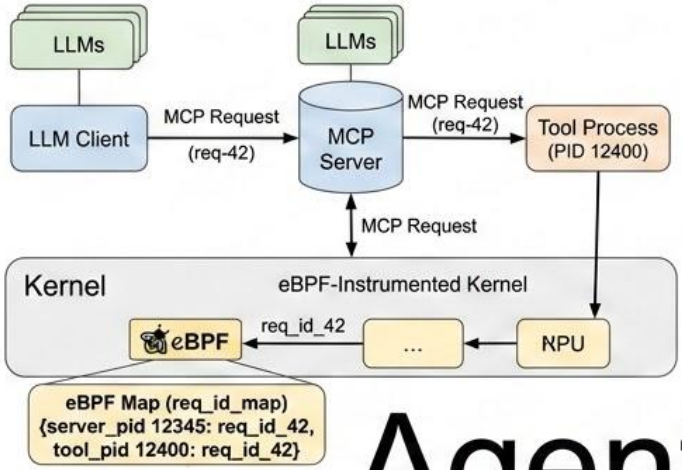
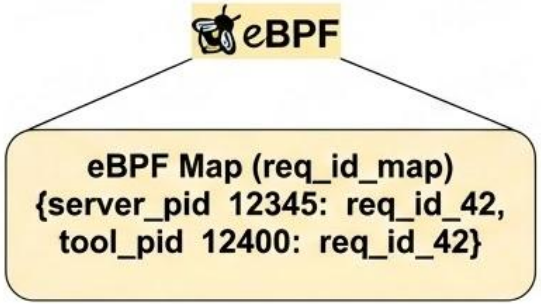
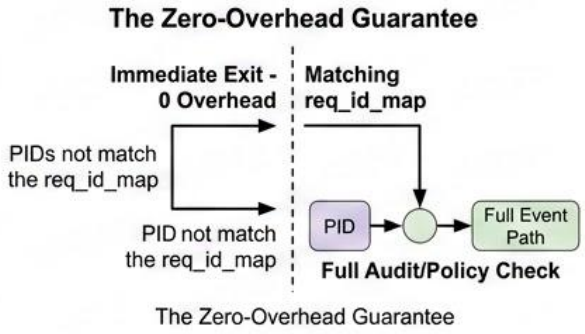


Agentless Agents

Replacing Sidecar Observability With eBPF + MCP Tool Chains



By: Harini Anand



$whoami

- **SDE at IBM** under the **Data & AI** Division
- **Software Engineering Research** at **UIUC**
- **Computational Cognition Research** at **Georgia Institute of Technology**
- **Biomedical XAI Research** at **Dartmouth College**
- **Google KaggleX Mentee, AWS Scholar, Harvard WE Tech Fellow, and Cohere, University of Oxford & MIT** Summer School alumna
- **Stanford** High Impact WiDS Ambassador

The Sidecar Tax

Every observability sidecar you deploy is:

+1 container to patch

+1 process to crash

+1 auth surface to secure

+1 resource consumer per pod

+1 thing that can silently stop reporting

+1 configuration file to keep synchronized

At 200 pods, that's 200 observability agents observing 200 services.

Who observes the observers?

Current state of the art:

Service Pod

├── your-app (256MB)

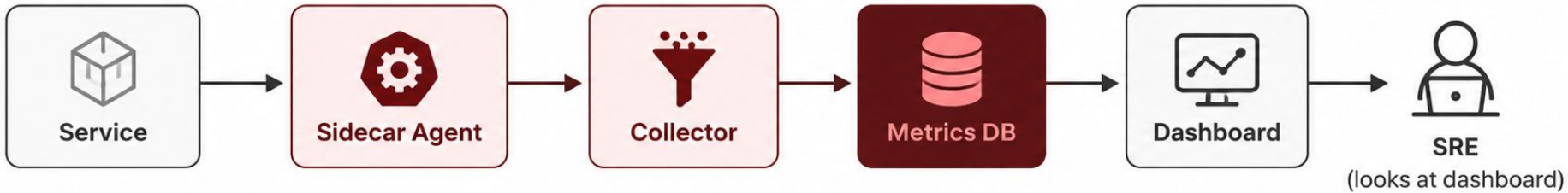
├── prometheus-exporter sidecar (48MB)

├── otel-collector sidecar (96MB)

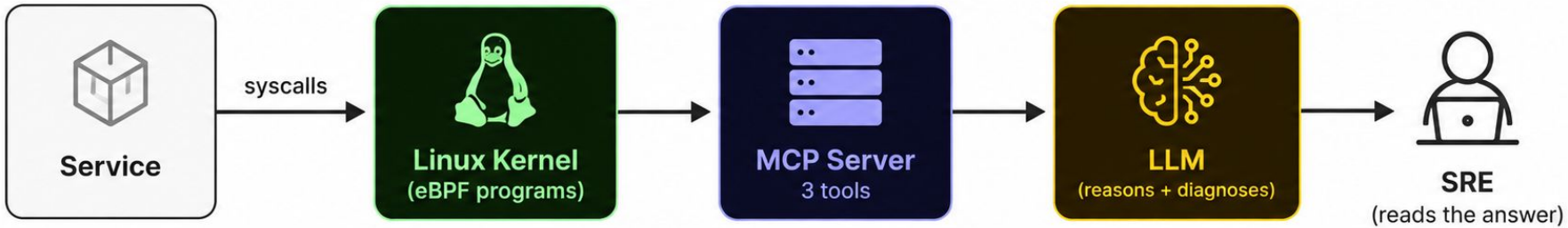
└── log-shipper sidecar (64MB) ← 40% overhead, all before serving a request

The Flip

Instead of agents that push data to dashboards:

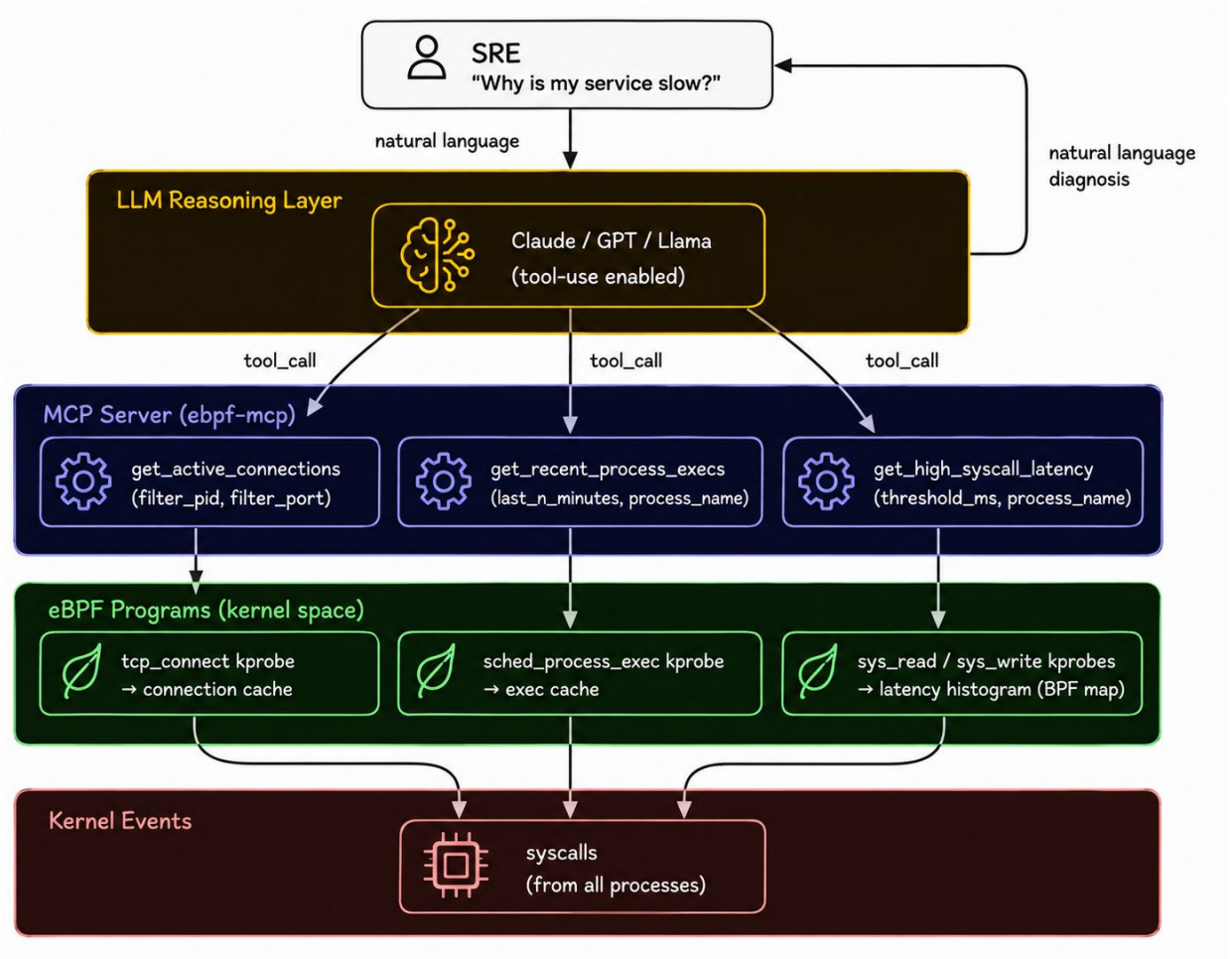


Use eBPF to expose kernel truth as queryable MCP tools
let the LLM pull what it needs.


Service
syscalls
Linux Kernel
(eBPF programs)
MCP Server
3 tools
LLM
(reasons + diagnoses)
SRE
(reads the answer)

Zero sidecars. Zero exporters. One interface. The LLM asks; the kernel answers.

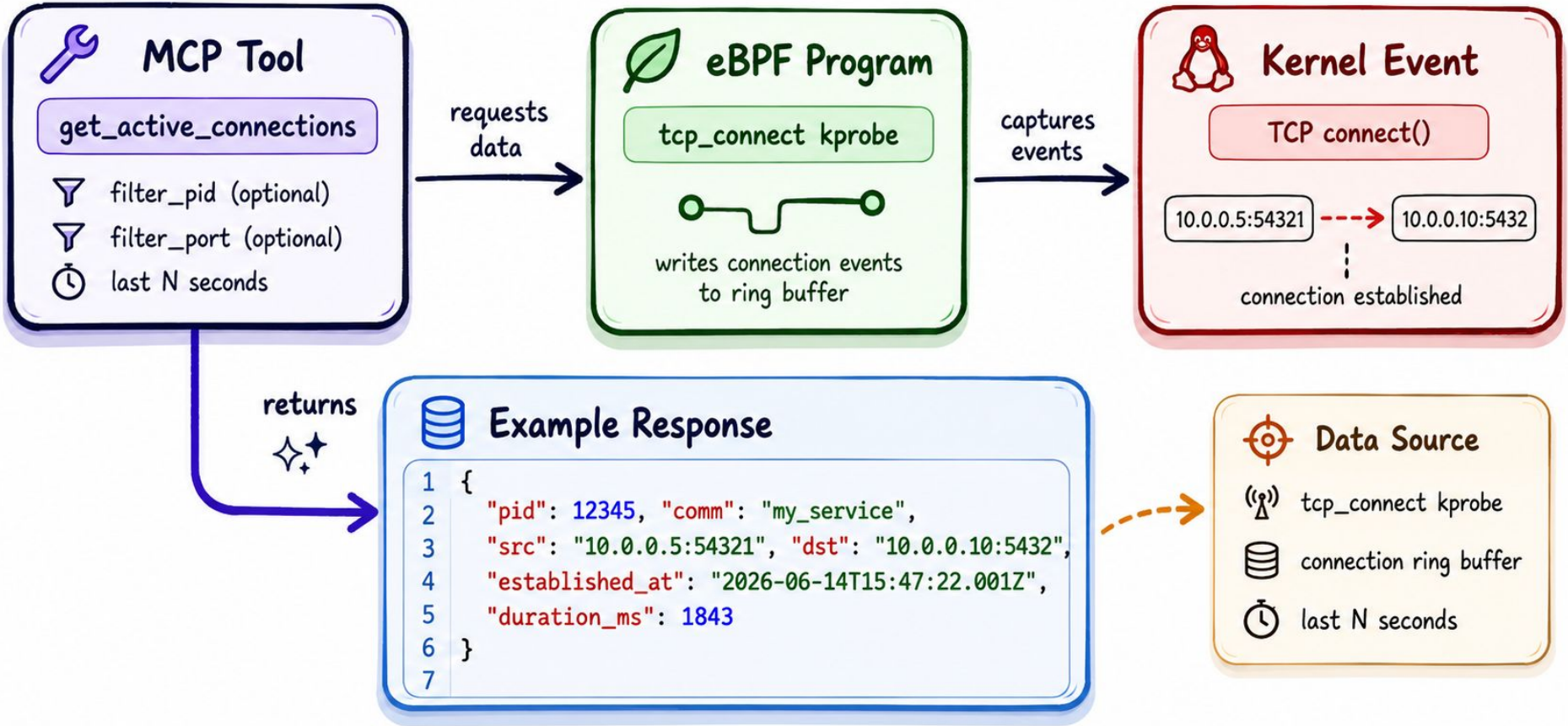
System Architecture



The Three Tools: What They Each See

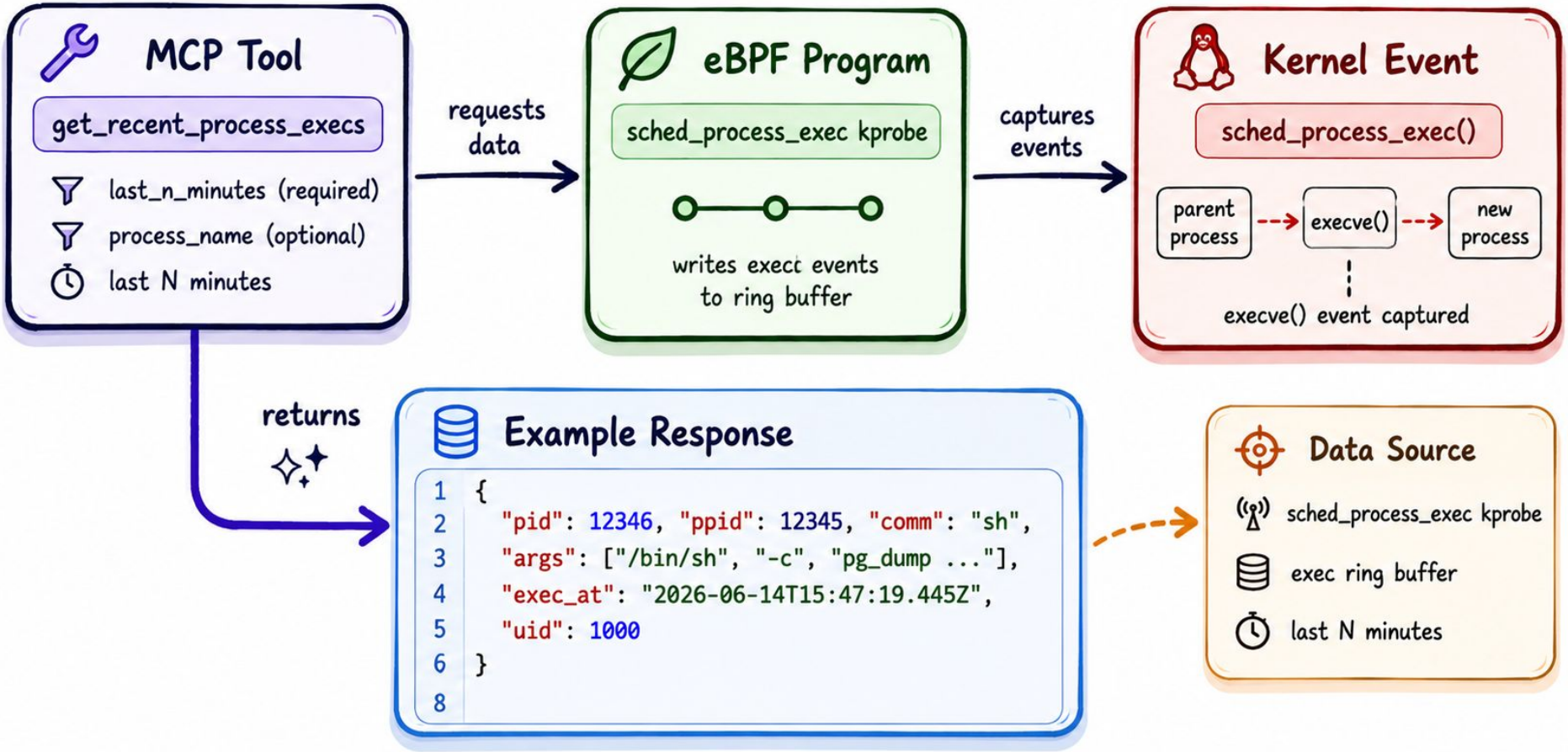
Tool 1: get_active_connections

Returns connections established in the last N seconds



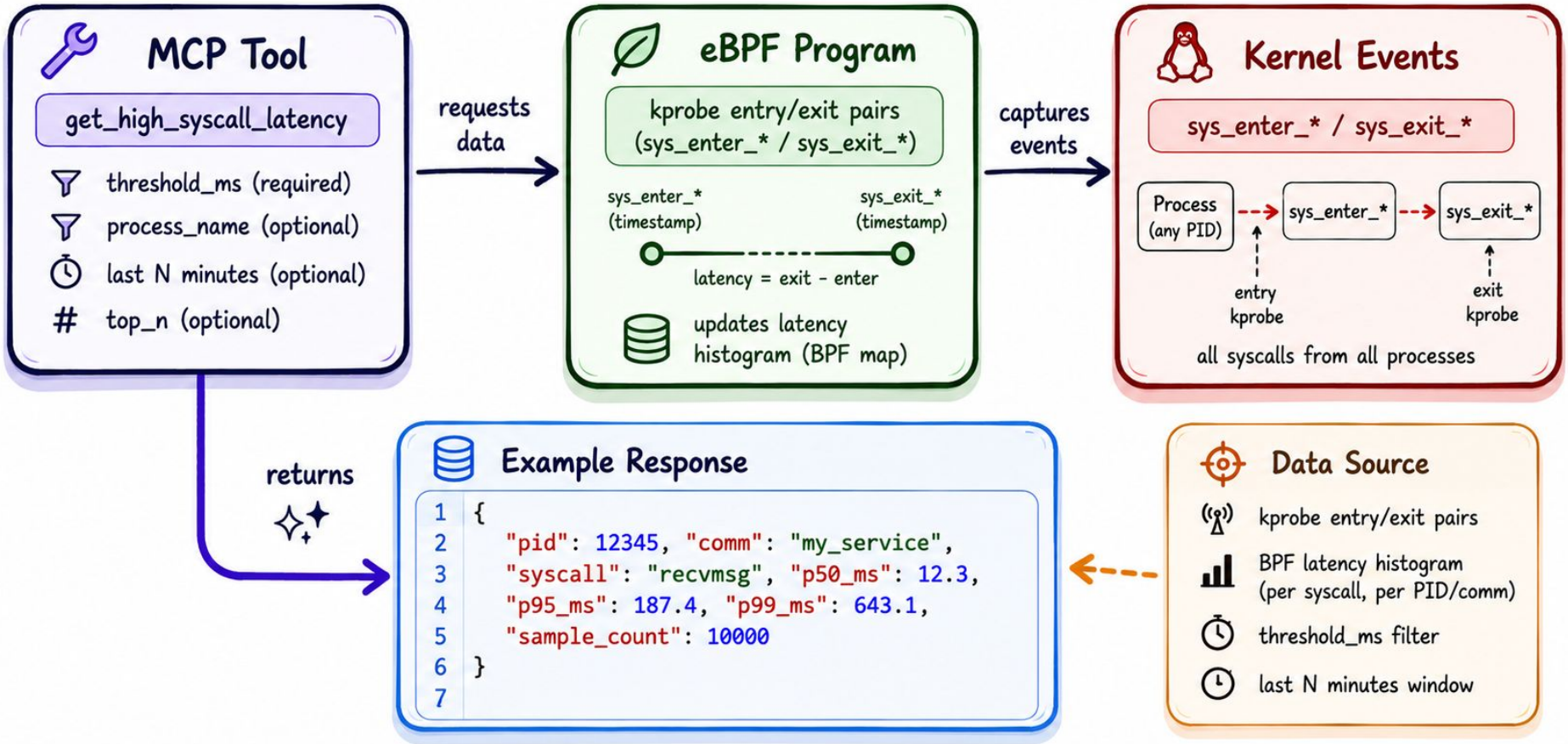
Tool 2: get_recent_process_execs

Returns execve() events from the last N minutes

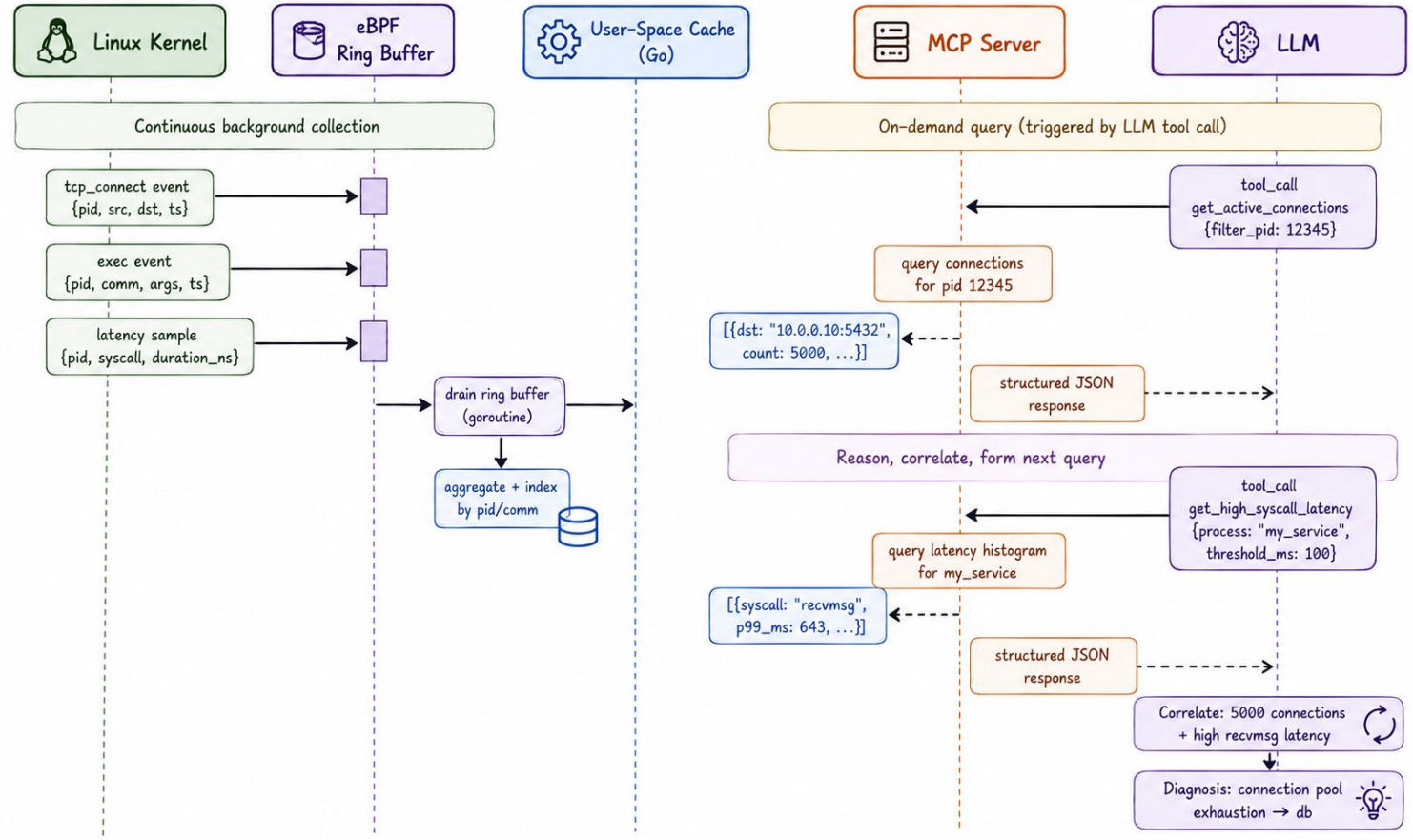


Tool 3: get_high_syscall_latency

Returns processes with syscall latency > threshold_ms



Data Flow: Kernel to LLM



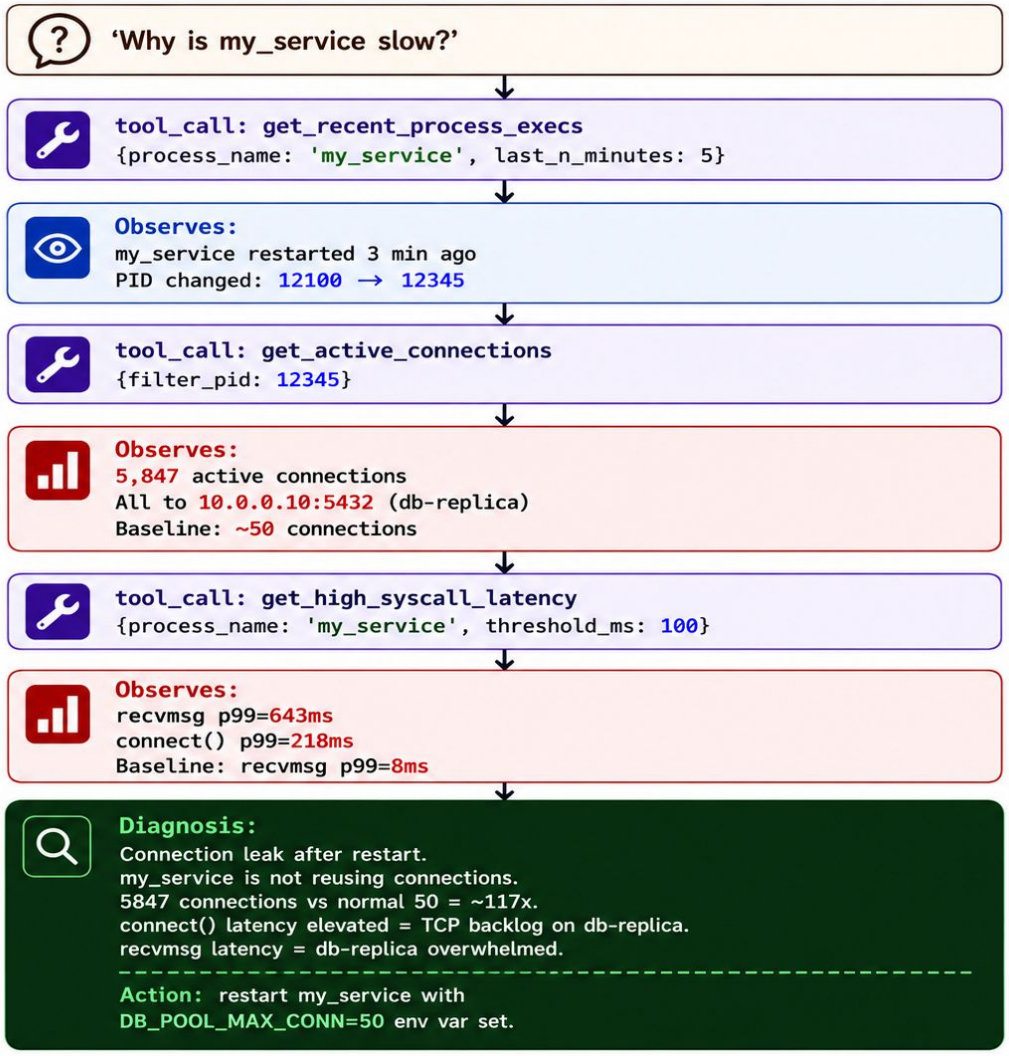
Demo: "Why Is My Service Slow?"

Setup: `my_service` is experiencing degraded response times. No alerts fired. SRE notices latency creeping in dashboards.

The query:

SRE → LLM: "Why is my_service slow right now?"

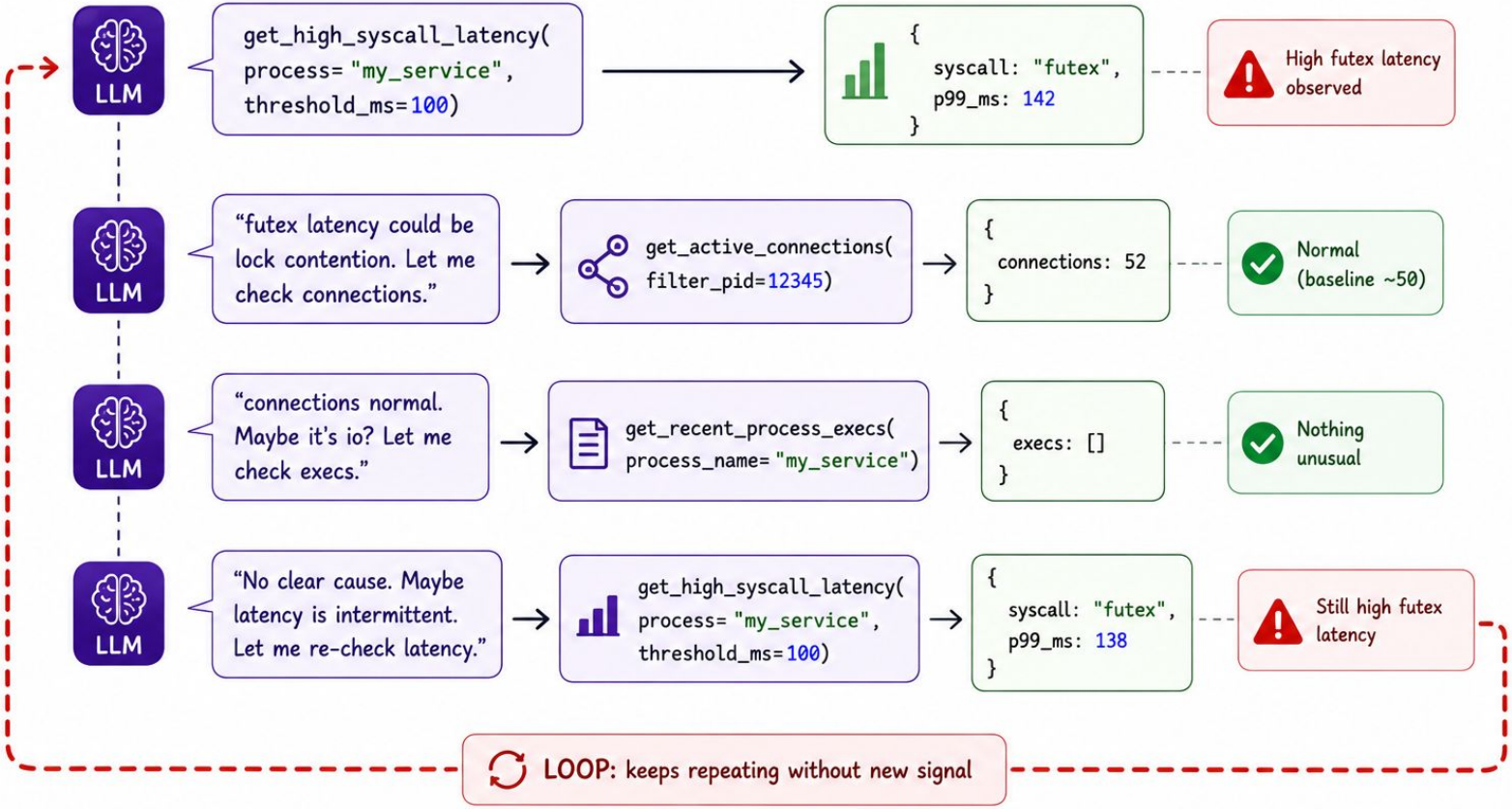
What the LLM actually does (multi-tool chain):


'Why is my_service slow?'
tool_call: get_recent_process_execs
{process_name: 'my_service', last_n_minutes: 5}
Observes:
my_service restarted 3 min ago
PID changed: 12100 → 12345
tool_call: get_active_connections
{filter_pid: 12345}
Observes:
5,847 active connections
All to 10.0.0.10:5432 (db-replica)
Baseline: ~50 connections
tool_call: get_high_syscall_latency
{process_name: 'my_service', threshold_ms: 100}
Observes:
recvmsg p99=643ms
connect() p99=218ms
Baseline: recvmsg p99=8ms
Diagnosis:
Connection leak after restart.
my_service is not reusing connections.
5847 connections vs normal 50 = ~117x.
connect() latency elevated = TCP backlog on db-replica.
recvmsg latency = db-replica overwhelmed.
Action: restart my_service with
DB_POOL_MAX_CONN=50 env var set.

Failure Mode 1: Tool-Call Loops

What happens: LLM receives ambiguous latency signal → calls more tools → gets more data → still ambiguous → calls more tools...

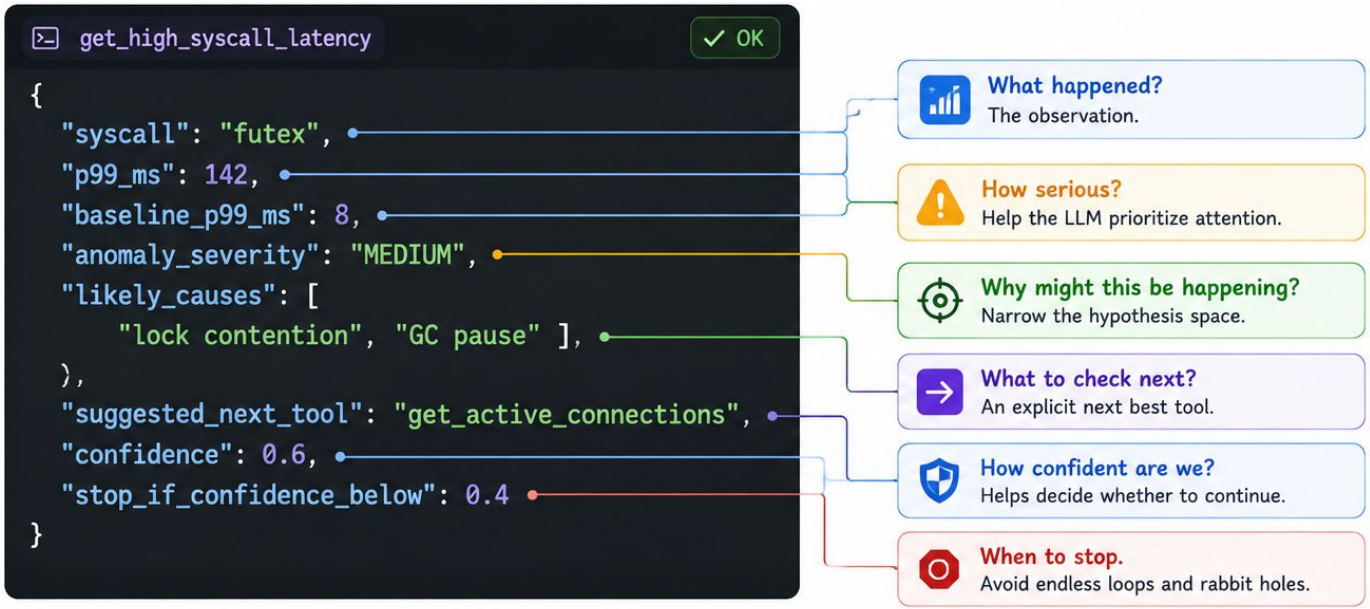
The Loop Pattern (Bad)



The fix: constrain with `**max_tool_calls**` + structured `**confidence**` in responses

✨ Tool Response That Guides, Not Just Reports

Enrich tool responses with guidance so the LLM knows what to do next (or when to stop).



How the LLM should use this

1. **Check severity**
   If LOW → consider stopping.
2. **Consider likely causes**
   Focus on the top 1–2 causes.
3. **Use suggested tool**
   Try suggested_next_tool unless it doesn't fit.
4. **Respect stop threshold**
   If confidence < stop_if_confidence_below → STOP and report.

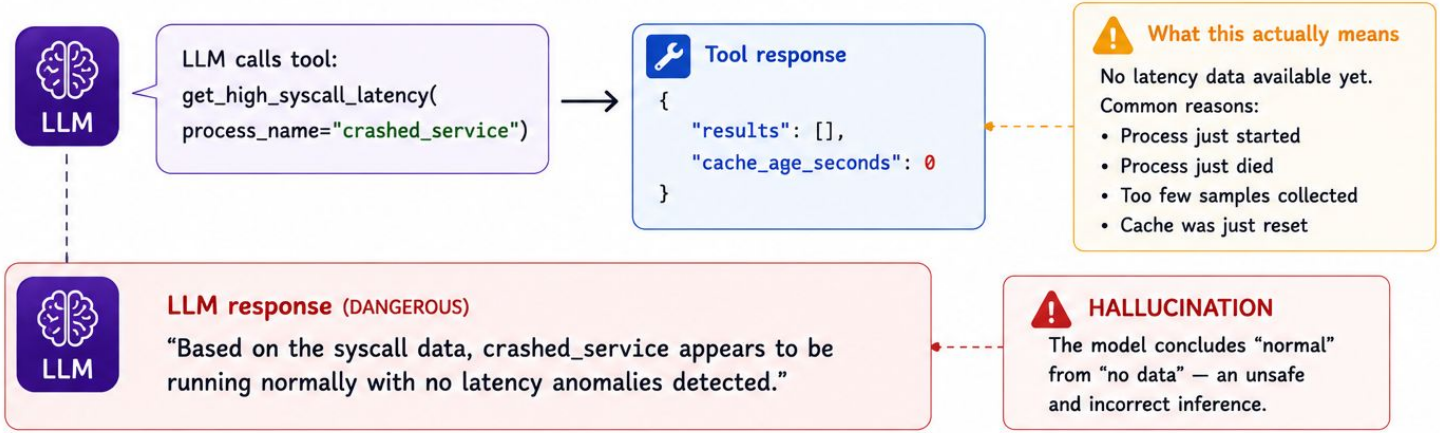
Key benefit: The tool response doesn't just report data — it guides the LLM toward efficient, high-signal next steps.

Failure Mode 2: Hallucination With Sparse eBPF Data

What happens: BPF ring buffer is behind or drained → LLM receives empty/stale data → fabricates a diagnosis

Dangerous Pattern

Empty data ≠ Normal. It might mean: no data, not no problem.



Safer Behavior for the LLM

1. Check metadata first

```
{
  "results": [],
  "cache_age_seconds": 0
}
```

→ No data available.

2. Reason about state

The process may have just died or there's insufficient data. Cannot assess latency.

3. Take safe next step

- Check if the process is still running
- Look at recent exits / crashes
- Check logs or core dump
- Re-check latency after some time

Correct response example

"I don't have syscall latency data for crashed_service yet (cache_age_seconds=0). The process may have just died or there's insufficient data. Let me check recent process execs and exits to understand what happened."

The fix: explicit data freshness + confidence bounds in every response

Honest Response Schema

Be transparent about data quality. Help the LLM (and user) make correct inferences.

```
get_high_syscall_latency(process_name="crashed_service")
```

```
{
  "results": [],
  "cache_age_seconds": 0,
  "data_quality": {
    "status": "INSUFFICIENT",
    "reason": "Process PID not found in last 30s
               of eBPF eventes",
    "process_last_seen_at": null,
    "recommended_action": "REPORT_UNCERTAINTY_TO_USER"
  }
}
```

results
Empty because no latency data was found.

cache_age_seconds
How fresh the data is. 0s = very fresh.

data_quality.status
INSUFFICIENT = not enough data to make a reliable assessment.

reason
Explains why the data is insufficient.

process_last_seen_at
When we last saw this process in events. null = never seen or not seen recently.

recommended_action
Tells the LLM what to do next.

How the LLM should interpret this

1. Check data quality first
Always read data_quality.status before drawing conclusions.

→

2. Respect uncertainty
If INSUFFICIENT or STALE, do not claim "normal" or "no issues".

→

3. Take recommended action
Follow recommended_action. Ask for more time, check other tools, or report uncertainty.

→

4. Communicate honestly
Tell the user what is known, what is unknown, and what you will do next.

Example honest LLM response

"I don't have syscall latency data for crashed_service right now (process not seen in the last 30s of eBPF events). The process may have just exited or there may be insufficient data. Let me check recent process exits and crashes."

Rule:
Never let an MCP tool return `{"results": []}` without a `data_quality` explanation.

Failure Mode 3: Kernel Probe Detachment Under Load

What happens: Under heavy load, the eBPF ring buffer fills faster than the consumer reads it. The kernel starts dropping events. The consumer reports partial data.

Scenario: 50,000 TCP connects/sec during a DDoS or traffic surge

Ring buffer size: 1MB (default)

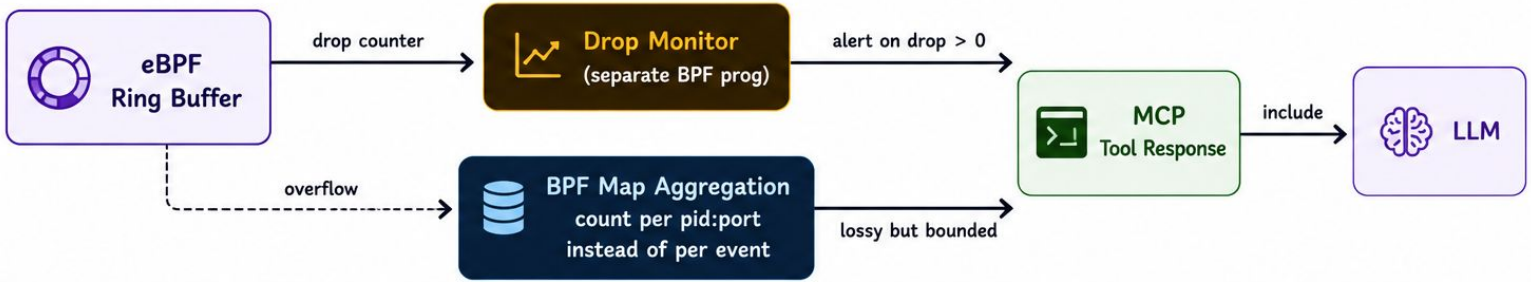
Consumer drain rate: ~40,000 events/sec

Result: ring buffer drop counter increments

LLM sees: "only 5,000 connections" (reality: 50,000)

LLM diagnosis: "moderate traffic, probably fine"

Handle Data Loss Proactively



Key principle: Never hide data loss. Surface it, quantify it, and guide the LLM to reason with uncertainty.

Always include drop stats in MCP response

```
{
  "connections": [...],
  "ebpf_health": {
    "ring_buffer_drops": 1247,
    "data_completeness_pct": 87.3,
    "under_load": true,
    "llm_note": "Data may undercount. Treat all counts as lower bounds."
  }
}
```

ring_buffer_drops
Non-zero = some events were lost.
Higher = less trustworthy.

data_completeness_pct
Estimated % of data captured.
Use to reason about confidence.

under_load
True means the system is under pressure.
Expect more loss.

llm_note
Direct guidance for the LLM.
Encourages honest, bounded conclusions.

What the LLM should do
Acknowledge potential undercount. Treat counts as lower bounds. Ask for aggregation or reduced time window if needed.

Designing MCP Tool Responses That Constrain LLM Reasoning

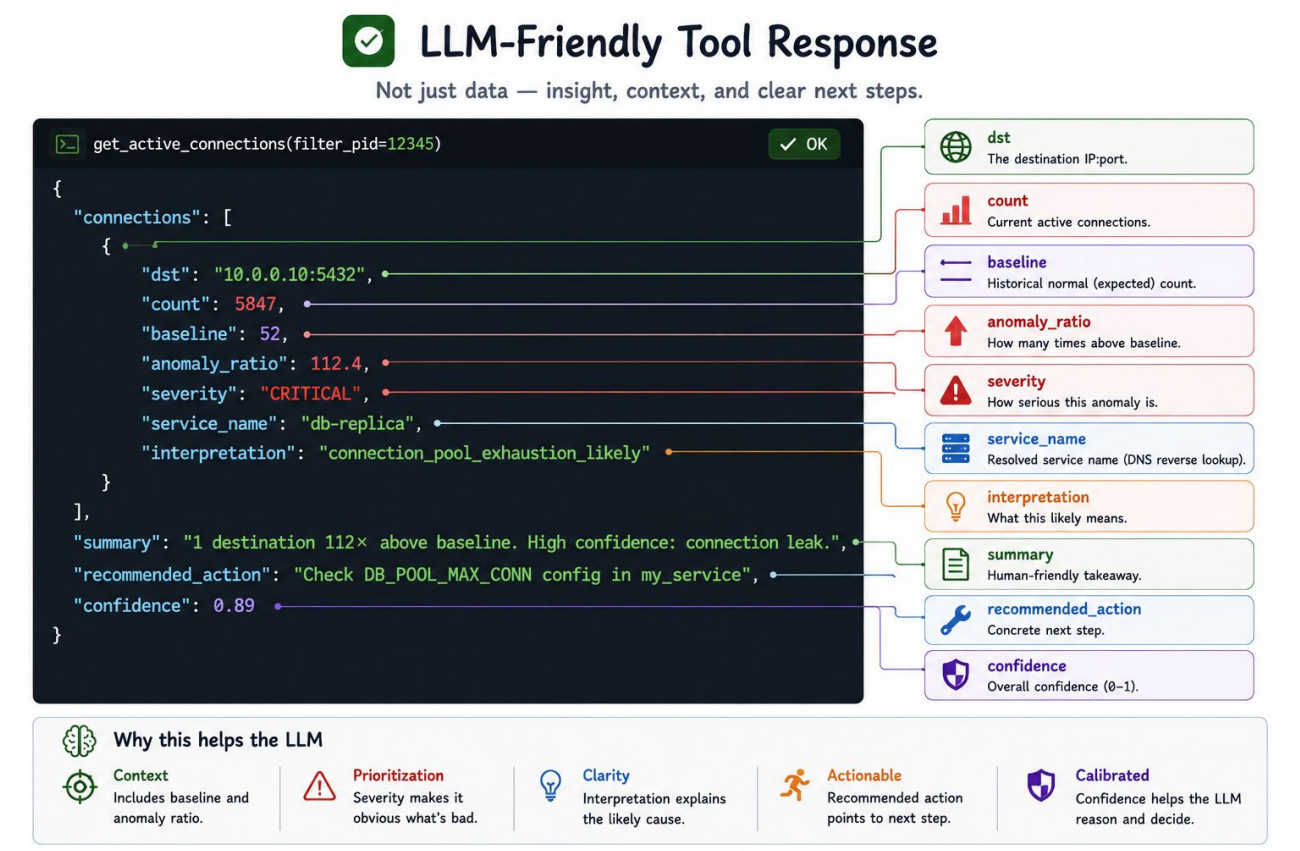
The tool response is a prompt engineering surface. Design it carefully.

Bad response (too open-ended):

```
python
{
 "connections": [
  {"dst": "10.0.0.10:5432", "count": 5847}
 ]
}
```

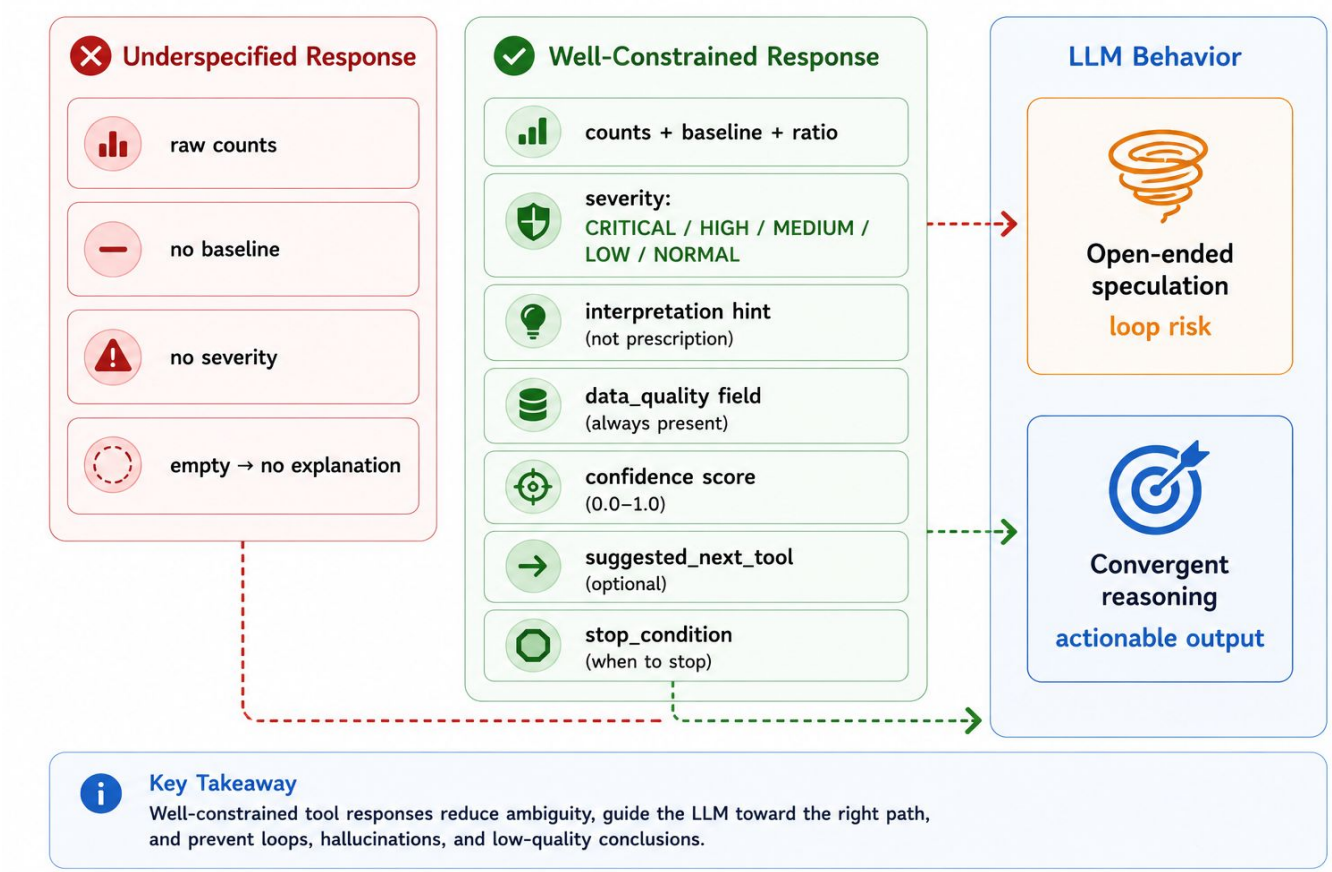
→ LLM says: **"There are 5847 connections. This could mean many things..."**

Good response (structured, bounded, actionable):



→ LLM says: "Connection pool exhaustion with 89% confidence.
Recommended: check DB_POOL_MAX_CONN."

Response Schema Design



The schema rule: Every field you add to a tool response is a constraint on the LLM's reasoning space. More constraints → less hallucination → faster convergence → safer system.

Pre-Flight Checklist

Before you ship an eBPF+MCP observability system

eBPF Layer

- [ ] **Ring buffer size tuned for your syscall rate** (monitor drop counter)
- [ ] **Fallback to BPF map aggregation** if ring buffer drops > 0
- [ ] **kprobes have filter_pid path** to avoid noise from non-target processes
- [ ] **eBPF programs tested** on your exact kernel version (BTF required)

Why it matters

- Prevent data loss
- Reduce noise
- Ensure compatibility
- Keep overhead low

MCP Layer

- [ ] **Every tool response includes `data_quality` field**
- [ ] **Baseline values are computed from a `rolling window`**, not hardcoded
- [ ] **`max_tool_calls` limit enforced at the MCP server level**
- [ ] **Tool response size bounded** (no unbounded `results` arrays sent to LLM)

Why it matters

- Enforces data honesty
- Prevents runaway loops
- Protects context window
- Improves reliability

LLM Layer

- [ ] **System prompt includes explicit stop condition** ("`stop if confidence > 0.85`")
- [ ] **System prompt instructs LLM to `report uncertainty`**, not fill it with speculation
- [ ] **Multi-turn conversation includes tool call history** for context continuity
- [ ] **Output is parseable** (ask LLM for **structured JSON** diagnosis, not prose only)

Why it matters

- Stops infinite loops
- Reduces hallucination
- Maintains context
- Enables automation

Ship with Confidence

A trustworthy system is built on data fidelity, bounded behavior, and clear guidance to the LLM.

- No silent data loss
- No infinite loops
- No speculation
- Structured, actionable output

Comparative: Sidecar vs eBPF+MCP

Feature	Sidecar Model	eBPF + MCP
Deployment	Per-pod sidecar container	DaemonSet (one per node)
Resource overhead	40–60% per pod	<1% per node
Data freshness	Scrape interval (15–60s)	Sub-second (ring buffer)
Visibility	App-layer metrics only	Kernel syscalls (all processes)
Reasoning	Human looks at dashboard	LLM reasons across tool calls
Diagnosis latency	Minutes (human)	Seconds (LLM)
Attack surface	Per-sidecar auth/config surface	`CAP_BPF` + read-only BPF maps
Kernel version req.	None	≥ 4.9 (kprobes), ≥ 5.7 (LSM)
Cold start	Container startup	eBPF program load (~100ms)

Not a replacement for metrics/tracing.
A complement that closes the kernel visibility gap.

Key Takeaways

1. Sidecars observe applications. eBPF observes the kernel.
They are not the same thing. The kernel sees what applications try to hide, miss, or haven't instrumented.

2. MCP makes kernel data queryable by reasoning systems.
The LLM pulls what it needs, when it needs it. No push pipeline, no stale dashboards.

3. Design tool responses as reasoning constraints, not just data.
Include baselines, severity, confidence, and data_quality. Every field narrows the LLM's hypothesis space.

4. Know your failure modes before you ship.
Tool-call loops (add stop conditions), hallucination on sparse data (add data_quality), ring buffer drops (monitor and aggregate).

5. The demo runs in Colab.
No kernel access required: simulation + real Anthropic API calls.
See companion notebook.

Companion Resources

eBPF.io resources : https://ebpf.io/get-started

cilium/ebpf (Go) : https://github.com/cilium/ebpf

Anthropic tool use docs : https://docs.anthropic.com/en/docs/tool-use

MCP spec : https://modelcontextprotocol.io

bpftrace MCP server: https://github.com/bpftrace-mcp

Thank You

- Harini Anand
- Connect: linkedin.com/in/harinianand