UbiClaw: Foreground-Preserving Background LLM Inference for On-Device Systems

Jeonghoon Park
hoonably@unist.ac.kr
Department of Computer Science and Engineering, UNIST
Ulsan, Republic of Korea

Jaemin Kim
jm611@unist.ac.kr
Department of Computer Science and Engineering, UNIST
Ulsan, Republic of Korea

Seongwoon Jo
whtjddns1234@unist.ac.kr
Department of Computer Science and Engineering, UNIST
Ulsan, Republic of Korea

Jongwon Lee
belluno@unist.ac.kr
Department of Computer Science and Engineering, UNIST
Ulsan, Republic of Korea

Abstract

On-device LLMs are useful as background assistants, but token generation competes with the foreground app for CPU, GPU, memory bandwidth, power, and thermal headroom. Standard LLM benchmarks optimize tokens per second in isolation, which misses the user-visible failure mode: the foreground app becomes less responsive. UbiClaw reframes the problem as *foreground-preserving background inference*: maximize background LLM throughput subject to a foreground responsiveness service-level objective (SLO). UbiClaw first profiles the foreground workload alone to derive an app-specific FPS SLO and hardware pressure profile. During overlap, it observes SLO deficit and CPU/GPU/DRAM pressure, then controls two lightweight actuators: token-boundary decode delay and operating-system QoS demotion. On 2D and 3D rendering workloads, UbiClaw reduces cumulative SLO deficit by 96.8% and 94.4% relative to no throttling, while retaining substantially more throughput than conservative static throttling. The main contribution is therefore not only the controller, but the metric and objective that make foreground responsiveness a first-class constraint.
Project Page: https://jeonghoonpark.com/project/ubiclaw

Keywords

on-device LLM, foreground SLO, mobile computing, resource scheduling, QoS

1 Introduction

Local LLM inference is no longer only an explicitly launched benchmark or developer workload. Recent desktop and phone platforms increasingly expose language models as ambient assistants that remain available while users continue working in foreground applications. Apple Intelligence integrates on-device foundation models into macOS and iOS for everyday actions such as writing assistance, notification summarization, and in-app actions; Microsoft's Windows AI APIs expose Phi Silica as a hardware-accelerated local language model for application features such as summarization and rewriting; and Android's AICore manages Gemini Nano as a system-level service for private, offline generative AI [3, 6, 15]. This setting changes the systems objective. A background LLM with high tokens/s is not useful if it makes the foreground application drop frames. For user-facing on-device computing, the foreground application is the user-visible workload, so the right question is: *how fast can the LLM run without harming the foreground experience?*

UbiClaw answers this question by protecting a foreground SLO. The runtime first measures the foreground app alone, derives a baseline FPS, and defines an acceptable lower bound. During foreground+LLM overlap, it continuously measures recent FPS and hardware pressure. If the foreground falls below the SLO, UbiClaw makes the LLM more polite by increasing decode delay, demoting the LLM worker's QoS, or both. If the foreground later has enough headroom, it restores the most recent throttle action conservatively.

The project's central contribution is the problem formulation. Instead of treating LLM throughput as the only metric, UbiClaw introduces foreground SLO violation time and cumulative SLO deficit as first-class metrics, then maximizes tokens/s inside that constraint. This exposes an otherwise hidden failure mode: a policy can look excellent by LLM metrics while producing an unacceptable foreground experience.

2 Problem Definition and Motivation

The motivating workload is not that every user intentionally runs a 9B model while gaming. Instead, the broader trend is that local assistants are becoming resident OS/app services whose requests can naturally overlap with latency-sensitive foreground work. A user may ask an assistant to summarize documentation, explain an error, generate notes, tune settings, or provide system guidance without leaving a renderer, media pipeline, or game. NVIDIA's G-Assist is one concrete graphics-oriented example: it is a local assistant for RTX systems that responds to voice/text prompts and helps with game and system optimization [17]. We therefore use a 9B-class local model as a representative stress point for upper-end privacy-preserving local assistant workloads on commodity unified-memory systems, not as a claim that every deployed assistant has this exact size.

2.1 A constrained objective

A local LLM shares finite on-device resources with the foreground app. On Apple Silicon and many unified-memory SoCs, CPU cores, integrated GPUs, and accelerators access a shared memory system. Unified memory is useful for local LLMs because it avoids explicit CPU-GPU copies and supports large models in a common memory

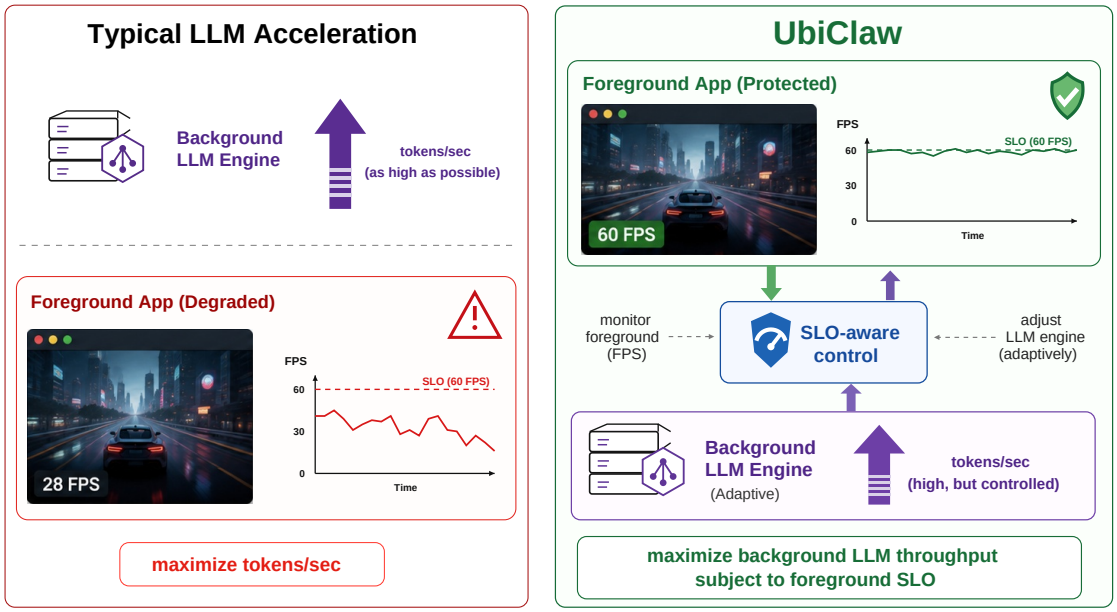


Figure 1: Problem formulation. Conventional local-LLM optimization maximizes background tokens/s, which can degrade the foreground app. UbiClaw instead maximizes background LLM throughput subject to a foreground SLO.

pool, but it also means foreground rendering and background decode can contend for GPU service time and DRAM bandwidth [7]. This motivates the foreground-preserving objective:

$$\max_{\pi \in \Pi} R_{\text{LLM}}(\pi) \quad \text{s.t.} \quad \text{Debt}(\pi) \le \epsilon_D,\ \text{ViolationTime}(\pi) \le \epsilon_T, \quad (1)$$

where π is a runtime policy, R_{LLM} is average LLM tokens/s, and Debt and ViolationTime measure foreground degradation. Equation 1 is the key design choice: LLM throughput is optimized only after foreground responsiveness is protected.

2.2 Foreground SLO metrics

The foreground SLO is derived from a foreground-only baseline rather than fixed globally. Let f_{base} be the baseline mean FPS. The default threshold is $\text{SLO}_{\text{FPS}} = 0.9 f_{\text{base}}$. During overlap, recent FPS $f(t)$ is converted into:

$$\begin{aligned} \text{Deficit}(t) &= \max\left(0, \frac{\text{SLO}_{\text{FPS}} - f(t)}{\text{SLO}_{\text{FPS}}}\right), \\ \text{ViolationTime} &= \sum_t \Delta t \cdot \mathbb{1}[f(t) < \text{SLO}_{\text{FPS}}], \\ \text{Debt} &= \sum_t \text{Deficit}(t) \cdot \Delta t. \end{aligned} \quad (2)$$

Violation time captures how long the foreground is below the SLO; cumulative deficit captures both duration and severity. This distinction matters because a short mild dip and a long severe drop should not be scored equally.

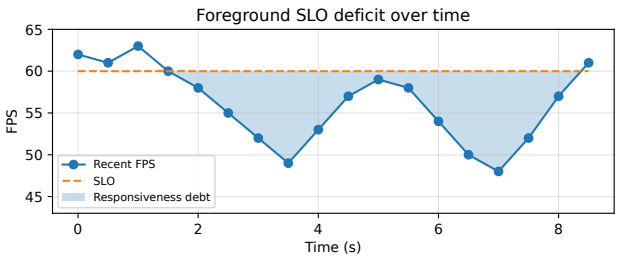


Figure 2: Foreground SLO metrics. The shaded area is cumulative SLO deficit, or responsiveness debt.

3 Related Work and Positioning

LLM serving and scheduling systems. Recent LLM serving systems optimize the LLM request path. vLLM improves throughput via PagedAttention and KV-cache memory management [13], while Sarathi-Serve and DistServe schedule prefill/decode phases under LLM latency constraints [1, 19]. HeteroInfer, CLONE, and Agent.xpu further optimize local or edge LLM execution through heterogeneous acceleration, latency/energy adaptation, and LLM-flow scheduling [8, 16, 18]. These systems can speed up the background engine, but their protected entity is still the LLM request rather than a co-running foreground app.

On-device LLM efficiency. MobileLLM, LLM in a Flash, and Apple-Silicon characterization work make local assistants more practical by reducing model cost, handling limited DRAM, or characterizing unified-memory inference pressure [2, 7, 14]. They motivate local assistant workloads, but do not define an app-derived SLO for foreground work that overlaps with assistant inference.

Table 1: Positioning of UbiClaw across related system lineages.

Lineage	Main objective	Why it is not enough for UbiClaw
LLM serving systems [1, 13, 19]	LLM request throughput/latency	Protects LLM-side SLOs, not a co-running foreground app.
On-device LLM systems [2, 7, 14]	Local model efficiency	Makes local inference feasible, but not foreground-aware.
Edge/SoC LLM scheduling [8, 16, 18]	LLM latency/energy and flow scheduling	Optimizes AI flows rather than foreground responsiveness.
SLO-aware GPU/ML serving [9, 10, 12]	Inference SLOs and GPU sharing	Requires serving/GPU-control assumptions beyond lightweight app-level control.
OS QoS principles [4, 5, 11]	Tail latency and priority	Supplies principles, not a foreground-preserving LLM objective.
UbiClaw	**LLM tok/s under foreground SLO**	**SLO is the constraint; delay/QoS are online actuators.**

SLO-aware ML serving and GPU resource control. Clockwork, GPU partitioning, and LithOS show how request-level SLOs and GPU sharing can improve ML serving predictability [9, 10, 12]. Their focus is server-side inference isolation or GPU scheduling. UbiClaw instead uses lightweight user-level controls on commodity on-device systems: foreground FPS debt is the correctness signal, QoS demotion provides scheduler-level politeness, and decode pacing handles GPU/DRAM contention that CPU priority alone cannot directly control.

4 Technical Approach

4.1 Two-stage runtime

UbiClaw has two stages. In the foreground-only lead-in, the runtime collects frame intervals and hardware telemetry. It computes the foreground SLO and an initial pressure profile from CPU utilization, GPU utilization, process GPU time, and DRAM bandwidth. The pressure score is a simple heuristic:

$$P = 0.10P_{\text{CPU}} + 0.35P_{\text{GPU}} + 0.25P_{\text{procGPU}} + 0.30P_{\text{DRAM}}. \quad (3)$$

The score is not intended to be a learned resource model. It is a coarse admission signal for the first overlap decision and a bottleneck hint for choosing the first actuator. We weight GPU utilization and DRAM bandwidth most heavily because the target interference pattern is foreground rendering plus LLM decode on a unified-memory device; process GPU time separates global GPU pressure from pressure caused by the experiment process; CPU receives a smaller weight because CPU priority can already be addressed through QoS demotion. If the foreground already places high pressure on GPU or DRAM, the LLM starts at Background QoS; if pressure is moderate, it starts at Utility; otherwise it starts at Normal. This avoids beginning overlap with an overly aggressive LLM configuration.

In the overlap phase, the controller receives foreground FPS observations and system telemetry. It then updates the LLM engine through two controls: decode delay and QoS mode. Decode delay inserts a sleep between token decode iterations, reducing the frequency of memory- and GPU-intensive LLM work. QoS demotion lowers the OS scheduling priority of the LLM worker so user-visible work can take precedence.

4.2 Closed-loop control

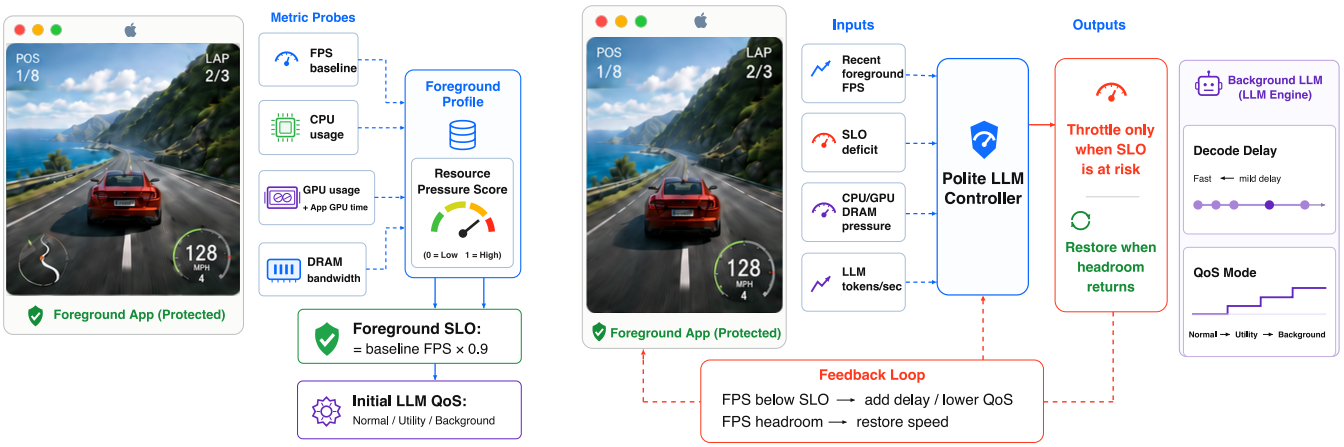
The controller classifies each observation into three zones: deficit, near-SLO, and healthy. A 3% deadband avoids oscillation around the SLO. Two consecutive deficit decisions are required before throttling, while four consecutive healthy decisions are required before recovery. Throttling can occur every 0.6 s, but recovery waits 1.8 s. This asymmetry clamps down quickly when the foreground is harmed and releases slowly to avoid reintroducing contention.

Algorithm 1 summarizes the policy. Severe deficits (≥20%) can trigger both delay increase and QoS demotion. If CPU pressure dominates, QoS demotion is preferred. If GPU or DRAM pressure dominates, delay is preferred. The delay increases in small steps and is capped at 160 ms. Recovery uses a last-in-first-out stack: when the foreground has sustained headroom and hardware pressure is below a recovery gate, the most recent throttle is undone first.

The threshold values are conservative control parameters, not claimed optima. The 3% deadband filters small FPS jitter around the SLO, while the 6% healthy margin requires visible headroom before recovery. The throttle cooldown is shorter than the recovery cooldown because foreground degradation should be corrected quickly but released slowly. The 160 ms delay cap prevents decode pacing from becoming an implicit pause; beyond that point, QoS demotion is the safer remaining action. Since the policy re-observes SLO deficit after every action, pressure weights only affect which actuator is tried first, not the correctness criterion.

Algorithm 1 SLO-aware Polite LLM control

```
Observe recent FPS f, SLO SLO_FPS, and pressure profile P
δ ← (SLO_FPS − f)/SLO_FPS
if δ > 0.03 for two decisions and throttle cooldown expired then
    if δ ≥ 0.20 then
        increase delay and demote QoS if possible
    else if CPU pressure dominates then
        demote QoS
    else if GPU/DRAM pressure dominates then
        increase decode delay
    else
        apply the least intrusive available throttle
    end if
    push action to throttle-history stack
else if f > 1.06 · SLO_FPS for four decisions and recovery cooldown expired then
    if pressure is below recovery gate then
        pop and undo latest throttle
    end if
else
    no operation
end if
```



(a) Foreground-only profiling. **(b) Closed-loop overlap control.**

Figure 3: Technical overview. During lead-in, UbiClaw derives an FPS-based foreground SLO and a foreground pressure profile. During overlap, it monitors FPS and hardware pressure, then actuates decode delay and QoS mode to protect the foreground app.

4.3 Implementation

The prototype uses an in-process llama.cpp Metal backend. The Swift runtime passes decode delay and QoS mode through an Objective-C++ bridge into the generation loop. The loop reads these atomic values at token boundaries, applies the current QoS, and sleeps before token sampling. This design is simple and deployable: it does not require custom Metal kernels or hardware partitioning, yet it can react during an active generation. Greedy decoding is used to reduce sampling variance. Logs include phase markers, foreground FPS, controller actions, LLM throughput summaries, and CPU/GPU/DRAM telemetry, allowing post-run alignment of SLO debt, actions, and tokens/s.

The controller is lightweight by construction: each decision is an $O(1)$ update over scalar telemetry, and actions only update atomic delay/QoS variables read at token boundaries. The full telemetry and overhead path is summarized in Appendix C.

4.4 Execution scenario and measurement pipeline

Each overlap run separates model loading from measured interference. The in-process model is preloaded, the foreground workload runs alone to establish both FPS baseline and hardware profile, and only then does the LLM begin generating tokens. The logger records phase markers, recent FPS, SLO deficit/debt, hardware snapshots, LLM throughput, and every controller action. Thus, each aggregate number in Tables 2 and 3 can be traced back to the moment when foreground FPS dropped, the controller reacted, and LLM throughput changed. Appendix B provides the repeated-run protocol and telemetry audit details.

5 Experimental Setup

Experiments use a 13-inch MacBook Air with Apple M5, a 10-core CPU, an 8-core GPU, and 16 GB unified memory. The background

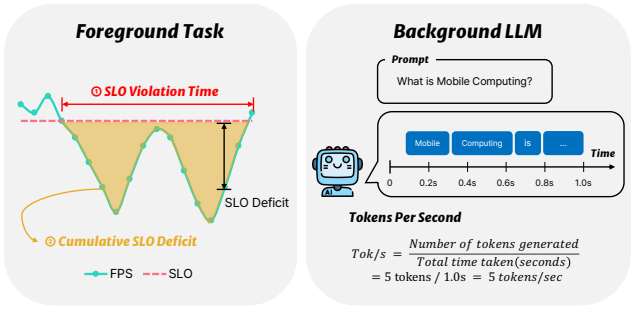


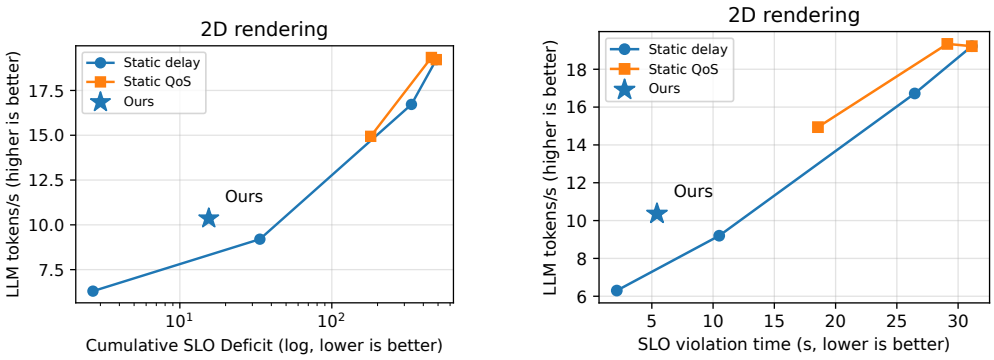
Figure 4: Dual-objective evaluation. Foreground protection is measured by SLO violation time and cumulative SLO deficit, while background usefulness is measured by LLM tokens/s.

model is a 4-bit quantized Qwen3.5-9B model running through llama.cpp with Metal. We use this 9B-class model as a stress workload for upper-end local assistants rather than as a claim about the exact model size used by all platform assistants. Because the device is fanless, it is cooled for two minutes before each run. The prototype supports multiple foreground scenarios, but the main quantitative study focuses on 2D rendering and 3D rendering because they provide continuously sampled FPS and expose repeatable user-visible frame-rate degradation.

We compare against static delay and static QoS baselines. Static delay fixes QoS to Normal and inserts 0, 50, 100, or 150 ms delay between token decode iterations. Static QoS fixes delay to 0 and uses Normal, Utility, or Background QoS. We also evaluate two ablations: Delay Only and QoS Mode Only. Each configuration is repeated five times with the same prompt, model, foreground configuration, and measurement window; the tables report mean ± sample standard deviation across the five runs. We report cumulative SLO deficit and SLO violation time for foreground protection, and LLM tokens/s for background progress.

Table 2: Comparative study on the 2D rendering task. Values are mean ± sample standard deviation over five repeated runs. Static baselines vary either the delay (with QoS fixed to Normal) or the QoS mode (with delay fixed to 0); the (Delay = 0 / QoS Normal) setting is shared by both groups.

Method	Cumulative SLO Deficit ↓	SLO Violation Time ↓	LLM Tokens/s ↑
Static Delay Baselines (QoS Normal)			
Delay = 0	486.8 ± 93.5	31.1 ± 2.4	19.2 ± 0.2
Delay = 50	334.2 ± 32.3	26.5 ± 1.9	16.7 ± 0.0
Delay = 100	33.6 ± 7.5	10.5 ± 2.4	9.2 ± 0.0
Delay = 150	2.7 ± 1.8	2.1 ± 1.1	6.3 ± 0.0
Static QoS Mode Baselines (Delay = 0)			
QoS Normal	486.8 ± 93.5	31.1 ± 2.4	19.2 ± 0.2
QoS Utility	452.2 ± 16.9	29.1 ± 1.2	19.3 ± 0.1
QoS Background	180.2 ± 41.6	18.6 ± 2.4	14.9 ± 0.1
Ours	**15.5** ± 5.7	**5.4** ± 1.4	**10.4** ± 0.2



(a) Cumulative SLO deficit. **(b) SLO violation time.**

Figure 5: 2D rendering trade-off between background LLM throughput and foreground SLO preservation. The left plot uses cumulative SLO deficit, which captures both drop duration and severity; the right plot uses violation time, which captures how long the foreground remains below the SLO.

6 Results and Analysis

6.1 Static policies expose the throughput-SLO trade-off

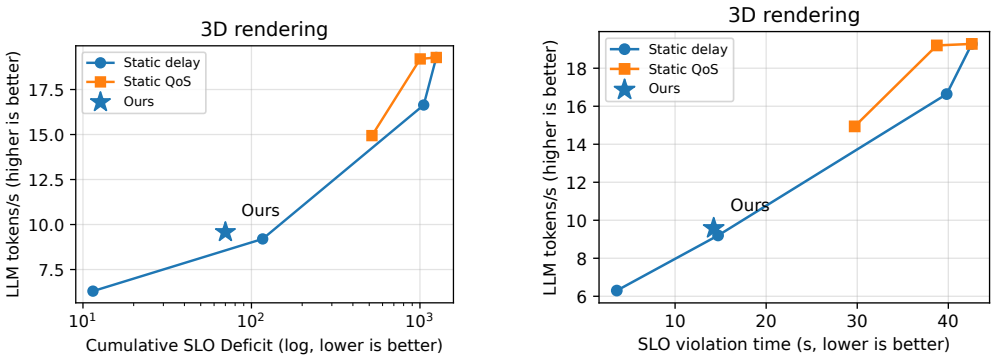
Tables 2 and 3 report the main comparison using mean ± standard deviation over five repeated runs per configuration. With no throttling, the LLM reaches about 19 tokens/s, but foreground SLO debt is severe: 486.8 ± 93.5 cumulative deficit on 2D and 1250.0 ± 140.6 on 3D. This is the failure mode that ordinary LLM throughput benchmarks hide. Static delay reduces deficit, but only by sacrificing throughput. On 2D, a 150 ms delay lowers deficit to 2.7 ± 1.8, but throughput falls to 6.3 ± 0.0 tokens/s. On 3D, the same delay lowers deficit to 11.5 ± 9.3, again at only 6.3 ± 0.0 tokens/s. Static QoS is less effective: Background QoS improves foreground metrics compared with Normal, but still leaves large deficit, especially on 3D. This indicates that CPU priority alone cannot regulate all contention on a unified-memory device.

UbiClaw moves toward the useful knee of the trade-off. On 2D, it reduces cumulative deficit by 96.8% and violation time by 82.6% relative to no throttling, while keeping 10.4 ± 0.2 tokens/s. Compared with static 100 ms delay, UbiClaw has lower deficit (15.5 ± 5.7 vs. 33.6 ± 7.5) and higher throughput (10.4 ± 0.2 vs. 9.2 ± 0.0). Compared with conservative 150 ms delay, it accepts a small amount of extra debt but provides 64.4% higher throughput. On 3D, UbiClaw reduces cumulative deficit by 94.4% and violation time by 66.6% relative to no throttling. It is also slightly better than static 100 ms delay on both deficit and tokens/s. Across the five runs, throughput variance is small while SLO-debt variance reflects bursty foreground interference; the ordering of the main baselines remains consistent.

The 2D result shows why fixed delay is too coarse. Delay=100 is near the useful operating region, but it is still static: it pays the same cost after the foreground has recovered and cannot become more conservative only during harmful intervals. Delay=150 nearly eliminates SLO debt, but over-paces the LLM throughout the whole

Table 3: Comparative study on the 3D rendering task. Values are mean ± sample standard deviation over five repeated runs. Static baselines vary either the delay (with QoS fixed to Normal) or the QoS mode (with delay fixed to 0); the (Delay = 0 / QoS Normal) setting is shared by both groups.

Method	Cumulative SLO Deficit ↓	SLO Violation Time ↓	LLM Tokens/s ↑
Static Delay Baselines (QoS Normal)			
Delay = 0	1250.0 ± 140.6	42.6 ± 1.9	19.3 ± 0.1
Delay = 50	1052.6 ± 68.0	39.8 ± 2.1	16.6 ± 0.1
Delay = 100	116.6 ± 15.0	14.7 ± 1.0	9.2 ± 0.0
Delay = 150	11.5 ± 9.3	3.6 ± 1.5	6.3 ± 0.0
Static QoS Mode Baselines (Delay = 0)			
QoS Normal	1250.0 ± 140.6	42.6 ± 1.9	19.3 ± 0.1
QoS Utility	1003.6 ± 94.1	38.7 ± 3.7	19.2 ± 0.2
QoS Background	518.9 ± 32.4	29.7 ± 2.4	14.9 ± 0.1
Ours	**70.1** ± 18.7	**14.2** ± 2.3	**9.6** ± 0.3



(a) Cumulative SLO deficit. **(b) SLO violation time.**

Figure 6: 3D rendering trade-off between background LLM throughput and foreground SLO preservation. The left plot uses cumulative SLO deficit, which captures both drop duration and severity; the right plot uses violation time, which captures how long the foreground remains below the SLO.

overlap interval. UbiClaw changes delay only when recent SLO observations justify it, then restores constraints when the foreground remains healthy.

The 3D workload is more stressful and makes the same point more strongly. The large gap between Delay=50 and Delay=100 indicates a threshold effect: once rendering is close to saturation, small changes in decode aggressiveness can produce large changes in SLO debt. In this regime, choosing too little delay leaves the foreground harmed, while choosing too much delay wastes LLM progress. The online controller tracks the workload state better than any single fixed setting.

Static QoS also reveals an operating-system lesson. Demoting the LLM to Background QoS improves SLO metrics, but not enough to match decode pacing. This is expected because QoS is primarily a scheduler hint for CPU-side work. It can help foreground threads run sooner, but it does not directly reserve GPU time or memory bandwidth for rendering. The full controller therefore treats QoS as a complementary actuator rather than the sole mechanism.

6.2 Why joint control matters

Table 4 isolates the two actuators. Delay Only is much better than no throttling, confirming that decode pacing reduces GPU/DRAM pressure. However, it still leaves 85.0 ± 5.1 cumulative deficit on 2D and 226.1 ± 41.2 on 3D. QoS Mode Only keeps more LLM throughput but protects the foreground poorly, especially for 3D. The full controller has the best foreground metrics among adaptive policies: it reduces cumulative deficit by 81.8% over Delay Only on 2D and by 69.0% on 3D. The cost is lower tokens/s, but that cost is intentional because the objective is to maximize throughput only after respecting foreground SLOs.

The ablation also validates the bottleneck-aware design. Delay Only is strong when the main bottleneck is GPU or memory bandwidth, but it has no way to communicate background priority to the OS scheduler. QoS Mode Only gives that priority signal, but it cannot directly slow the memory-bound token loop. The full controller combines both: it uses delay as a physical pacing mechanism

Table 4: Ablation study on the 2D and 3D rendering tasks. Values are mean ± sample standard deviation over five repeated runs; Delay Only and QoS Mode Only isolate each actuator, while *Ours* jointly controls both.

Method	2D			3D		
	Cumulative SLO Deficit ↓	SLO Violation Time ↓	LLM Tokens/s ↑	Cumulative SLO Deficit ↓	SLO Violation Time ↓	LLM Tokens/s ↑
No Throttling	486.8 ± 93.5	31.1 ± 2.4	19.2 ± 0.2	1250.0 ± 140.6	42.6 ± 1.9	19.3 ± 0.1
Delay Only	85.0 ± 5.1	21.8 ± 1.5	13.5 ± 0.3	226.1 ± 41.2	33.0 ± 4.3	10.8 ± 0.3
QoS Mode Only	346.9 ± 45.5	29.9 ± 2.7	16.9 ± 0.0	727.1 ± 120.3	38.6 ± 3.9	16.5 ± 0.2
Ours	**15.5** ± 5.7	**5.4** ± 1.4	**10.4** ± 0.2	**70.1** ± 18.7	**14.2** ± 2.3	**9.6** ± 0.3

and QoS as a priority mechanism. This is why the two knobs should not be interpreted as redundant. They affect different layers of the system stack.

6.3 Metric-level interpretation

The results show why the proposed metrics are necessary. If only tokens/s were reported, no throttling would look best; if only foreground preservation were reported, a very conservative static delay would look best. Neither matches the desired behavior of an on-device background assistant. Cumulative SLO deficit and violation time reveal whether a policy is merely fast or actually polite, so UbiClaw treats background throughput as the reward and foreground SLO debt as the guardrail.

7 Discussion and Future Work

UbiClaw is a polite co-scheduling layer rather than strict resource isolation: QoS mainly affects CPU scheduling, and decode delay reduces access frequency rather than reserving GPU or DRAM bandwidth. The controller is also heuristic and FPS-oriented by design. In this prototype, its thresholds, pressure weights, and recovery rules are tuned for one fanless device and two rendering workloads, which keeps the policy simple and interpretable. Five-run averages reduce run-level noise, but the strongest claim within this scope is that foreground-preserving background inference is measurable and controllable on a real on-device system. A natural next step is to collect traces across multiple devices, thermal designs, foreground workloads, and model backends, then use them to auto-tune or learn controller policies that adapt these parameters per device while keeping the foreground SLO as the guardrail. Future work should also add SLO-multiplier sweeps, richer foreground workloads, and SLOs beyond FPS or graphics.

8 Conclusion

UbiClaw reframes local LLM scheduling around the foreground user experience. It defines foreground SLO violation time and cumulative SLO deficit, then uses these metrics to control background LLM decode delay and QoS. The experiments show that dynamic, bottleneck-aware control keeps the LLM useful while keeping the foreground app close to its responsiveness target.

A Controller constants

Table 5 lists the main constants used by the heuristic controller. The values are intentionally simple so that the policy remains interpretable and easy to tune. The constants should be read as a safety policy rather than as fitted model parameters: the FPS-derived SLO remains the primary correctness signal, and the pressure score only guides which actuator to try first.

Table 5: Main controller constants used in the prototype.

Parameter	Value	Role
SLO deadband	≈ 0.03	Near-SLO zone width used to avoid oscillation.
Throttle streak	2 decisions	Consecutive deficit decisions required before throttling.
Recovery streak	4 decisions	Consecutive healthy decisions required before releasing.
Throttle cooldown	0.6 s	Minimum interval between throttle actions.
Recovery cooldown	1.8 s	Conservative interval between restores.
Severe deficit	$\delta \geq 20\%$	Allows simultaneous delay increase and QoS demotion.
Delay ceiling	160 ms	Maximum controller delay before relying more on QoS.
Recovery pressure gate	maxPressure < 0.45	Requires hardware headroom before restoring.

B Measurement protocol and repeated-run aggregation

Each quantitative configuration in the main tables was repeated five times. In each run, the model and prompt were fixed, the in-process backend was preloaded before the measured overlap phase, and the device was cooled before the next run. Reported values are arithmetic means over the run-level CSV summaries. This protocol does not claim cross-device generality, but avoids relying on a single lucky trace and makes comparisons among static delay, static QoS, ablations, and the full controller more stable.

The telemetry stream serves both control and diagnosis. It gives the controller a low-dimensional view of CPU, GPU, process-GPU, and DRAM pressure, while also providing an audit trail for SLO debt. Per-run logs align foreground FPS drops, controller actions, and resulting LLM tokens/s changes on the same time axis. For this reason, we report both SLO violation time and cumulative deficit rather than average FPS alone.

C Controller overhead and telemetry path

The closed-loop controller is intentionally lightweight. Each decision reads a fixed-size observation, updates deficit/healthy streaks and cooldowns, and writes atomic delay or QoS values only when an action is selected. It does not run a second model, launch additional GPU kernels, or modify the Metal execution plan. Since telemetry collection is already part of the logging path, the controller consumes summarized scalar signals instead of re-parsing raw traces. This keeps the controller aligned with its goal: protecting the foreground workload without adding another high-overhead component.

D Controller design rationale

The pressure-score weights are deliberately coarse. GPU utilization and DRAM bandwidth receive the largest weights because local LLM decoding and rendering both stress the unified-memory path. Process GPU time helps distinguish pressure from the experimental process from background system noise. CPU utilization is included with a smaller weight because QoS demotion already directly targets CPU scheduling priority. The score is therefore used as a bottleneck hint, not as a precise FPS predictor.

The temporal constants follow the same principle. Throttling after two consecutive deficit observations filters out single-frame noise while still reacting quickly. Requiring four healthy observations before recovery avoids removing constraints after a brief rebound. LIFO recovery releases the most recent constraint first, smoothly restoring LLM throughput while preserving older safety constraints added under heavier foreground pressure.

E Per-run controller timeline

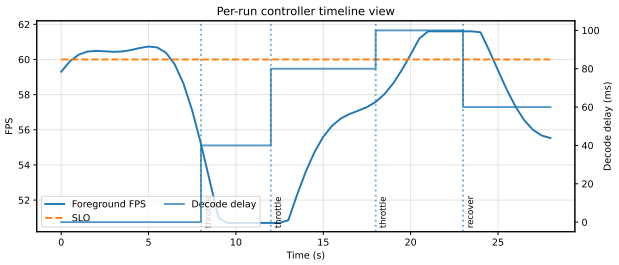


Figure 7: Per-run timeline schematic used to inspect the closed-loop behavior of UbiClaw. The plot aligns foreground FPS, the foreground SLO, controller actions, and decode delay on a single time axis.

F Additional technical schematics

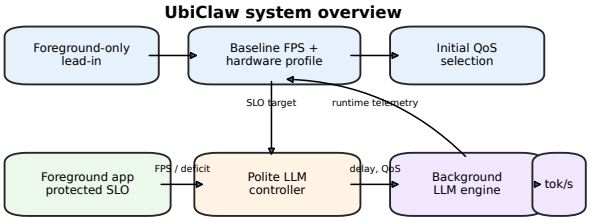


Figure 8: System overview of UbiClaw.

G Telemetry example across phases

This telemetry view provides a sanity check that LLM overlap increases system pressure beyond the foreground-only phase.

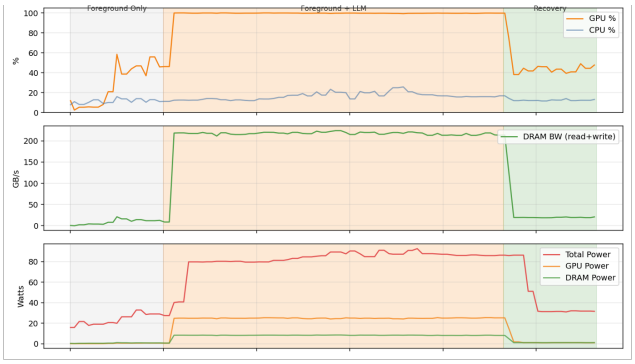


Figure 9: Example hardware telemetry across foreground-only, foreground+LLM, and recovery phases.

H Prototype interface

The prototype UI makes each overlap run auditable by colocating foreground state, controller state, telemetry, and LLM output.

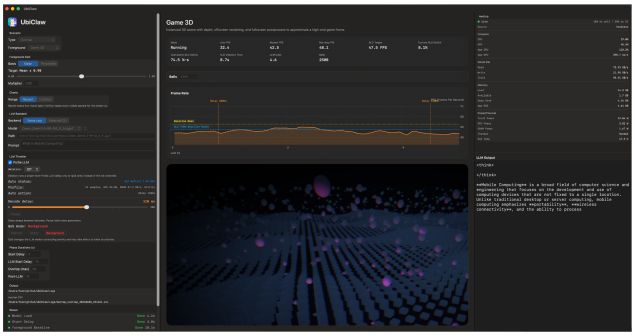


Figure 10: UbiClaw prototype interface during an overlap run.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. arXiv:2403.02310 [cs.DC] https://arxiv.org/abs/2403.02310

[2] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C. Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. LLM in a Flash: Efficient Large Language Model Inference with Limited Memory. arXiv:2312.11514 [cs.CL] https://arxiv.org/abs/2312.11514

[3] Android Developers. 2026. Gemini Nano. https://developer.android.com/ai/gemini-nano.

[4] Apple. 2016. Energy Efficiency Guide for iOS Apps: Prioritize Work with Quality of Service Classes. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/PrioritizeWorkWithQoS.html.

[5] Apple. 2016. Energy Efficiency Guide for Mac Apps: Prioritize Work at the Task Level. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html.

[6] Apple Machine Learning Research. 2024. Apple Intelligence Foundation Language Models. https://machinelearning.apple.com/research/apple-intelligence-foundation-language-models.

[7] Afsara Benazir and Felix Xiaozhu Lin. 2025. Benchmarking and Characterization of Large Language Model Inference on Apple Silicon. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 9, 3 (2025), Article 48. doi:10.1145/3771563

[8] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. 2025. Characterizing Heterogeneous SoCs for Accelerating LLM Inference. arXiv:2501.14794 [cs.DC] https://arxiv.org/abs/2501.14794

[9] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2021. Multi-model Machine Learning Inference Serving with GPU Spatial Partitioning. arXiv:2109.01611 [cs.DC] https://arxiv.org/abs/2109.01611

[10] Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C. Mowry, and Dimitrios Skarlatos. 2025. LithOS: An Operating System for Efficient Machine Learning on GPUs. arXiv:2504.15465 [cs.OS] https://arxiv.org/abs/2504.15465

[11] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80. doi:10.1145/2408776.2408794

[12] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. arXiv:2006.02464 [cs.DC] https://arxiv.org/abs/2006.02464

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:2309.06180 [cs.LG] https://arxiv.org/abs/2309.06180

[14] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. 2024. MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases. arXiv:2402.14905 [cs.CL] https://arxiv.org/abs/2402.14905

[15] Microsoft. 2026. Get started with Phi Silica in the Windows App SDK. https://learn.microsoft.com/en-us/windows/ai/apis/phi-silica.

[16] Chunlin Tian, Xinpeng Qin, Kahou Tam, Li Li, Zijian Wang, Yuanzhe Zhao, Minglei Zhang, and Chengzhong Xu. 2025. CLONE: Customizing LLMs for Efficient Latency-Aware Inference at the Edge. In *Proceedings of the USENIX Annual Technical Conference*. https://arxiv.org/abs/2506.02847

[17] Tom Warren. 2025. Nvidia's AI assistant is here to optimize your gaming PC. *The Verge*. https://www.theverge.com/news/635155/nvidia-g-assist-ai-assistant-available-download

[18] Xinming Wei, Jiahao Zhang, Haoran Li, Jiayu Chen, Haoning Guan, Rui Qu, Maoliang Li, Xiang Chen, and Guojie Luo. 2025. Agent.xpu: Efficient Scheduling of Agentic LLM Workloads on Heterogeneous SoC. arXiv:2506.24045 [cs.DC] https://arxiv.org/abs/2506.24045

[19] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. arXiv:2401.09670 [cs.DC] https://arxiv.org/abs/2401.09670