Enabling Robust Parallel Tool Calling in Agentic AI via OS-Level Scheduling for the iPWR Simulator

Seongsu Chae [a, b], Yonggyun Yu [a], Seung Geun Kim [a], Yujong Kim [a*]
[a] *Applied Artificial Intelligence Section, KAERI, Daejeon 34057, Republic of Korea*
[b] *Department of Computer Engineering, Hanbat National University, Daejeon 34158, Republic of Korea*
[*] *Corresponding author: yjkim@kaeri.re.kr*

***Keywords :** Model Context Protocol, parallel tool calling, agentic AI, OS-level scheduling, iPWR simulation

1. Introduction

Modern nuclear plant operations prioritize safety and procedural compliance. As digital Instrumentation and Control (I&C) and simulators generate increasingly high-frequency data, operators must simultaneously log events, check alarms, perform routine calculations, and track trends, which raises cognitive load and the risk of delay or omission. In this context, automation should be introduced to assist operators rather than replace them, by handling procedurally defined repetitive tasks and reducing the risk of human error in nuclear power plant operations. Within these boundaries, agentic artificial intelligence (Agentic AI) is useful not for autonomous operation but as a tool-invocation mechanism operating under predefined authority and procedural constraints [1]. It automates repetitive monitoring and structured calculations, records actions and their rationales for auditability, and supports decisions under human review. At the same time, to maintain safety, the system must continuously acquire and ingest simulator state and instrumentation data while always providing immediate responses to operator queries so that operator interaction is not interrupted.

However, large language model (LLM) hosts that underpin such agentic configurations generally enforce session idle timeouts. In a sessionized host composed of LM Studio to utilize the LLM and a Model Context Protocol (MCP) server for tool control, long-running tasks implemented as threads or subprocesses are terminated when the session closes, or tool-call issuance stalls. This degrades the user's ability to check results in time and take action.

We address this constraint with a minimal pattern that decouples long-running tasks from the host session and delegates their lifecycle to the Operating System (OS) scheduler through OS-level reparenting. Workers start in a detached state. Termination is handled safely at sampling-cycle boundaries by polling a file-system sentinel such as stop.txt. At invocation time an acknowledgment is always returned immediately so operator interaction is never blocked, and completion results from long-running tasks are delivered even if the original session drops. The focus is on execution-and-delivery semantics rather than a particular framework. Although we target an on-premises LM Studio plus MCP deployment, the pattern applies to other hosts with similar session constraints.

For evaluation we use the iPWR simulator as a testbed [2]. It reproduces representative pressurized water reactor scenarios in the SMR family and allows simultaneous composition of long tasks such as cooldown or heatup [3] with continuous monitoring and short tasks such as reading neutron power at a specific instant. Controlled scenarios and consistent monitoring support reproducible comparative experiments that assess parallel invocation and task reliability safely and repeatedly.

We define the performance concepts used in the paper as follows. Continuity means task survival when a session terminates and guaranteed result delivery from a later session if needed. Responsiveness means always providing a first visible response upon request and accepting subsequent commands without delay. Non-blocking means one task does not prevent other tool calls, so multiple tools can be invoked and responded to immediately even while a long-running task is in progress.

In summary, we propose execution-and-delivery semantics that let operators observe and act without interruption under host session timeouts. We validate the approach on the iPWR simulator and then present an empirical analysis of session-bound termination, the design and implementation of the detach-and-reparent pattern, and an evaluation against these metrics.

2. Concept: Parallel Tool Calling for MCP

2.1. Problems with Synchronous MCP Server

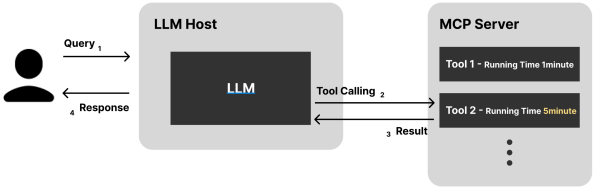

Fig. 1. Conventional synchronous MCP server workflow.

In conventional MCP servers, tool invocation is synchronous: once a user calls a tool, the session blocks until the tool finishes, and no other actions can be performed in the meantime. In particular, long-running

tasks such as continuous monitoring or reactor heatup/cooldown monopolize the session, until they complete, operators cannot invoke other tools, undermining the immediacy required for query and control.

To overcome this limitation, we execute long-running jobs in the background while handling short, interactive tools in parallel, with the goal of simultaneously achieving non-blocking operation, continuity of data collection, and interactive responsiveness.

2.2. Initial Idea

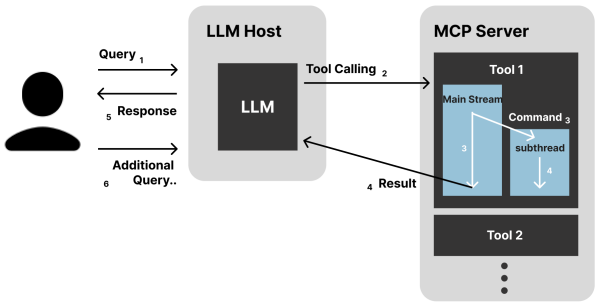

Fig. 2. Multithreaded MCP server tool invocation workflow.

Each tool endpoint in the MCP server operates as the primary request–response stream. When a long-running tool (e.g., monitoring or heat-up/cool-down) is invoked, the endpoint spawns a background subthread for that task and returns immediately. This allows the LLM host to continue servicing subsequent query/tool calling requests, while the long-running task proceeds in a separate thread that performs periodic data acquisition and logging.

The experimental validation of this idea, its limitations, and the refined design are presented in Sections 3–4.

3. Baseline Experiment & Root Cause Analysis

To test the initial idea described in Section 2.2, we ran a baseline on the iPWR simulator in Windows 10 with LM Studio as the host with idle timeouts enabled, an MCP server built on FastMCP [4], and the Qwen2.5-32B-Instruct model. The long-running tool was implemented as both a subordinate thread and a subprocess, each launched by the tool endpoint. After starting the long task for measured-value monitoring, the endpoint returned immediately, a short read of neutron power followed at once, the host remained non-blocking, and monitoring value updated at 1 Hz initially. After about 15 seconds of inactivity, the host idle timeout triggered session cleanup, the MCP-parented worker was terminated, and data and log updates stopped. In sum, the baseline confirmed responsiveness and non-blocking behavior, but continuity failed at the idle-timeout boundary.

To understand this failure, we examined the system in the same experimental environment as the baseline and repeatedly observed that long-running tasks (threads and subprocesses) were terminated once the host's session idle timeout (~15 sec) elapsed. Consequently, background data collection ceased immediately after the idle period; to identify the program responsible for the session teardown, we tracked changes in active processes using Windows Task Manager.

As a result, the observations converged on a single mechanism. Specifically, the MCP server and the threads/processes it creates are bound to the LM Studio (host) session. When the session idle timeout is reached, the host performs session cleanup, during which we observed one JavaScript process and two Python processes initiating the teardown. Thus, the MCP server process and its dependent workers (threads/subprocesses) are terminated together. For clarity, in this paper "subthread" denotes an additional execution path created within the same MCP server tool; "subprocess" denotes a separate child process created by the MCP server tool; and "session-management processes" refers to host-launched auxiliary runtimes (e.g., Node.js or Python) that maintain the session and transport. The specific manifestations are as follows.

Table I: Session-Bound Termination Patterns of MCP Workers under Host Idle Timeout

Worker Type	Termination Patterns
Subthread	When the MCP server process is terminated during session cleanup, child threads vanish with it → long-running tasks are interrupted.
Subprocess	Even with a separate process boundary, the parent lineage remains the MCP server; when the session ends, the child process is terminated → liveness cannot be ensured.
Selective termination of session-management processes	Forcibly killing a particular python/node spawned by the host may give a brief illusion of survival, but it soon breaks host query handling → unsuitable for practical use.

All failures stem from the session-coupled lifecycle between the LLM host (LM Studio) and the MCP server. As long as the parent–child lineage resides under the host session, workers are reaped when the host cleans up the session/pipe at idle timeout. Therefore, the necessary remedy is to transfer parentage to the OS and grant the worker an independent lifecycle (detached, OS-parented). This requirement is realized in Section 4 via session detachment and OS scheduler re-parenting combined with a sentinel-based graceful shutdown.

4. Solution Design & Implementation

We separated the problematic long-running MCP tool into an independent Python script and created a batch file (.bat) to invoke it. We then registered that batch file with the Windows Task Scheduler [5] and triggered it for immediate execution "run now". This makes the worker OS-parented rather than host-session-parented, so it continues running regardless of the host's session idle timeout. For termination, the worker checks on each iteration for the presence of a file-system sentinel (stop.txt); upon detection, it exits the loop and shuts down cleanly. The execution and shutdown flow is summarized in Fig. 3.

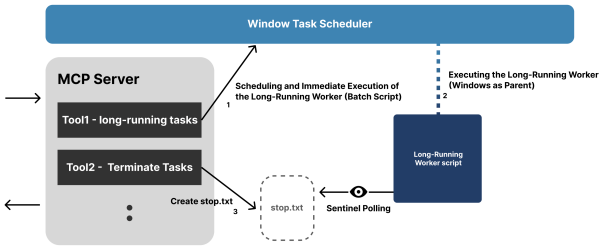

Fig. 3. Scheduling the batch-launched long-running script via Windows Task Scheduler and graceful shutdown using a file-system sentinel (stop.txt).

Building on the workflow in Fig. 3, we implemented the approach as follows.

```
# MCP Server Tool 1
# …
subprocess.run(
    [
        'schtasks', '/Create', '/TN', task_name,
        '/TR', run_command, '/SC', 'ONCE',
        '/ST', run_time, '/F',
    ], check=True, capture_output=True,
)
subprocess.run(
    [
        'schtasks', '/Run', '/TN', task_name,
    ], check=True, capture_output=True,
)
subprocess.run(
    [
        'schtasks', '/Delete', '/TN',
         task_name, '/F',
    ], check=True, capture_output=True,
)
```

Code 1. Invoking Windows Task Scheduler (schtasks) from Python

Figure 3's core flow is implemented with the three schtasks calls in Code 1. The MCP server's tool connects the pre-prepared long-running script to a batch file (.bat) and registers a one-shot task with subprocess.run(['schtasks','/Create', …]), then starts it immediately with ['schtasks','/Run', …]. Here, task_name is a unique identifier for the job (e.g., measured value monitoring task), run_command is the batch file that launches the long-running script, and run_time is a scheduled time required by the interface, while the actual immediate start is performed by /Run. After launch, ['schtasks','/Delete', …] removes the entry to keep the scheduler namespace clean. The worker script executes the long-running job at a fixed period, for example 1 Hz, and on every iteration it polls for the presence of the file-system sentinel stop.txt. When the user invokes the termination tool, the MCP server creates stop.txt in the corresponding workspace; the worker detects it in the loop and exits to shut down cleanly.

This arrangement re-parents the worker to the Windows Task Scheduler rather than to the host session, granting an independent lifecycle that is unaffected by the host's session idle timeout and decoupling the long-running execution from the request/response path so the server can return immediately, effectively enabling other tool invocation. We validated the approach on the iPWR simulator; Section 5 reports the results.

5. Application to the iPWR Simulator and Demonstration

This section applies the parallel tool-calling method of Section 4 to the iPWR simulator and presents an end-to-end usage flow. The demonstration proceeds with two long-running tasks in parallel and an immediate short call: (i) launch reactor cooldown; (ii) start measured-value monitoring concurrently; and (iii) then invoke the short task read neutron power to obtain the value immediately. Figure 4 illustrates that the host returns a response right after the long-running task is started, enabling the user to issue additional tools without waiting.

Fig. 4. iPWR demo of non-blocking parallel tool invocation: (a) continuous monitoring with concurrent reactor cooldown; (b) immediate host responses including a short tool call (read neutron power) while long-running tasks continue.

In the baseline experiment of Section 3, tool execution stopped the moment session cleanup began due to the host's idle timeout (approximately 15 seconds). The worker dependent on the MCP server was reaped and terminated, the monitoring task halted, and, upon forcibly terminating the program that manages the session, further tool calls became impossible. In contrast, after applying the detachment and OS reparenting proposed in this paper, even under the same baseline environment, timestamps continued at the configured 1 Hz period, reactor cooldown proceeded concurrently, and the short task read neutron power responded immediately upon invocation. From these results, we emphasize three properties in the iPWR demo:

Table II: iPWR demo—key properties

Property	Observation & Evidence
Responsiveness	Even when a long-running task is started, the MCP server returns an immediate reply.
Non-blocking	Because the long-running worker is OS-parented and continues in the background, the user can invoke other control/query tools without waiting (e.g., run cooldown while monitoring is active).
Continuity	Continuous collection for the long-running task is maintained regardless of the host session idle timeout. During the demo, monitor.csv timestamps continued at the configured 1 Hz period without omission; upon a stop request, the worker detected stop.txt and terminated the loop(clean shutdown).

In summary, this application case demonstrates in the iPWR environment that simultaneous invocation of long-running tools (monitoring/cooldown) achieves uninterrupted parallel tool calling, and that, thanks to immediate responses, short tasks such as reading neutron power can be processed without issue.

6. Conclusions

This study resolves the session-bound lifecycle problem, where long-running tasks are prematurely terminated by the host session idle timeout, by using session detachment and OS scheduler re-parenting together with file-sentinel-based loop termination. In the iPWR simulator the method ran continuous monitoring in parallel with reactor cooldown while a short read of neutron power returned immediately, demonstrating continuous collection, immediate responses, and non-blocking concurrent operation without mutual interference. The method operates reliably for long-duration activities in complex reactor operating procedures and maps directly to procedures such as heatup and cooldown, boron concentration adjustment, and control-rod maneuvers.

Going forward, we will consider linking the present methodology's parallel tool-calling MCP server with a large language model trained on nuclear domain knowledge and the iPWR operating manuals, and we plan to evaluate, beginning with small pilots, a human-in-the-loop operation in which standardized repetitive tasks are proposed and executed by the model while critical decisions and controls proceed only with operator review and approval. In parallel, without disrupting the existing Main Control Room workflow, we will emphasize auditability (action and rationale logs), separation of duties, and reinforcement of procedural compliance, with the objective that long-running monitoring continues under host idle timeouts and that short queries and control actions are handled without delay, thereby contributing to practical automation.

On the other hand, because MCP exposes tools through a common protocol with stable schemas and transport, adding, swapping, or relocating capabilities is comparatively easy, and portability can be expected across OS-native job schedulers on Windows, Linux, and macOS and across diverse LLM hosts; accordingly, we will consider packaging an iPWR-simulator-dedicated MCP server together with the iPWR simulator as a training and practice kit so that parallel orchestration and fault-recovery procedures can be practiced safely and repeatedly without any plant intervention, contributing as an educational product.

ACKNOWLEDGEMENTS

This work was supported in part by Korea Atomic Energy Research Institute R&D Program under Grant KAERI-524540-25.

REFERENCES

[1] Y. P. Lee and J. Cha, *Large Language Model Agent for Nuclear Reactor Operation Assistance,* Nuclear Engineering and Technology, Vol. 57, Article 103842, 2025. doi:10.1016/j.net.2025.103842

[2] International Atomic Energy Agency (IAEA), *Integral Pressurized Water Reactor Simulator Manual*, Training Course Series No. 65, Vienna: IAEA, 2017.

[3] International Atomic Energy Agency (IAEA), *Integral Pressurized Water Reactor Simulator Manual: Exercise Handbook*, Training Course Series No. 65, Vienna: IAEA, 2017.

[4] FastMCP, *Welcome to FastMCP 2.0!, Getting Started*, 2025. (accessed Jul. 25, 2025).

[5] Microsoft, *Task Scheduler Reference, Microsoft Learn*, 2019. (accessed Aug. 7, 2025).