MCP Dev Summit · Bengaluru 2026

The stdio Deadlock Nobody Warned Us About

What we’re covering today

01 The Mystery

A perfect system goes completely silent. No errors, no warnings.

02 The Hidden Trap

A tiny, 64KB traffic jam hidden deep inside your computer.

03 The Smoking Gun

How to catch the exact moment your code freezes in under 5 seconds.

04 The Escape Plan

Three simple lines of code to keep the data flowing forever.

05 The Survival Guide

The ultimate checklist to make sure this never happens to your apps.

Wednesday morning. Everything is running. Nothing is working.

```
$ ps aux¦ grep orchestrator
user   7991   0.0   0.0   ...   S  [orchestrator]    - alive


$ ps aux¦ grep llama-server
user   6378   0.0   0.0   ...   S  [llama-server]    - alive


$ top
CPU:0.0%                                          - nobody home
```

Diagnosis

Processes: alive. Work: zero. Logic: broken.

THIS REALLY HAPPENED

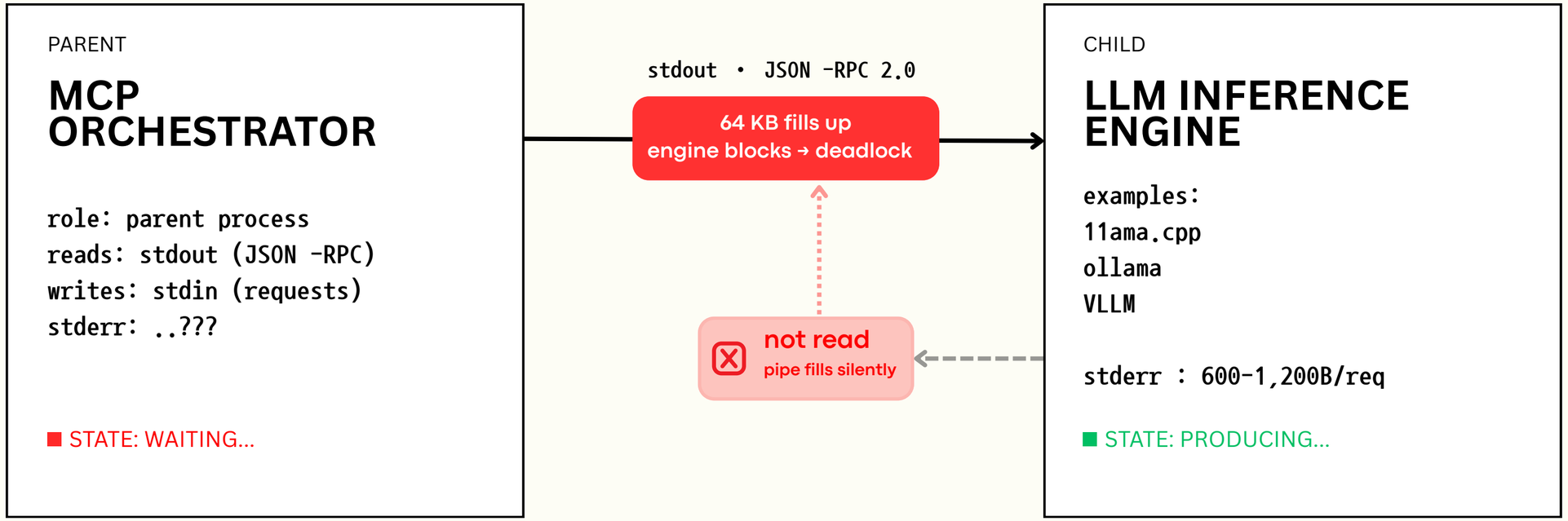
A real production system. 16+hours. 70+ zombie processes. Zero output.

[---------- 65,536 bytes ----------]

It's a pipe.
A 64-kilobyte pipe.

THAT'S IT. THAT'S THE WHOLE MYSTERY.

Two pipes. One protocol. One ignored.



16 pages. 65, 536 bytes. Zero mercy.

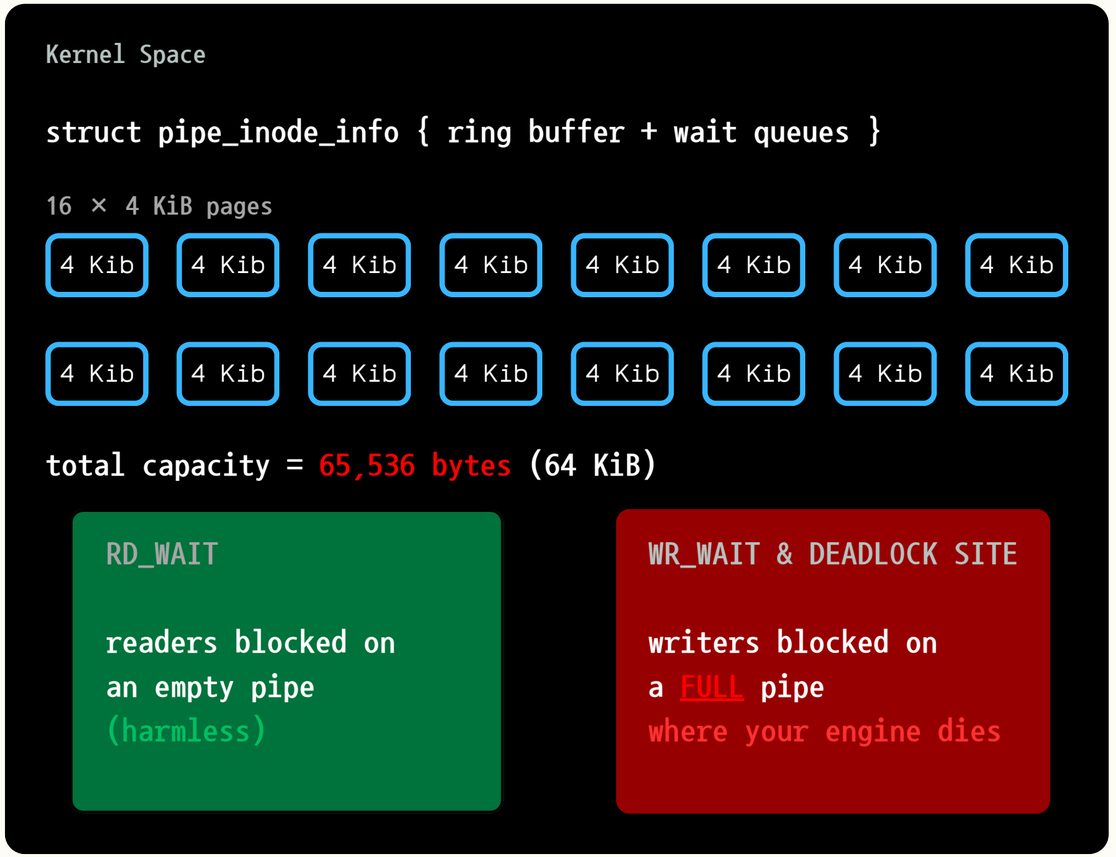
```
$ sys.stderr.write(line)
$ logging.info("eval...")
        ↓
$ write(2, buf, n)
```

WHAT'S HAPPENING

1.Every log call becomes a write(2) syscall, crossing into kernel space.

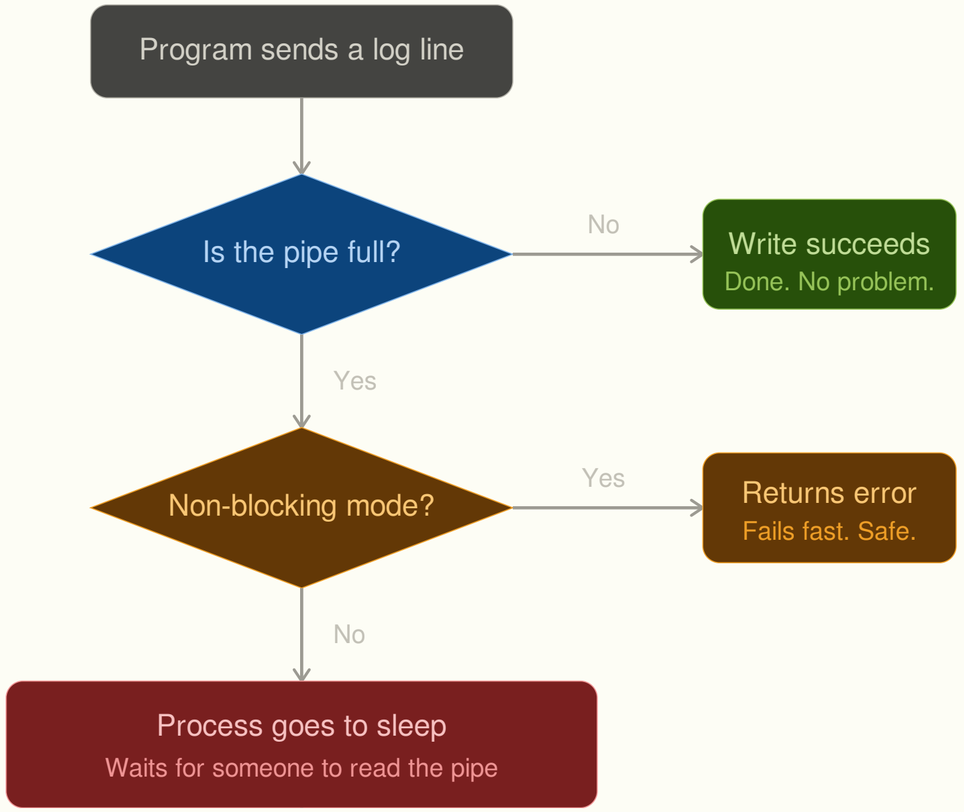
2.The kernel buffers writes in a 64 KiB ring buffer. Once full, the writer is blocked resulting no progress.

3.With no reader draining the pipe, the buffer never empties. Your process hangs indefinitely.



Why Your Program Freezes Without Warning

- Every print or log call puts text into a pipe, a small tube the OS uses to pass data between programs.
- That pipe has a fixed size limit of 64 KB. Once full, nothing more can go in.
- If the pipe has space, the message goes through instantly and everything is fine.
- If the pipe is full and nothing is reading from it, your program silently freezes and waits.
- It never times out. No error, no crash, no warning. It just stops, forever.



Your MCP server ticks every box.

This is not a bug · this is a design gap

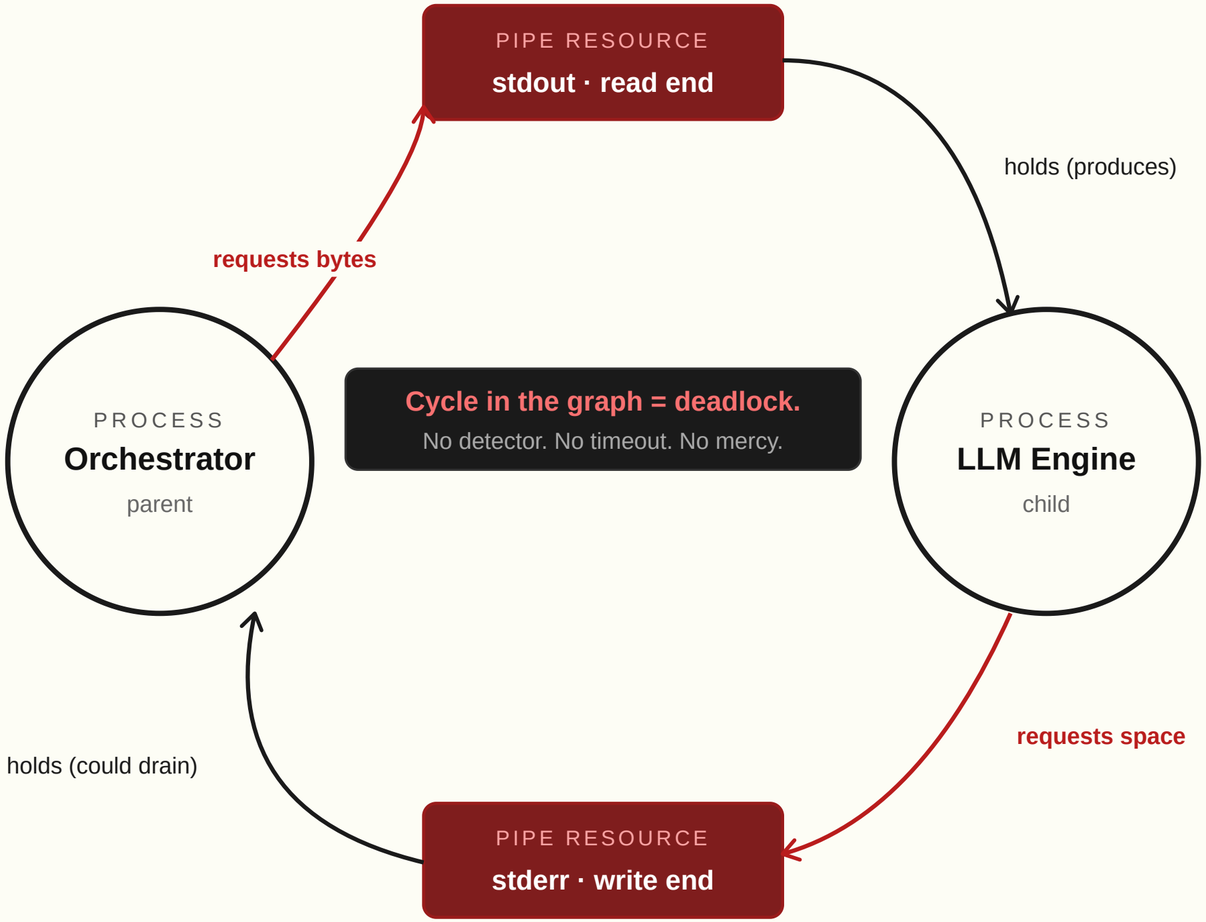
Condition	What it means in your server	Holds?
Mutual Exclusion	Pipe slot is locked until a reader consumes it	✓
Hold and Wait	Orchestrator holds the stdout FD open while waiting for bytes	✓
No Preemption	The kernel cannot force-drain the pipe for you	✓
Circular Wait	engine → stderr → pipe → orchestrator → stdout → engine Everyone is waiting on everyone else	✓

*All four conditions met. Textbook deadlock.

The kernel is not wrong. Python is not wrong. The MCP spec is not wrong.

The architecture is wrong.

Your MCP Server Is Already in This Loop!



Linux · macOS · Windows — none have a pipe deadlock detector. You are on your own.

Run two commands. See exactly where it's stuck.

When your MCP server hangs, this tells you why in under 5 seconds

LLM Engine · PID 2707

```
$ cat /proc/2702/wchan
```

pipe_write

Your engine is frozen here
Trying to log something, but the
stderr pipe is completely full.
It cannot move until someone reads it.

```
state: S (sleeping)      ->    cpu: 0.0%
syscall: write(2, ...)   ->    pipe @ 65,536 / 65,536
```

Orchestrator · PID 2210

```
$ cat /proc/2210/wchan
```

pipe_read

Your orchestrator is frozen here
Waiting for the engine to produce
output on stdout - but the engine
is stuck and will never respond.

```
state: S (sleeping)      -> cpu: 0.0%
syscall: read(0, ...)    -> awaits engine output
```

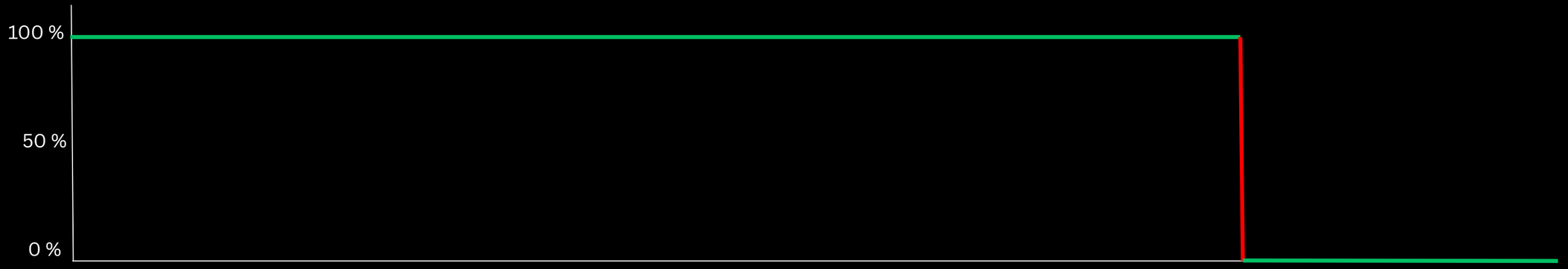
What is /proc/PID/wchan? Every Linux process has a file here. It shows the exact kernel function the process is sleeping inside. pipe_write means stuck trying to write. pipe_read means stuck waiting to read. Two frozen processes, each waiting on the other — that is your deadlock, confirmed.

Now fix your subprocess setup.

Works perfectly. Then hits 64 KB. Stops.

STDERR PIPE BUFFER - HOW FULL IS IT?

Your MCP server's productivity over time →



No slowdown. No warning. No error message.
100% productive → 0% productive. The cliff is vertical.

‘strace’ shows the exact moment it froze.

STRACE -P 5678 — ATTACHED TO LLAMA-SERVER

```
$ strace -p 5678 -e write,read
Process 5678 attached

write(2, "[INFO] slot 0: prompt eval time = 234ms", 44)                = 44 ✓
write(2, "[INFO] eval: token 891/1024 | 47.3 tok/s", 41)               = 41 ✓
write(2, "[INFO] kv cache: 78% utilized, n_ctx=4096", 42)              = 42 ✓

write(2, "[INFO] slot update: n_decoded=892, n_remain=132", 48
↑ NO CLOSING PAREN. NO RETURN VALUE. PROCESS IS INSIDE THE SYSCALL.

// Every write above returned instantly — pipe had space, moved on.
// This one never returned. Pipe was full. Process is frozen here.
// The murder weapon: a perfectly normal KV cache log line.
```

Every previous write returns immediately. Then one doesn't.

That's the moment.

Real engines. Real numbers. Real cliff.

How many requests until you freeze?

Engine	Log verbosity	Stderr bytes / request	Requests → pipe full
llama.cpp	default (v3)	600 – 1,200 B	**54 – 107**
llama.cpp	debug (v4)	4,000 – 16,000 B	**4 – 16**
llama-cpp-python	verbose=True (default!)	~3× llama.cpp	**18 – 36**
vLLM	default	~200 B / 5 sec	**hours**
Whisper.cpp	default	1,000 – 3,000 B / seg	**21 – 65**

LANGCHAIN'S LLAMACPP WRAPPER

Sets `verbose=True` by default. 36 inferences before freeze.

You didn't choose that setting. Someone made it for you. Generously. With love.

No OS timeout exists.

no ton Linux • not on macOS • not on Windows

Your process will sleep until you kill it manually.

Drain ALL pipes. Simultaneously. Forever.

WRONG

```
orchestrator:
   await proc.stdout.readline()
   [blocked waiting for data]

LLM engine:
   writes to stderr...
   [fills pipe]
   [BLOCKED in kernel wr_wait]

// both sleeping
// neither getting what they need
```

- They're waiting on each other. Nobody moves.
- One blocks, the other fills up. Game over.
- Two processes, both stuck, waiting forever.

CORRECT

```
asyncio.TaskGroup:
   ├── Task 1: drain stdout → parse JSON-RPC
   ├── Task 2: drain stderr → logger.debug()
   │      ↑ THE KEY
   └── Task 3: watch for process exit

// no pipe ever fills
// no deadlock possible
```

- All three run at once - no one ever has to wait.
- Three jobs, running side by side, always draining.
- Parallel tasks mean no pipe ever gets full.

Three lines changed. hang prevented.

mcp_runner.py

```python
import asyncio, logging

async def drain(stream: asyncio.StreamReader,
name: str, handler) -> None:
"""Continuously drain a pipe. Runs until EOF."""
try:
while True:
line = await stream.readline()
if not line: # EOF — child closed write-end
return
handler(name, line)
except asyncio.CancelledError: # clean shutdown
pass

async def run_mcp_server(cmd: list[str]) -> int:
proc = await asyncio.create_subprocess_exec(
*cmd,
stdin=asyncio.subprocess.PIPE,
stdout=asyncio.subprocess.PIPE,
stderr=asyncio.subprocess.PIPE,
limit=4 * 1024 * 1024, # 4 MiB — absorb debug bursts
)

async with asyncio.TaskGroup() as tg: # Python 3.11+
tg.create_task(drain(proc.stdout, "stdout", on_jsonrpc))
tg.create_task(drain(proc.stderr, "stderr", on_log))

return await proc.wait()
```

drain()

Loops forever, one line at a time. Stops only on EOF or cancel. Never blocks other tasks.

limit= 4 MiB

Default 64 KiB fills up instantly with verbose logging. 4 MiB gives breathing room.

TaskGroup

All three tasks run at once. If one crashes, the rest are cancelled cleanly. No leaks.

Five layers. Each catches a different failure.

Layer	What it catches?	How it works
L5	Stops crash-loops or runaway log explosions before they flood your system	**if stderr > 10 MiB → kill process**
L4	Kills a runaway inference or a loop that refuses to stop on its own	**wall > 10 min → SIGTERM + SIGKILL**
L3	Catches a silent process — alive, but stuck and producing no output	**no progress > X seconds → escalate**
L2	Detects a process that is technically alive but completely unresponsive	**MCP ping every 30s → disconnect**
L1	Wraps every individual tool call so a slow tool cannot block forever	**asyncio.wait_for(req, 30s)**

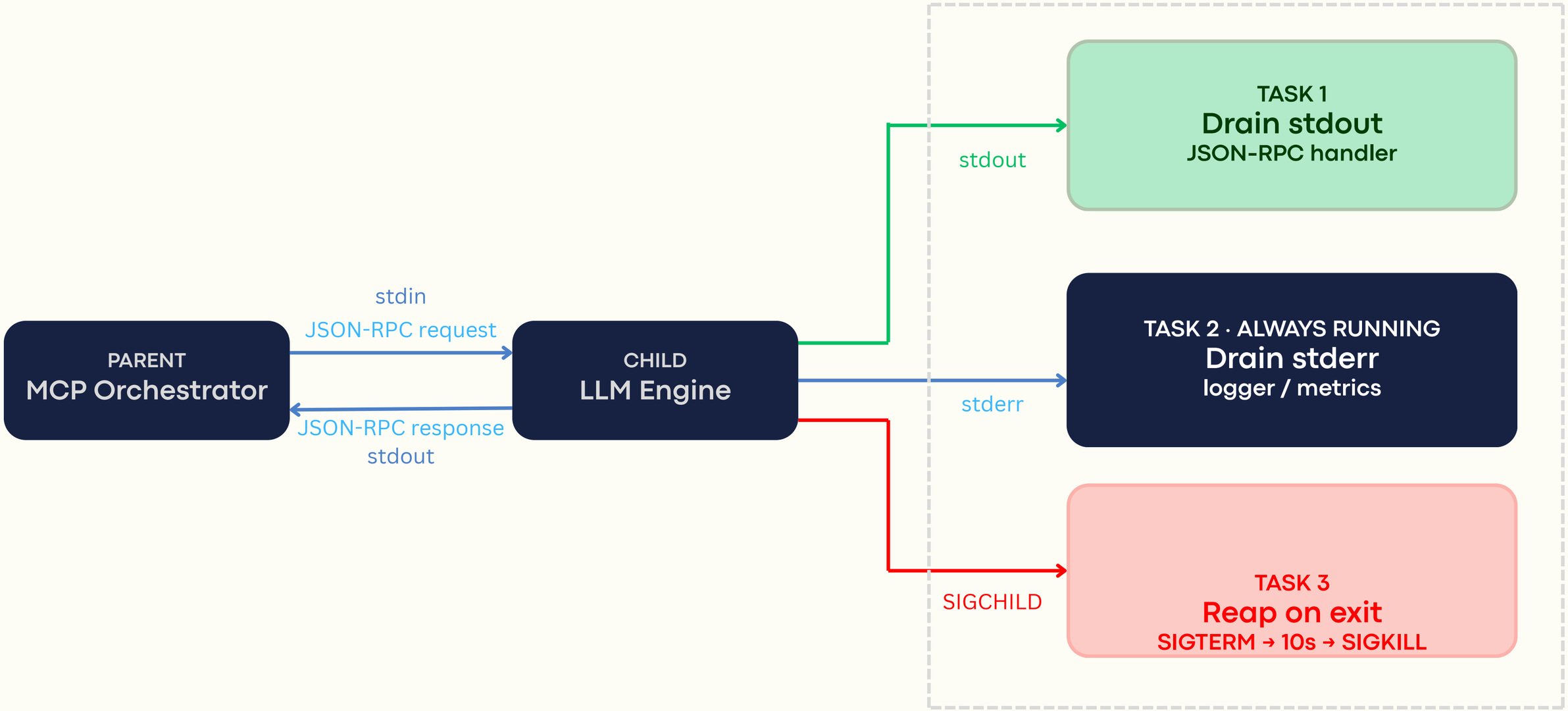
Layers stack - if L1 misses it, L2 catches it, and so on up the chain.

WHAT YOU ASSUMED (HTTP)	WHAT ACTUALLY HAPPENS (STDIO)
Connection dropped? Browser sends RST/FIN. Server knows instantly.	**Child process keeps running forever.** No goodbye signal exists. The pipe stays open.Your process has no idea anyone left.
Read timeout → connection closes. If a request takes too long, the socket shutsdown and everything stops cleanly.	**Timeout fires - process keeps running.** The timer gives up, but the child processis still alive, still spinning, still consuming memory.
Failed? Just reconnect. HTTP has a built-in retry handshake. Reconnecting is standard and automatic.	**There is no reconnect. Ever.** stdio is just pipes. No reconnect protocol.If it dies, you start over from scratch.

The Deadlock-Free Playbook

ALWAYS	NEVER
• Always read error output (stderr) in a background task - never let it pile up	• Never wait for the process to finish before reading its output - you'll deadlock
• Always read normal output (stdout) in a background task - same reason	• Never assume a timeout actually stopped the process - it probably didn't
• Always cap how much output you'll read (4 MB limit) so you don't run out of memory	• Never assume "no more output" means the child process is dead - it isn't
• Always start your child process with output flushing on - so you actually get the data in real time	• Never use HTTP-style timeouts on a direct pipe connection - they don't work the same way
• Always shut down gracefully: ask nicely first (SIGTERM), wait 10 seconds, then force-kill (SIGKILL), then wait for it to fully die	• Never skip the 200 KB stderr stress test - if your orchestrator can't finish it in under 10 seconds, it will hang in production

Three tasks. All pipes drained. Zero deadlock.



DRAIN everything.

We learned this the hard way so you don't have to.

The stdio Deadlock Nobody Warned Us About

THANK YOU!

Yuvraj

Archana