FLASHRT: AGENT HARNESS FOR GUIDING AGENTS TO DEPLOY REAL-TIME MULTIMODAL APPLICATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Real-time multimodal applications, such as voice agents and interactive video generation, compose heterogeneous models into pipelines whose efficient deployment requires application-specific decisions about placement, streaming, and intra-model parallelism. Existing serving systems and auto-parallelism compilers commit to limited transformations and fixed workload assumptions, so each new application requires a hand-crafted implementation to achieve high performance. We present FLASHRT, an agent harness that guides a generic coding agent to lift a simple developer-written reference implementation into optimized multi-GPU deployments that flexibly weigh target metrics such as latency and throughput. Following a new **chain-of-program** paradigm, FLASHRT directs the agent to convert the reference into an intermediate representation (IR) that captures data dependencies and persistent state, validate the IR with a sequential interpreter, and statically analyze it to identify candidate transformations, which the agent then implements, verifies, and benchmarks in a measurement-gated optimization loop. Across five applications, including video world models and multimodal LLMs, FLASHRT delivers up to $\sim$**70**$\times$ latency reduction and **2.8**$\times$ throughput improvement on NVIDIA B200 GPUs and up to **3.6**$\times$ throughput improvement on AMD MI355X GPUs, and it reduces Qwen3-Omni's response latency by up to **65%** relative to the expert-built vLLM-Omni deployment.

1 INTRODUCTION

As strong generative models emerge across various modalities, there is increasing adoption of real-time multimodal applications (Team et al., 2026; Ball et al., 2025; Défossez et al., 2024; Kodaira et al., 2025), creating demand to serve them efficiently. Compared to extensive prior work on LLM serving (Kwon et al., 2023; Yu et al., 2022; Zheng et al., 2024; Zhong et al., 2024), multimodal applications present a fundamentally different design space: models of various modalities and architectures are composed into complex pipelines, each with its own deployment considerations. For example, an application that prioritizes throughput (e.g., frame rate) may prefer full disaggregation, while a latency-sensitive application may prefer finer-grained intra-component scheduling and co-location. A single system that hosts only a limited set of deployment policies cannot serve all such applications effectively, so each new application requires its own systems effort, and **the steady growth of new and diverse applications makes hand-crafting efficient deployments unscalable**.

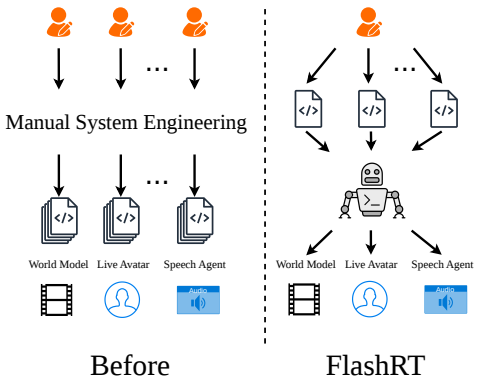


Figure 1: Previously, efficiently serving diverse multimodal applications required manual systems effort per application. FLASHRT solves this by guiding a coding agent to design efficient deployments.

Existing systems try to automate deployment but fall short for three reasons. **(1) Limited deployment strategies:** multimodal serving frameworks such as vLLM-Omni (Yin et al., 2026) and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Cornserve (Chung et al., 2026) provide useful development abstractions but restrict deployments to static policies (e.g., disaggregation at stage boundaries and co-location within a stage). **(2) Limited workload coverage:** auto-parallelism frameworks such as FlexFlow (Jia et al., 2019b), GSPMD (Xu et al., 2021), Alpa (Zheng et al., 2022), and Unity (Unger et al., 2022) automatically choose sharding, placement, and pipelining, but each targets a single fixed workload and does not generalize to diverse multimodal applications. **(3) Optimization granularity:** compiler frameworks such as TVM (Chen et al., 2018) and TASO (Jia et al., 2019a) are highly effective at optimizing deep learning workloads, but they target only operator-level improvements at a fixed granularity that cannot express pipeline-level decisions such as placement or streaming.

An ideal serving system would allow a developer to write an intuitive, unoptimized, single-GPU implementation and automatically obtain deployments specialized to their application's needs. This is difficult to achieve with rule-based systems: the deployment problem is NP-hard (Section 3), so searching for optimal deployments at the lowest kernel-level granularity is prohibitively expensive, and coarser granularities are not even well defined, since the optimal granularity may differ across pipeline components.

Fortunately, AI agents (Yang et al., 2024; Wang et al., 2024) have shown impressive capabilities in synthesizing code from simple specifications and in low-level optimization (Ouyang et al., 2025; Liao et al., 2026; Wei et al., 2025; Shypula et al., 2024). Our key insight is that **coding agents are natively capable of writing efficient end-to-end implementations for complex multimodal pipelines** (Figure 1): **they can employ higher-level reasoning to organize a pipeline at adaptive, heterogeneous granularities, reducing the deployment search space while enabling discovery of highly efficient implementations**. However, we observe that naively prompting an agent to directly transform a baseline implementation into an efficient deployment is not effective. We address this with two design principles:

- **Chain-of-program paradigm**: current agents cannot perform this translation effectively in a single step. Inspired by chain-of-thought prompting (Wei et al., 2022) and by compilers that progressively lower frontend code into a backend, we find that agents are much more effective when they first convert the reference into a structured intermediate representation (IR), analyze it statically, and then translate the IR into effective deployments.
- **Application-grounded validation loop**: agents also tend to explore the search space narrowly and can produce incorrect solutions. We therefore enforce a self-driven loop in which the agent explores, implements, and measures hypotheses using an application-specific test harness that drives simulated user input through the backend, grounding validation and benchmarking in simulated user interactions.

Combining these insights, we introduce FLASHRT, a system that guides a generic coding agent through this structured workflow to convert flexible user implementations into efficient deployments, requiring no human intervention beyond a reference implementation. Across five multimodal applications on NVIDIA B200 and AMD MI355X GPUs, FLASHRT reduces latency by up to $\sim$**70**$\times$ and increases throughput by up to **3.6**$\times$ relative to the references. Its deployments flexibly trade off time-to-first-output (TTFO) latency against throughput and are competitive with or better than expert-designed systems.

2 RELATED WORK

Serving real-time multimodal pipelines. Many new applications compose heterogeneous models into real-time pipelines, including speech-to-speech voice agents (Défossez et al., 2024; Xu et al., 2025a; Seamless Communication et al., 2023), audio-driven avatars (Zhang et al., 2025a; Tian et al., 2024; Xu et al., 2024; Peng et al., 2024; Huang et al., 2025), real-time video world modeling (Kodaira et al., 2025; Ball et al., 2025; Sun et al., 2025; Team et al., 2026), and embodied vision-language-action policies (Black et al., 2026). Multimodal serving systems such as vLLM-Omni (Yin et al., 2026), Cornserve (Chung et al., 2026), and ModServe (Qiu et al., 2026) express such pipelines as graphs of predefined stages, applying rigid inter-stage disaggregation to deploy them; FLASHRT instead derives each deployment from the application's own reference implementation rather than choosing among a fixed set of policies.

Auto-parallelism, compilers, and dataflow IRs. Auto-parallelism systems (Jia et al., 2019b; Xu et al., 2021; Zheng et al., 2022; Unger et al., 2022; Narayanan et al., 2019) and operator-level compilers (Chen et al., 2018; Jia et al., 2019a; Ragan-Kelley et al., 2013) encode their search spaces, including which transformations are legal within them, by hand for a fixed class of workloads at a fixed granularity (Section 1); Appendix B.5 shows analytically that fixed placement and parallelization strategies fall short on the applications we test in Section 5. FLASHRT instead builds on streaming dataflow models such as synchronous dataflow (Lee & Messerschmitt, 1987) and StreamIt (Thies et al., 2002) by using an IR that adds persistent-state annotations from which an agent derives legal transformations at the granularity each component requires.

Coding agents for performance optimization. Coding agents (Yang et al., 2024; Wang et al., 2024; Hong et al., 2024; Shinn et al., 2023) use environment feedback to navigate large codebases and refine their own outputs, and agents have been applied to GPU kernel generation and optimization (Ouyang et al., 2025; Liao et al., 2026; Wei et al., 2025; Lange et al., 2025; Baronio et al., 2025; Li et al., 2025b;a; Zhang et al., 2025b), compiler pass selection (Lin et al., 2025), scientific discovery (Novikov et al., 2025; Romera-Paredes et al., 2024), and reward design (Ma et al., 2023). FLASHRT instead optimizes end-to-end deployments that span multiple processes, GPUs, and model runtimes. Appendix A discusses these lines of work in more detail.

3 PROBLEM FORMULATION

FLASHRT takes as input a synchronous single-GPU reference implementation P_{ref} of a multimodal application, which defines the application semantics and any persistent state (e.g., caches or streaming buffers), and returns a multi-GPU deployment P_{dep} that preserves its behavior. We model the application as a task graph $G = (\mathcal{V}, E)$ whose nodes are computation regions, each instantiated once per input batch j; an edge $(u, v, \lambda) \in E$ denotes an intra-batch data dependence ($\lambda = 0$) of (j, v) on (j, u) or a cross-batch state dependence ($\lambda = 1$) of (j, v) on $(j - 1, u)$, e.g., a KV cache that batch j inherits from batch $j - 1$. A deployment chooses a placement $\rho : \mathcal{V} \to \mathcal{R}$ onto single- or multi-GPU resources and, for each resource r, a non-preemptive, time-indexed schedule $A_r : \mathbb{R}_{\geq 0} \to \{(j, v) : \rho(v) = r\} \cup \{\varnothing\}$ ($\varnothing$ when r is idle) that respects these dependencies under placement-dependent execution and transfer times (Appendix B.1).

The objective is to optimize serving metrics such as response latency or throughput. Latency is governed by the critical path from input to output, whereas sustainable throughput is governed by the most heavily loaded resource; hence co-location shortens the critical path, while disaggregation with pipelining across batches relieves the bottleneck resource, and no single deployment is best for both (Appendices B.1, B.3 and B.4). Finding an optimal deployment is NP-hard even without dependencies or communication costs, since it contains classical multiprocessor scheduling (Lenstra et al., 1977) (Appendix B.2). An efficient deployment can therefore neither be prescribed by a fixed policy nor solved exactly; it must be discovered per application by jointly reasoning about program structure, resource assignment, scheduling, and parallelism.

4 FLASHRT

Unlike rule-based serving systems and auto-parallelism compilers, a coding agent can read an arbitrary reference and produce a deployment specialized to it, but naively tasking an agent with optimizing a reference can fail. Section 4.1 presents a case study of such failure modes, and Sections 4.2 and 4.3 describe how FLASHRT addresses them through hierarchical planning via an IR and a self-driven validation loop, respectively.

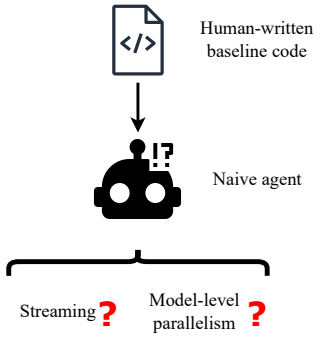
4.1 CASE STUDY

We consider a face-to-face conversational agent that answers a user's spoken query with a talking-head video: an automatic speech recognition (ASR) model transcribes the query, a large language model (LLM) generates a text response, a text-to-speech (TTS) model synthesizes the response audio, and a sound-to-video (S2V) model generates a video of a person speaking it; the audio and video are then synchronized and returned to the user. Figure 2a shows an example reference implementation (the input to FLASHRT), which uses Qwen3-ASR (Shi et al., 2026), Qwen3-4B (Yang et al.,

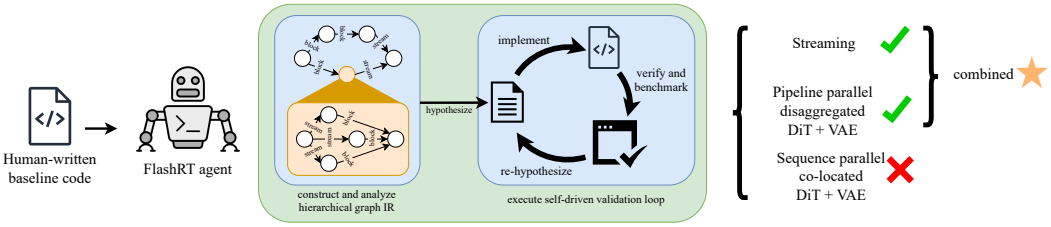
```python
def run_conversational_agent(user_prompt_audio):
    user_prompt_text = run_asr(user_prompt_audio, HF)
    response_text = run_llm(user_prompt_text, vLLM)
    audio_chunks = run_tts(response_text, vLLM_Omni)
    response_audio = concat(audio_chunks)
    response_video = _run_liveavatar_s2v(response_audio)
    return combined(response_audio, response_video)

def _run_liveavatar_s2v(audio):
    video = []
    for idx in range(0, len(audio), MODEL_BATCH_SIZE):
        x = audio[idx : idx + MODEL_BATCH_SIZE]
        for step in range(MODEL_DIFFUSION_STEPS):
            x = denoise(x, step=step)
        video.append(run_vae(x))
    return concat(video)
```

(a) Sequential reference implementation (input).



(b) Naive agent.



(c) FLASHRT agent workflow.

Figure 2: Agent workflows for the face-to-face conversational agent. Given the sequential reference in (a), a naively prompted agent (b) finds only a subset of the optimization axes and never composes them, whereas FLASHRT (c) lifts the reference into an IR for analysis and then runs a self-driven validation loop that tests and composes diverse strategies for the target metric.

2025), Qwen3-TTS (Hu et al., 2026), and LiveAvatar (Huang et al., 2025). Suppose the response latency budget is flexible and the agent must instead optimize throughput (i.e., frame rate), which favors inter-module streaming and pipeline parallelism. We manually identify two optimization axes:

- *Application level*: TTS produces audio in chunks and the autoregressive LiveAvatar model consumes audio in chunks, so TTS can be streamed into S2V; likewise, generated chunks can be streamed into a display buffer while later chunks are still being processed.
- *Model level*: LiveAvatar consists of a diffusion transformer (DiT) and a variational autoencoder (VAE) that share no internal state (e.g., KV cache), so the DiT can process one batch while the VAE decodes the previous batch on a separate GPU. Additionally, LiveAvatar trains each diffusion step to use an independent KV cache, so the steps can be deployed with pipeline parallelism (PP), one GPU per step. Appendices B.3 and B.4 formally analyze the latency–throughput implications of these two choices.

When we provide the agent with a synchronous, single-GPU reference and naively instruct it to find a more efficient deployment, we observe two failure modes (Figure 2b):

- **The agent misses model-level parallelism opportunities, so it fails to address low throughput.** Recognizing that the diffusion steps and the VAE can be pipelined requires reasoning about which parts of the pipeline share persistent state, and an agent that reads the reference as ordinary code does not necessarily carry out this analysis on its own.
- **The agent explores narrow optimization patterns and terminates at simple optimizations.** It streams the generated video chunks into the output buffer but overlooks that TTS can be streamed into S2V for the same reason, and it never combines strategies into a single deployment.

Sections 4.2 and 4.3 describe how FLASHRT eliminates these failures without human intervention (Figure 2c), and Section 5.5 shows that removing either of FLASHRT's key components reproduces them.

4.2 CHAIN-OF-PROGRAM PARADIGM FOR HIERARCHICAL PLANNING

The first failure mode stems from **expecting the agent to transform a baseline implementation into an effective deployment in one monolithic step**. In contrast, human developers reason about end-to-end workflows at varying levels of granularity before designing an implementation. This is analogous to chain-of-thought (Wei et al., 2022), which lets language models reason about a prompt before responding; it is also analogous to intermediate representations used by conventional compilers, which progressively lower a frontend implementation into a backend over multiple simpler passes. We therefore instruct the agent to first convert the baseline into a structured intermediate representation (IR) and only then convert the IR into a set of candidate deployments, which makes it far more robust at discovering all axes for efficient implementation. We call this paradigm **chain-of-program**.

IR design. The IR represents a pipeline as one or more hierarchical directed acyclic graphs whose nodes encapsulate self-contained operations and whose edges denote data dependencies. These graphs instantiate the task graph $G = (\mathcal{V}, E)$ of Section 3: dataflow edges are the intra-batch dependencies, and the node annotations below supply the cross-batch ones. The IR exposes three structural properties that drive the subsequent analysis:

- **Graph hierarchy**: the agent builds a top-level graph of the application workflow and nested graphs for the internals of each module whose implementation is exposed, while opaque third-party serving systems remain single nodes, so the reference can freely compose custom model implementations with such systems. The hierarchy also forces the agent to consider multiple levels of granularity: exposing a module's internal stages as nodes allows a deployment to place or parallelize them separately, which a flat, module-level graph would preclude.
- **Node-level state annotation**: each node declares the persistent state it reads and writes (e.g., a KV cache), excluding transient internal state. Nodes that touch the same persistent state are joined by the cross-batch ($\lambda = 1$) edges (Section 3) and must be ordered across batches, whereas nodes with disjoint state can be disaggregated and executed in parallel, which reveals opportunities for disaggregation and pipeline parallelism.
- **Edge-level streaming annotation**: each data edge is marked as blocking or streaming; in the case study, for example, the ASR $\rightarrow$ LLM edge is blocking while the TTS $\rightarrow$ S2V edge is streaming. Streaming edges mark where a consumer can begin before its producer finishes, so the two can overlap on separate resources, whereas blocking edges serialize and therefore favor co-location.

Agentic IR analysis. The agent is further instructed to validate and analyze its IR using tools we provide. First, an IR interpreter executes the IR sequentially in topological order, so the agent can run the reference and its IR on the same sample inputs and confirm that the IR is faithful. Then, IR analysis tools derive, from the graph edges and shared persistent state, which operators are independent across batches, a partition of operators into pipeline stages, the streaming overlaps, and the critical path. These tools make the agent's analysis more reliable and deterministic in surfacing the deployment strategies it can try to implement.

4.3 APPLICATION-GROUNDED VALIDATION LOOP

Even with a well-defined candidate space surfaced by IR analysis, an unguided agent may still focus on a single optimization axis (the second failure mode in Section 4.1) or settle for an inefficient implementation of a strategy it has identified. Work on agentic kernel generation (Ouyang et al., 2025; Liao et al., 2026; Wei et al., 2025) and local code optimization (Shypula et al., 2024) introduces self-driven loops in which the agent continuously iterates on a solution, verifies correctness, and benchmarks performance. For multimodal pipelines, however, correctness and performance are highly application-specific, so in FLASHRT the agent first designs a test harness grounded in the user experience and then enters a self-driven validation loop during which it uses the harness to measure the correctness and performance of each candidate deployment.

Verification and benchmarking. Although the agent cannot interact with a backend through its user-facing frontend, it can implement a test harness that writes simulated input to the backend's input buffers and reads the corresponding output from its output buffers, just as a user-facing frontend would. Driving the same simulated inputs through the reference and the agent's backend validates

correctness, and timing the buffer writes and reads enables measuring application-specific latency and throughput metrics, grounding the agent's hypotheses in measurements that mimic real user interactions.

Iteration step. The loop is driven by a self-evolving variant queue initialized from the IR analyses. Each iteration begins with a hypothesis that names the analysis result justifying the transformation's legality and the bottleneck it targets. The agent implements the variant in an isolated environment, verifies element-wise output equivalence against the reference on sample inputs (debugging as needed), and benchmarks it; it then updates the queue, appending any new variants the outcome suggests and re-ranking pending ones by expected impact. The loop terminates only when every queued variant has been measured or removed for a documented reason, effectively requiring the agent to validate every hypothesis and to consider both diverse strategies and their compositions.

5 EXPERIMENTS

We evaluate FLASHRT on a diverse set of real-time multimodal applications to answer five questions: **(1)** Can the agent generate highly efficient implementations that balance high throughput and low time-to-first-output (TTFO) latency? **(2)** Does the framework generalize to diverse applications? **(3)** Does the framework generalize across hardware? **(4)** Which parts of the harness are responsible for its effectiveness? **(5)** How robust is the result to the agent configuration and to run-to-run variation? We organize the experiments as follows:

- Section 5.2 presents a case study on the application from Section 4.1 of how the agent composes strategies to achieve $\sim$**70**$\times$ latency reduction from the baseline and high theoretical frame rate.
- Section 5.3 evaluates the agent on four additional applications, each with a different compute structure. The agent delivers principled deployments that trade off latency and throughput.
- Section 5.4 reruns the agent on an AMD 8$\times$ MI355X node for the same applications and finds that it recovers the same class of efficient deployments, demonstrating hardware generalization.
- Section 5.5 ablates the two harness components and finds that each is necessary.
- Section 5.6 measures how FLASHRT behaves under a different agent configuration and across independent runs.

5.1 SETUP

Unless otherwise stated, experiments run on a single NVIDIA 8$\times$ B200 node (Section 5.4 additionally uses an AMD 8$\times$ MI355X node). We use Anthropic Claude Code (Anthropic, 2025) running Claude Opus 4.8 (Anthropic, 2026) at maximum reasoning effort. Each session runs unattended and starts from scratch, with no memory of previous runs and no visibility into runs on other applications, and its starting prompt specifies only the application and the available GPU resources, with no hints about deployment strategies. Each application is supplied with a synchronous, single-GPU reference implementation that performs no inter-component streaming, so the agent must discover all deployment patterns on its own.

5.2 CASE STUDY: FACE-TO-FACE CONVERSATIONAL AGENT

We first evaluate FLASHRT on the face-to-face conversational agent from Section 4.1, starting from a sequential reference structured like Figure 2a. Table 1 reports the results, and Table 3 (Appendix C.1) shows the placements:

- With two GPUs, FLASHRT reduces TTFO latency by $\sim$**70**$\times$. The agent starts S2V inference on TTS audio chunks as soon as they become available and streams each generated video chunk to the output buffer while later chunks are still being produced, so the first frame is emitted long before the response is fully rendered. It co-locates the ASR with the S2V DiT and VAE, and the LLM with the TTS: these components never execute concurrently, so disaggregating them would bring no additional benefit.
- To raise frame rate, the agent expands to six GPUs and pipelines the S2V model across its denoising steps, the strategy identified in Section 4.1, which sustains at least the frame rate of sequence parallelism (SP) when SP speedup is sublinear (Appendix B.4). This improves frame rate by **4.3**$\times$ over the 2-GPU deployment with minimal impact on latency.

Table 1: Deployments generated by FLASHRT on NVIDIA B200 GPUs versus each application's sequential baseline and, for Qwen3-Omni, the hand-engineered vLLM-Omni deployment. Latency is time-to-first-output. The LiveAvatar baseline generates its full video offline and thus has no frame rate; its FLASHRT frame rates are theoretical sustainable rates, which can exceed the frame rate of the S2V model's training videos. For Qwen3-Omni, throughput is the real-time factor (RTF), the ratio of generation time to audio length. Placements are in Appendix C.1.

Application	Deployment	# GPUs	Latency (s) ↓	Frame rate (FPS) ↑
LiveAvatar	Baseline (sequential)	1	117.10	–
	FLASHRT (streaming)	2	**1.69**	30.92
	FLASHRT (streaming + S2V PP)	6	1.72	**133.48**
			Latency (s) ↓	**RTF $<$ 1**
Qwen3-Omni	Baseline (sequential)	1	42.713	✓
	vLLM-Omni	3	0.433	✓
	FLASHRT	3	**0.323**	✓
			Latency (s) ↓	**Frame rate (FPS) ↑**
Krea + SAM 3	Baseline (sequential)	1	1.715	6.82
	FLASHRT (parallel)	2	1.014	11.54
	FLASHRT (latency-optimized)	4	**0.491**	17.18
	FLASHRT (frame-rate-optimized)	5	0.517	**19.41**
WorldPlay	Baseline (sequential)	1	0.869	20.4
	FLASHRT (latency-optimized)	2	**0.493**	25.5
	FLASHRT (frame-rate-optimized)	2	0.625	**31.0**
LongLive	Baseline (sequential)	1	0.674	25.8
	FLASHRT (latency-optimized)	2	**0.512**	36.5
	FLASHRT (frame-rate-optimized)	2	0.630	**47.8**

5.3 GENERALIZATION TO OTHER APPLICATIONS

We evaluate four additional applications with different compute structures; Table 1 reports their results, Appendix C.1 their placements, and Appendix D an independent correctness check against the references.

Multimodal LLM (Qwen3-Omni). Qwen3-Omni (Xu et al., 2025b) responds to a prompt through a Thinker LLM that produces a text response, a Talker LLM that converts it into audio codec tokens, and a Vocoder that synthesizes audio from the tokens. vLLM-Omni (Yin et al., 2026) explicitly implements disaggregation and inter-stage streaming for this model. Given a reference that serves each component independently and coordinates them synchronously, the FLASHRT agent discovers the disaggregation structure and streaming logic on its own and achieves **25%** lower latency than vLLM-Omni by using lighter-weight inter-component data transfer than vLLM-Omni's more general implementation, while keeping the RTF below 1 so that its audio output remains real-time.

Video background editor (Krea-Realtime + SAM 3). The user streams webcam video that Krea-Realtime (Millon, 2025), an autoregressive video-to-video model, restyles, while SAM 3 (Carion et al., 2025) segments the user's body in each frame; the composed output keeps the original pixels on the body, so only the background is restyled. The two paths are independent until the final composition, and the agent automatically places SAM 3 on a separate GPU from the video model. Its latency-optimized deployment co-locates the Krea-Realtime DiT and VAE with SP-3, reducing latency by **3.5**×, while its frame-rate-optimized deployment keeps the DiT at SP-3 and places the VAE on a separate GPU to pipeline the two, raising frame rate by **2.8**×.

Video world model (WorldPlay). The user issues keyboard actions (WASD or arrow keys), and HY-WorldPlay 5B (Sun et al., 2025), an autoregressive DiT followed by a streaming VAE, renders them as movement in a generated video stream. Since the DiT and VAE share no persistent state, they can be disaggregated onto separate GPUs and pipelined across batches, or co-located and

Table 2: Ablation of the two components of the FLASHRT workflow on the face-to-face conversational agent. Each configuration receives the same starting prompt, 8-GPU budget, and requirement to verify correctness; we report the best deployment each produces.

Configuration	IR pass	Loop	Best deployment	# GPUs	Latency (s) ↓	Frame rate (FPS) ↑
Reference	–	–	Sequential baseline	1	117.10	–
Naive	×	×	S2V output streaming	2	10.45	30.99
No IR	×	✓	Full streaming	2	1.68	30.67
No Loop	✓	×	Full streaming + S2V PP	6	2.37	44.69
FLASHRT	✓	✓	Full streaming + S2V PP	6	**1.72**	**133.48**

sequence-parallelized on a shared group. The agent uses both: for higher frame rate, it disaggregates the DiT and VAE so that they execute in parallel on consecutive batches; for lower latency, it co-locates them, which affords a higher SP degree within the same GPU budget and shortens the critical path (here also raising frame rate). At 2 GPUs, co-location reaches the lower latency (0.493 s) and disaggregation the higher frame rate (31.0 FPS), as our analysis of this pipeline predicts (Appendix B.3). Scaling these strategies to larger GPU budgets reaches up to **51%** lower latency and **2.2**× the baseline frame rate (Appendix C.2).

Video narrator (LongLive). The user speaks prompts that an ASR model transcribes, and LongLive-2.0-5B (Chen et al., 2026), also an autoregressive DiT with a streaming VAE, renders a continuous video stream that reflects each new prompt. The ASR shares no persistent state with the video model and supplies a new prompt only when the user speaks, so it can run asynchronously on a separate GPU. As for WorldPlay, the agent offers a co-located, latency-optimized deployment and a disaggregated, frame-rate-optimized one; at 2 GPUs, the latter trades **23%** higher latency for **31%** higher frame rate. At larger GPU budgets, the agent reaches **1.6**× lower latency and **2.6**× the baseline frame rate (Appendix C.3).

5.4 GENERALIZATION ACROSS HARDWARE

To test whether FLASHRT generalizes across accelerators, we separately execute the full agent pipeline on an AMD 8× MI355X node, with the same reference implementations and GPU budgets, no hardware-specific hints, and no information about the B200 runs. FLASHRT reduces latency by up to ∼**70**× and improves throughput by up to **3.6**×, while beating the vLLM-Omni deployment of Qwen3-Omni by **65%** with RTF below 1 (all deployments are listed in Appendix C.4). Across all five applications, the agent recovers the deployment families it found on B200 and the same tradeoff between them: co-location minimizes latency, while disaggregation raises frame rate.

The agent also adapts its choices to the hardware. MI355X's high-degree co-location is strong enough that WorldPlay and LongLive reach *lower* best latency than on B200 (0.320 s and 0.343 s, respectively). On the conversational agent, since per-step pipelining admits only four S2V stages, the agent instead merges the denoising steps into two pipeline stages with SP-3 each, which uses the full 8-GPU budget and, under matched streaming, sustains roughly 30% higher frame rate than per-step pipelining at six GPUs. On the video background editor, the pipeline is bound by the SAM 3 segmentation path, so disaggregating the VAE yields little additional frame rate.

5.5 ABLATION STUDY

We now identify which components of the harness are responsible for these results, using the face-to-face conversational agent on an 8× B200 node, since it admits the richest combination of strategies in our set. We remove each of the two workflow components: *Naive* removes both and generically instructs the agent to develop optimized deployments, *No IR* keeps only the validation loop, and *No Loop* keeps only the IR pass. Table 2 reports the best deployment of each configuration, and Appendix E.1 details all variants each configuration measured. Removing each component fails in a different way, matching the two failure modes described in Section 4.1:

- **Without the IR pass, the agent does not identify the pipeline-parallel axis at all.** *Naive* finds only that generated video can be streamed to the output buffer. Adding the loop (*No IR*) keeps the

agent exploring after its first success, so it also streams the TTS into the S2V model and attains the same $\sim$70$\times$ latency reduction as the full system, but neither configuration proposes pipeline parallelism, so both remain near 31 FPS. Pipeline parallelism is discovered only when the IR annotates the per-step KV caches as disjoint persistent state, which makes the denoising steps visibly independent.

- **Without the loop, the agent surfaces the pipeline-parallel axis but does not turn it into an efficient implementation.** *No Loop* finds both full streaming and S2V pipeline parallelism during IR analysis and directly implements a combined variant, but it orchestrates the four steps within a single process so that every synchronization event blocks all steps; it terminates at this prototype with 44.69 FPS. FLASHRT instead isolates pipeline parallelism as its own variant, iterating to a multi-process implementation with non-blocking NCCL transfers between steps, and then composes it with streaming, reaching **3.0**$\times$ the frame rate of *No Loop* at **1.4**$\times$ lower latency.

The two components are therefore complementary: the IR pass determines which optimization axes the agent targets, and the loop determines how well it exploits them and composes them into optimized deployments.

5.6 ROBUSTNESS

The deployments reported so far come from a single agent configuration, with only a single run per application. To measure the robustness of FLASHRT, we conduct two additional studies to evaluate a modified agent configuration and reliability across independent runs, respectively. Both studies below rerun the face-to-face conversational agent from Section 5.2 on the 8$\times$ B200 node (details in Appendices E.2 and E.3).

Sensitivity to the underlying model. We conduct an independent trial using OpenAI Codex (OpenAI, 2025) with GPT-5.6 Sol (OpenAI, 2026) under the same prompt and 8-GPU budget. The agent discovers both key optimizations, full streaming and S2V pipeline parallelism, and reaches essentially the same latency and frame rate (1.77 s and 131.98 FPS, versus 1.72 s and 133.48 FPS; Table 13a). It additionally shards the VAE with sequence parallelism and thus uses 8 GPUs instead of 6, which changes neither metric since frame rate is capped by the denoising steps, each already slower than the single-rank VAE. The design principles of FLASHRT are thus not tailored to one coding agent or model.

Consistency across runs. We run the configuration from Section 5.2 three times independently, including the original run. All converge on full streaming composed with S2V pipeline parallelism, within 1.63–1.78 s of latency and 125.30–133.48 FPS (Table 13b), and all follow the same exploration path: a streaming-only variant, then a pipeline-parallel-only variant, then their composition. The runs differ only in how they place the ASR, LLM, TTS, and VAE, which has little effect since these components never execute concurrently.

6 CONCLUSION

We present FLASHRT, an agent harness for using a coding agent to flexibly deploy real-time multimodal applications. We show that a properly structured workflow, consisting of an IR conversion step that makes data dependencies and streaming behavior explicit and a self-driven validation loop that encourages diverse and principled exploration, enables agents to convert simple, human-authored implementations into highly efficient deployments that optimize flexible targets such as latency or throughput, significantly outperform the handwritten baselines, and can match or exceed solutions from other serving systems. FLASHRT is also inexpensive: running the agent for each application costs \$75–\$167 in API usage and a few hours of agent time (Appendix F), comparable to or below the expert engineering effort it replaces, so deployment effort scales with compute rather than with the availability of systems experts.

A current limitation is that FLASHRT does not yet integrate LLM kernel optimization agents, which we leave as future work to broaden the range of optimization opportunities. Moreover, although the harness transfers across the agent configurations we tested, we do not claim that FLASHRT is model-agnostic: it assumes a model capable of organizing a hierarchical IR and maintaining a hypothesis queue, and studying weaker or open-source models with modified scaffolding remains future work.

AI USE STATEMENT

Coding agents are fundamentally the subject of this work, as every deployment evaluated in Section 5 was generated by agents as part of our method. Separately, we used generative AI tools to assist with writing the proofs in Appendix B (developing theoretical models and formulating and proving mathematical claims) as well as implementing the backbone of the FLASHRT framework. Generating synthetic datasets, translation, and cleaning and formatting datasets are not applicable to this work, and we have not used generative AI tools for any of the other tasks that require disclosure. Additionally, we used generative AI tools for helping create and edit software code, summarize and analyze existing literature, and edit the paper to improve readability. We have reviewed all AI-assisted work; for example, we verified the generated code for correctness and manually reviewed every LLM-generated paper modification. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We include in the supplementary material the FLASHRT harness (agent instructions and IR framework), the reference implementations, the agent-generated deployments, and the independent test harness. For any mathematical claims, all proofs are provided in Appendix B.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824076. URL https://doi.org/10.14778/2824032.2824076.

Anthropic. Claude code: an agentic coding tool. https://github.com/anthropics/claude-code, 2025.

Anthropic. Introducing claude opus 4.8. https://www.anthropic.com/news/claude-opus-4-8, 2026.

Philip J. Ball, Jakob Bauer, Frank Belletti, Bethanie Brownfield, Ariel Ephrat, Shlomi Fruchter, Agrim Gupta, Kristian Holsheimer, Aleksander Holynski, Jiri Hron, Christos Kaplanis, Marjorie Limont, Matt McGill, Yanko Oliveira, Jack Parker-Holder, Frank Perbet, Guy Scully, Jeremy Shar, Stephen Spencer, Omer Tov, Ruben Villegas, Emma Wang, Jessica Yung, Cip Baetu, Jordi Berbel, David Bridson, Jake Bruce, Gavin Buttimore, Sarah Chakera, Bilva Chandra, Paul Collins, Alex Cullum, Bogdan Damoc, Vibha Dasagi, Maxime Gazeau, Charles Gbadamosi, Woohyun Han, Ed Hirst, Ashyana Kachra, Lucie Kerley, Kristian Kjems, Eva Knoepfel, Vika Koriakin, Jessica Lo, Cong Lu, Zeb Mehring, Alex Moufarek, Henna Nandwani, Valeria Oliveira, Fabio Pardo, Jane Park, Andrew Pierson, Ben Poole, Helen Ran, Tim Salimans, Manuel Sanchez, Igor Saprykin, Amy Shen, Sailesh Sidhwani, Duncan Smith, Joe Stanton, Hamish Tomlinson, Dimple Vijaykumar, Luyu Wang, Piers Wingfield, Nat Wong, Keyang Xu, Christopher Yew, Nick Young, Vadim Zubov, Douglas Eck, Dumitru Erhan, Koray Kavukcuoglu, Demis Hassabis, Zoubin Gharamani, Raia Hadsell, Aäron van den Oord, Inbar Mosseri, Adrian Bolton, Satinder Singh, and Tim Rocktäschel. Genie 3: A new frontier for world models. 2025.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL https://arxiv.org/abs/2507.11948.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke,

Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A vision-language-action flow model for general robot control, 2026. URL https://arxiv.org/abs/2410.24164.

Nicolas Carion, Laura Gustafson, Yuan-Ting Hu, Shoubhik Debnath, Ronghang Hu, Didac Suris, Chaitanya Ryali, Kalyan Vasudev Alwala, Haitham Khedr, Andrew Huang, Jie Lei, Tengyu Ma, Baishan Guo, Arpit Kalla, Markus Marks, Joseph Greer, Meng Wang, Peize Sun, Roman Rädle, Triantafyllos Afouras, Effrosyni Mavroudi, Katherine Xu, Tsung-Han Wu, Yu Zhou, Liliane Momeni, Rishi Hazra, Shuangrui Ding, Sagar Vaze, Francois Porcher, Feng Li, Siyuan Li, Aishwarya Kamath, Ho Kei Cheng, Piotr Dollár, Nikhila Ravi, Kate Saenko, Pengchuan Zhang, and Christoph Feichtenhofer. Sam 3: Segment anything with concepts, 2025. URL https://arxiv.org/abs/2511.16719.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pp. 579–594, USA, 2018. USENIX Association. ISBN 9781931971478.

Yukang Chen, Luozhou Wang, Wei Huang, Shuai Yang, Bohan Zhang, Yicheng Xiao, Ruihang Chu, Weian Mao, Qixin Hu, Shaoteng Liu, Yuyang Zhao, Huizi Mao, Ying-Cong Chen, Enze Xie, Xiaojuan Qi, and Song Han. Longlive2.0: An nvfp4 parallel infrastructure for long video generation. *arXiv preprint arXiv: 2605.18739*, 2026.

Jae-Won Chung, Jeff J. Ma, Jisang Ahn, Yizhuo Liang, Akshay Jajoo, Myungjin Lee, and Mosharaf Chowdhury. Cornserve: A distributed serving system for any-to-any multimodal models. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, pp. 1316–1321, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400724152. doi: 10.1145/3786335.3813209. URL https://doi.org/10.1145/3786335.3813209.

Alexandre Défossez, Laurent Mazaré, Manu Orsini, Amélie Royer, Patrick Pérez, Hervé Jégou, Edouard Grave, and Neil Zeghidour. Moshi: a speech-text foundation model for real-time dialogue. Technical report, 2024. URL https://arxiv.org/abs/2410.00037.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Hangrui Hu, Xinfa Zhu, Ting He, Dake Guo, Bin Zhang, Xiong Wang, Zhifang Guo, Ziyue Jiang, Hongkun Hao, Zishan Guo, Xinyu Zhang, Pei Zhang, Baosong Yang, Jin Xu, Jingren Zhou, and Junyang Lin. Qwen3-tts technical report, 2026. URL https://arxiv.org/abs/2601.15621.

Yubo Huang, Hailong Guo, Fangtai Wu, Shifeng Zhang, Shijie Huang, Qijun Gan, Lin Liu, Sirui Zhao, Enhong Chen, Jiaming Liu, and Steven Hoi. Live avatar: Streaming real-time audio-driven avatar generation with infinite length, 2025. URL https://arxiv.org/abs/2512.04677.

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 47–62, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359630. URL https://doi.org/10.1145/3341301.3359630.

Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 1–13, 2019b. URL https://proceedings.mlsys.org/paper_files/paper/2019/file/b422680f3db0986ddd7f8f126baaf0fa-Paper.pdf.

Akio Kodaira, Chenfeng Xu, Toshiki Hazama, Takanori Yoshimoto, Kohei Ohno, Shogo Mitsuhori, Soichi Sugano, Hanying Cho, Zhijian Liu, Masayoshi Tomizuka, and Kurt Keutzer. Streamdiffusion: A pipeline-level solution for real-time interactive generation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 12371–12380, October 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization, 2025. URL https://arxiv.org/abs/2509.14279.

E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. doi: 10.1109/PROC.1987.13876.

J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. In P.L. Hammer, E.L. Johnson, B.H. Korte, and G.L. Nemhauser (eds.), *Studies in Integer Programming*, volume 1 of *Annals of Discrete Mathematics*, pp. 343–362. Elsevier, 1977. doi: https://doi.org/10.1016/S0167-5060(08)70743-X. URL https://www.sciencedirect.com/science/article/pii/S016750600870743X.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025a.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Shum Chris. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025b.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, Zewei Jiang, Dianshi Li, Uladzimir Pashkevich, Varna Puvvada, Feng Shi, Matt Steiner, Ruichao Xiao, Liyuan Li, Nathan Yan, Xiayu Yu, Zhou Fang, Roman Levenstein, Kunming Ho, Haishan Zhu, Alec Hammond, Richard Li, Ajit Mathews, Kaustubh Gondkar, Abdul Zainul-Abedin, Ketan Singh, Hongtao Yu, Wenyuan Chi, Barney Huang, Sean Zhang, Noah Weller, Zach Marine, Wyatt Cook, Carole-Jean Wu, and Gaoxiang Liu. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta, 2026. URL https://arxiv.org/abs/2512.23236.

Hongyu Lin, Haolin Pan, Haoran Luo, Yuchen Li, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Awarecompiler: Agentic context-aware compiler optimization via a synergistic knowledge-data driven framework, 2025. URL https://arxiv.org/abs/2510.11759.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv: Arxiv-2310.12931*, 2023.

Erwann Millon. Krea realtime 14b: Real-time video generation, 2025. URL https://github.com/krea-ai/realtime-video.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: a distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pp. 561–577, USA, 2018. USENIX Association. ISBN 9781931971478.

Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 439–455, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522738. URL https://doi.org/10.1145/2517349.2522738.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI. Codex cli: Lightweight coding agent that runs in your terminal. https://github.com/openai/codex, 2025.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: can llms write efficient gpu kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ISCA '24, pp. 118–132. IEEE Press, 2025. ISBN 9798350326581. doi: 10.1109/ISCA59077.2024.00019. URL https://doi.org/10.1109/ISCA59077.2024.00019.

Ziqiao Peng, Wentao Hu, Yue Shi, Xiangyu Zhu, Xiaomei Zhang, Hao Zhao, Jun He, Hongyan Liu, and Zhaoxin Fan. Synctalk: The devil is in the synchronization for talking head synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 666–676, 2024.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ram Ramjee, and Rodrigo Fonseca. Modserve: Modality- and stage-aware resource disaggregation for scalable multimodal model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, SoCC '25, pp. 817–830, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722769. doi: 10.1145/3772052.3772254. URL https://doi.org/10.1145/3772052.3772254.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL https://doi.org/10.1145/2499370.2462176.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Seamless Communication, Loïc Barrault, Yu-An Chung, Mariano Coria Meglioli, David Dale, Ning Dong, Mark Duppenthaler, Paul-Ambroise Duquenne, Brian Ellis, Hady Elsahar, Justin Haaheim, John Hoffman, Min-Jae Hwang, Hirofumi Inaguma, Christopher Klaiber, Ilia Kulikov, Pengwei

Li, Daniel Licht, Jean Maillard, Ruslan Mavlyutov, Alice Rakotoarison, Kaushik Ram Sadagopan, Abinesh Ramakrishnan, Tuan Tran, Guillaume Wenzek, Yilin Yang, Ethan Ye, Ivan Evtimov, Pierre Fernandez, Cynthia Gao, Prangthip Hansanti, Elahe Kalbassi, Amanda Kallet, Artyom Kozhevnikov, Gabriel Mejia, Robin San Roman, Christophe Touret, Corinne Wong, Carleigh Wood, Bokai Yu, Pierre Andrews, Can Balioglu, Peng-Jen Chen, Marta R. Costa-jussà, Maha Elbayad, Hongyu Gong, Francisco Guzmán, Kevin Heffernan, Somya Jain, Justine Kao, Ann Lee, Xutai Ma, Alex Mourachko, Benjamin Peloquin, Juan Pino, Sravya Popuri, Christophe Ropers, Safiyyah Saleem, Holger Schwenk, Anna Sun, Paden Tomasello, Changhan Wang, Jeff Wang, Skyler Wang, and Mary Williamson. Seamless: Multilingual expressive and streaming speech translation. 2023.

Xian Shi, Xiong Wang, Zhifang Guo, Yongqi Wang, Pei Zhang, Xinyu Zhang, Zishan Guo, Hongkun Hao, Yu Xi, Baosong Yang, Jin Xu, Jingren Zhou, and Junyang Lin. Qwen3-asr technical report, 2026. URL https://arxiv.org/abs/2601.21337.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ix7rLVHXyY.

Wenqiang Sun, Haiyu Zhang, Haoyuan Wang, Junta Wu, Zehang Wang, Zhenwei Wang, Yunhong Wang, Jun Zhang, Tengfei Wang, and Chunchao Guo. Worldplay: Towards long-term geometric consistency for real-time interactive world modeling. *arXiv preprint arXiv:2512.14614*, 2025.

Robbyant Team, Zelin Gao, Qiuyu Wang, Yanhong Zeng, Jiapeng Zhu, Ka Leong Cheng, Yixuan Li, Hanlin Wang, Yinghao Xu, Shuailei Ma, Yihang Chen, Jie Liu, Yansong Cheng, Yao Yao, Jiayi Zhu, Yihao Meng, Kecheng Zheng, Qingyan Bai, Jingye Chen, Zehong Shen, Yue Yu, Xing Zhu, Yujun Shen, and Hao Ouyang. Advancing open-source world models. *arXiv preprint arXiv:2601.20540*, 2026.

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pp. 179–196, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540433694.

Linrui Tian, Qi Wang, Bang Zhang, and Liefeng Bo. Emo: Emote portrait alive generating expressive portrait videos with audio2video diffusion model under weak conditions. In *Computer Vision – ECCV 2024: 18th European Conference, Milan, Italy, September 29–October 4, 2024, Proceedings, Part LXXXIII*, pp. 244–260, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-73009-2. doi: 10.1007/978-3-031-73010-8_15. URL https://doi.org/10.1007/978-3-031-73010-8_15.

Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 267–284, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/unger.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, 2024. URL https://api.semanticscholar.org/CorpusID:271404773.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Jin Xu, Zhifang Guo, Jinzheng He, Hangrui Hu, Ting He, Shuai Bai, Keqin Chen, Jialin Wang, Yang Fan, Kai Dang, Bin Zhang, Xiong Wang, Yunfei Chu, and Junyang Lin. Qwen2.5-omni technical report, 2025a. URL https://arxiv.org/abs/2503.20215.

Jin Xu, Zhifang Guo, Hangrui Hu, Yunfei Chu, Xiong Wang, Jinzheng He, Yuxuan Wang, Xian Shi, Ting He, Xinfa Zhu, Yuanjun Lv, Yongqi Wang, Dake Guo, He Wang, Linhan Ma, Pei Zhang, Xinyu Zhang, Hongkun Hao, Zishan Guo, Baosong Yang, Bin Zhang, Ziyang Ma, Xipin Wei, Shuai Bai, Keqin Chen, Xuejing Liu, Peng Wang, Mingkun Yang, Dayiheng Liu, Xingzhang Ren, Bo Zheng, Rui Men, Fan Zhou, Bowen Yu, Jianxin Yang, Le Yu, Jingren Zhou, and Junyang Lin. Qwen3-omni technical report, 2025b. URL https://arxiv.org/abs/2509.17765.

Mingwang Xu, Hui Li, Qingkun Su, Hanlin Shang, Liwei Zhang, Ce Liu, Jingdong Wang, Yao Yao, and Siyu zhu. Hallo: Hierarchical audio-driven visual synthesis for portrait image animation, 2024.

Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. Gspmd: General and scalable parallelization for ml computation graphs, 2021. URL https://arxiv.org/abs/2105.04663.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Peiqi Yin, Jiangyun Zhu, Han Gao, Chenguang Zheng, Yongxiang Huang, Taichang Zhou, Ruirui Yang, Weizhi Liu, Weiqing Chen, Canlin Guo, Didan Deng, Zifeng Mo, Cong Wang, James Cheng, Roger Wang, and Hongsheng Liu. vllm-omni: Fully disaggregated serving for any-to-any multimodal models. *arXiv preprint arXiv:2602.02204*, 2026.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Yue Zhang, Zhizhou Zhong, Minhao Liu, Zhaokang Chen, Bin Wu, Yubin Zeng, Chao Zhan, Yingjie He, Junxin Huang, and Wenjiang Zhou. Musetalk: Real-time high-fidelity video dubbing via spatio-temporal sampling. *arxiv*, 2025a.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization, 2025b. URL https://arxiv.org/abs/2511.01884.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: towards optimal large language model serving throughput. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-47-2.

A Extended Related Work

This section expands on the lines of work summarized in Section 2.

LLM and multimodal serving systems. LLM serving engines exploit batching, paged attention, and continuous scheduling (Kwon et al., 2023; Yu et al., 2022; Zheng et al., 2024); they also enable separating or interleaving the prefill and decode phases of inference (Zhong et al., 2024; Patel et al., 2025; Agrawal et al., 2024; Qin et al., 2025), and they can pipeline compute, memory, and networking within a device (Zhu et al., 2025). Multimodal systems (Yin et al., 2026; Chung et al., 2026; Qiu et al., 2026) generalize these mechanisms across modalities, but they assume that the application is a graph over a small fixed set of manually defined stages, and their deployments are predominantly disaggregated at stage boundaries. They therefore suit throughput-oriented workloads but offer limited support for the streaming, intra-stage parallelism, and co-location decisions that low-latency pipelines often require. Even when every component of an application is served by such an engine, how the engines should be co-scheduled is still left to the developer.

Auto-parallelism and ML compilers. Auto-parallelism systems such as FlexFlow (Jia et al., 2019b), GSPMD (Xu et al., 2021), Alpa (Zheng et al., 2022), and Unity (Unger et al., 2022) jointly choose data, model, and pipeline parallelism for distributed deep learning, but their search spaces and cost models are calibrated to dense DNN training; likewise, pipeline-parallel formalisms such as PipeDream (Narayanan et al., 2019) assume a single homogeneous training graph. Neither transfers to inference pipelines of heterogeneous models with potentially disjoint runtimes, many of which are served by opaque engines whose internals cannot be inspected. Operator-level compilers such as TVM (Chen et al., 2018), TASO (Jia et al., 2019a), and Halide (Ragan-Kelley et al., 2013) aggressively rewrite kernel-level computation, below the level at which most deployment decisions for multimodal pipelines are made: they cannot co-locate different models or overlap different intra-model stages on shared resources. In both lines of work, extending the search space to a new workload requires dedicated systems effort.

Streaming and dataflow IRs. Decades of work on dataflow programming has formalized the abstractions FlashRT's IR builds on. Synchronous dataflow (Lee & Messerschmitt, 1987) establishes periodic-rate semantics that make a chunk the natural scheduling unit to reason about buffering. StreamIt (Thies et al., 2002) demonstrates that explicit streaming graphs enable aggressive pipelining and fission. Naiad (Murray et al., 2013) and Apache Beam (Akidau et al., 2015) extended these

ideas to large-scale, low-latency distributed processing. Distributed runtimes such as Ray (Moritz et al., 2018) provide the underlying infrastructure to deploy and execute such graphs in practice. We build on the periodic-rate and typed-edge design of these IRs, adapting it for real-time multimodal inference, where computations must maintain state across chunks and support stages that run on different runtimes. Our representation is designed to be general enough so it can capture diverse pipelines but structured enough so an agent can mechanically derive parallelism opportunities from graph analyses.

Coding agents for performance optimization. In addition to existing work on agents for kernel optimization (Ouyang et al., 2025; Liao et al., 2026; Wei et al., 2025; Lange et al., 2025; Baronio et al., 2025; Li et al., 2025b;a; Zhang et al., 2025b), compiler pass selection (Lin et al., 2025), scientific discovery (Novikov et al., 2025; Romera-Paredes et al., 2024), and reward design (Ma et al., 2023) (Section 2), prior work on local code edits (Shypula et al., 2024) explores agent-driven performance improvements on small code segments. FLASHRT differs from these in two respects. First, the optimization target is an end-to-end multimodal deployment that spans multiple processes, GPUs, and model runtimes, rather than a single kernel or file. Second, we structure the agent's work around an explicit IR that exposes the space of legal transformations, rather than asking the agent to derive deployment plans from raw source on each iteration. Combined with a measurement-gated, queue-driven loop, this scaffolding allows a generic coding agent to convert open-ended pipeline optimization into a structured search.

B FORMAL ANALYSIS OF PIPELINE GRAPHS

Here we provide formalizations for pipeline graphs to properly analyze various deployment strategies on a few applications. Although our analysis can extend to more general applications, we focus on applications with streaming generation so we can relate our analysis to applications we performed empirical analysis against in Section 5.

B.1 SETUP

We introduce a general framework for formally defining applications as pipeline graphs with explicit deployment considerations to assist with the later analysis.

B.1.1 INITIAL ASSUMPTIONS

We take some initial assumptions to simplify the downstream analysis. Assume a single request produces an indefinitely long stream of batches indexed by $j \geq 0$. Users emit N actions per batch period T, one action per subinterval of length $\Delta = \frac{T}{N}$, and the pipeline begins processing a batch once all of its actions have arrived. Action $jN + k$ arrives during

$$[jT + k\Delta,\ jT + (k+1)\Delta)$$

and becomes chunk k of batch j. We conservatively take batch j to be ready at

$$R_j = (j+1)T.$$

The output-rate is constant, so the output corresponding to chunk k of batch j is returned at

$$d_{j,k} \ = \ d_0 + jT + k\Delta,$$

where d_0 is a single offset, fixed once at the start of the stream and shared across every batch for the request. A smaller d_0 corresponds to lower latency, so the objective is to make the display schedule as tight as possible while remaining feasible across the entire stream of batches.

B.1.2 PIPELINE GRAPH

Each batch is processed by a fixed collection of operations (e.g., a DiT followed by a streaming VAE) that together turn the batch's N actions into N display-ready output chunks. We describe this processing pipeline by a static template plus a deployment plan. The template, independent of how we deploy, has:

- $\mathcal{V}$, a finite set of operation types, each running once per batch (so the instances are pairs (j, v) for $j \geq 0$, and a single batch's processing comprises all $|\mathcal{V}|$ instances $\{(j, v) : v \in \mathcal{V}\}$);
- $\mathcal{R}$, a finite set of resources (a single GPU, or a multi-GPU group treated as one resource);
- a set $E \subseteq \mathcal{V} \times \mathcal{V} \times \{0, 1\}$ of logical edges. Each edge $(u, v, \lambda) \in E$ has an *edge batch-shift* $\lambda \in \{0, 1\}$ and means that (j, v) cannot start until $(j - \lambda, u)$ finishes, with $\lambda = 0$ encoding an intra-batch dependency and $\lambda = 1$ encoding a dependency on the previous batch's instance;
- a subset $\mathcal{V}_{\text{in}} \subseteq \mathcal{V}$ of *entry* operations gated by action arrival, meaning that no instance (j, v) with $v \in \mathcal{V}_{\text{in}}$ may start in batch j before time R_j;
- an output map $o : \{0, \ldots, N-1\} \to \mathcal{V}$ identifying the operation whose completion releases the output for chunk k of any given batch.

The deployment plan specifies:

- an assignment $\rho : \mathcal{V} \to \mathcal{R}$;
- for each resource $r \in \mathcal{R}$, a nonpreemptive time-indexed schedule

$$A_r : \mathbb{R}_{\geq 0} \to \{(j, v) : j \geq 0,\ \rho(v) = r\} \cup \{\varnothing\},$$

  where $A_r(t) = (j, v)$ means that resource r is executing operation instance (j, v) at time t, and $A_r(t) = \varnothing$ means that r is idle.

When a resource follows a fixed cyclic per-batch order, we write $\sigma_r = (v_{r,1}, \ldots, v_{r,m_r})$ as shorthand for the corresponding time-indexed schedule A_r. In that special case, resource r executes $v_{r,1}, \ldots, v_{r,m_r}$ for batch j before beginning batch $j + 1$.

Given the deployment, the deployment-dependent timing data is:

- a processing time $\tau_v(\rho)$ for each operation, allowed to depend on ρ (e.g. a sequence-parallel implementation runs faster on a multi-GPU resource);
- a synchronization lag $\kappa_{u,v,\lambda}(\rho) \geq 0$ on each edge, also ρ-dependent (e.g., an inter-GPU transfer is non-zero only when the two operations are assigned to different resources).

For each operation instance (j, v) with $j \geq 0$ and $v \in \mathcal{V}$, let $S_{j,v}$ denote its start time and

$$C_{j,v} := S_{j,v} + \tau_v(\rho)$$

its completion time. A deployment (ρ, A) is feasible if, for every instance (j, v), the schedule satisfies

$$A_{\rho(v)}(t) = (j, v) \qquad \text{for all } t \in [S_{j,v}, C_{j,v}),$$

and $A_{\rho(v)}(t) \neq (j, v)$ outside this interval. Thus each operation runs nonpreemptively for exactly $\tau_v(\rho)$ time on its assigned resource, and no resource executes more than one operation instance at a time. In addition, all entry and logical-precedence constraints must hold:

$$S_{j,v} \geq R_j \qquad \text{for all } v \in \mathcal{V}_{\text{in}},$$

and, for every edge $(u, v, \lambda) \in E$,

$$S_{j,v} \geq C_{j-\lambda,u} + \kappa_{u,v,\lambda}(\rho)$$

whenever the predecessor instance $(j - \lambda, u)$ exists. A cross-batch ($\lambda = 1$) edge encodes a logical inter-batch dependency, for instance a shared KV cache that batch $j + 1$ inherits from batch j. Its κ captures any cost of moving that state between resources.

B.1.3 Latency metric

Chunk (j, k) becomes display-ready at $Y_{j,k} := C_{j,o(k)}$, the completion time of $o(k)$ in batch j.

Lemma B.1 (Display-offset characterization)**.** *Fix a feasible deployment (ρ, A). The display schedule with offset d_0 is feasible if and only if*

$$d_0 \geq Y_{j,k} - jT - k\Delta$$

for every (j, k). Consequently, the smallest feasible offset for the deployment is

$$d_0^*(\rho, A) \;:=\; \sup_{j,k}\big(Y_{j,k} - jT - k\Delta\big).$$

Proof. Chunk (j, k) is scheduled to be displayed at time $d_0 + jT + k\Delta$ and is display-ready at time $Y_{j,k}$. Thus the schedule is feasible exactly when

$$d_0 + jT + k\Delta \geq Y_{j,k}$$

for every (j, k), which is equivalent to

$$d_0 \geq Y_{j,k} - jT - k\Delta$$

for every (j, k). Taking the supremum over all chunks gives the smallest feasible offset. □

We use $d_0^*(\rho, A)$ as the action-to-visible latency metric for a deployment. When the goal is latency optimization, one seeks deployments with smaller $d_0^*(\rho, A)$. Below we will write $d_0^*(\rho)$ as shorthand for $d_0^*(\rho, A)$ when the schedule A is determined by context, for example by the earliest-start schedule or by a fixed cyclic order σ_r.

B.1.4 Two key bounds

Lemma B.2 (Path bound)**.** *Fix a feasible deployment (ρ, A). For any logical path $P : v_0 \to v_1 \to \cdots \to v_m$ in E from an entry operation $v_0 \in \mathcal{V}_{\text{in}}$ to a chunk-k output $v_m = o(k)$, write $\lambda_i \in \{0, 1\}$ for the edge batch-shift on the i-th edge of P and define the path's* total backward batch shift

$$\Lambda(P) := \sum_{i=1}^{m} \lambda_i$$

(the number of batches by which P's entry instance precedes its output instance). Define the path's total weight

$$w(P, \rho) \;:=\; \sum_{i=0}^{m} \tau_{v_i}(\rho) + \sum_{i=1}^{m} \kappa_{v_{i-1}, v_i, \lambda_i}(\rho).$$

Then, the smallest feasible display offset d_0^* satisfies

$$d_0^* \;\geq\; w(P, \rho) + (1 - \Lambda(P))T - k\Delta.$$

Proof. Traversing the edges of P from v_m back to v_0, each edge $(v_{i-1}, v_i, \lambda_i)$ shifts the batch index back by λ_i. Hence if $v_m = o(k)$ is at batch j, then v_i is at batch $j_i := j - \sum_{q=i+1}^{m} \lambda_q$. In particular, v_0 is at batch $j_0 = j - \Lambda(P)$. The entry constraint gives $S_{j_0, v_0} \geq R_{j_0}$, while the per-operation durations and per-edge precedence relations are

$$C_{j_i, v_i} = S_{j_i, v_i} + \tau_{v_i}(\rho), \qquad S_{j_{i+1}, v_{i+1}} \geq C_{j_i, v_i} + \kappa_{v_i, v_{i+1}, \lambda_{i+1}}(\rho).$$

Telescoping these along P yields

$$C_{j, o(k)} \;=\; C_{j_m, v_m} \;\geq\; S_{j_0, v_0} + \sum_{i=0}^{m} \tau_{v_i}(\rho) + \sum_{i=1}^{m} \kappa_{v_{i-1}, v_i, \lambda_i}(\rho) \;\geq\; R_{j_0} + w(P, \rho).$$

Since $R_{j_0} = (j_0{+}1)T = (j{+}1{-}\Lambda(P))T$, combining this with Lemma B.1 and using $Y_{j,k} = C_{j, o(k)}$ gives

$$\begin{aligned} d_0^* \;&\geq\; Y_{j,k} - jT - k\Delta \\ &\geq\; w(P, \rho) + (j + 1 - \Lambda(P))T - jT - k\Delta \\ &=\; w(P, \rho) + (1 - \Lambda(P))T - k\Delta. \end{aligned}$$

□

Intuitively, the bound implies that the latency-binding paths are those with high weight $w(P, \rho)$, low batch shift $\Lambda(P)$, and small chunk index k, which corresponds to the tightest display deadline.

For throughput analysis, we instead ask how small the inter-batch period T can be while maintaining finite display offset. This sustainable-period metric is captured by the following resource-load bound.

Lemma B.3 (Resource-load bound)**.** *Recall that T is the batch period, i.e., the duration between receiving consecutive batches. Any deployment with finite display offset must satisfy*

$$T \;\geq\; T_{\min}(\rho) \;:=\; \max_{r \in \mathcal{R}} W_r(\rho), \qquad W_r(\rho) \;:=\; \sum_{v:\rho(v)=r} \tau_v(\rho).$$

Proof. Fix a resource r. Each batch requires

$$W_r(\rho) = \sum_{v:\rho(v)=r} \tau_v(\rho)$$

units of processing time on resource r.

Assume the deployment has finite display offset. Then there exists a constant $B < \infty$ such that the output-relevant work for batch j completes no later than $jT + B$. In particular, the work on resource r for the first J batches must be completed by time at most $(J-1)T + B$.

The first J batches require $JW_r(\rho)$ units of processing on resource r. Since resource r can perform at most one unit of work per unit time, we must have

$$JW_r(\rho) \leq (J-1)T + B.$$

Dividing by J gives

$$W_r(\rho) \leq \left(1 - \frac{1}{J}\right) T + \frac{B}{J}.$$

Taking $J \to \infty$ yields

$$W_r(\rho) \leq T.$$

Since this holds for every resource r,

$$T \geq \max_{r \in \mathcal{R}} W_r(\rho) = T_{\min}(\rho).$$

□

Equivalently, $T_{\min}(\rho)$ is the deployment's sustainable-period metric, and $\frac{N}{T_{\min}(\rho)}$ is its peak displayed-frame rate. Thus smaller $d_0^*(\rho, A)$ corresponds to better latency, while smaller $T_{\min}(\rho)$ corresponds to better throughput.

B.2 NP-HARDNESS OF DEPLOYMENT SEARCH

We briefly instantiate the framework above to show that the deployment search problem already contains a classical NP-hard machine-scheduling problem as a special case. We reduce from identical parallel-machine makespan minimization, denoted $P||C_{\max}$ in the standard three-field scheduling notation: P indicates identical parallel machines, the empty middle field indicates that there are no additional job constraints, and $C_{\max}$ is the makespan objective. In the decision version, the input consists of jobs $1, \ldots, n$, processing times $p_1, \ldots, p_n$, m identical machines, and a makespan threshold B; the goal is to decide whether there exists a non-preemptive schedule with makespan at most B. This problem is NP-hard (Lenstra et al., 1977).

Given such an instance, construct a deployment-search instance with

$$\mathcal{V} = \{v_1, \ldots, v_n\}, \qquad \mathcal{R} = \{r_1, \ldots, r_m\}, \qquad E = \varnothing.$$

Each operation v_i corresponds to job i, and each resource r_ℓ corresponds to machine ℓ. The execution time of each operation is placement-independent:

$$\tau_{v_i}(\rho) = p_i$$

for every placement ρ. Since there are no logical edges, there are no data dependencies, state dependencies, or synchronization costs. We may take all operations to be entry operations, $\mathcal{V}_{\text{in}} = \mathcal{V}$; the output map is arbitrary, since this reduction concerns the sustainable-period objective.

For any placement $\rho : \mathcal{V} \to \mathcal{R}$, the per-batch load on resource r is

$$W_r(\rho) = \sum_{v_i : \rho(v_i) = r} p_i.$$

Thus the sustainable period from Lemma B.3 is

$$T_{\min}(\rho) = \max_{r \in \mathcal{R}} W_r(\rho) = \max_{r \in \mathcal{R}} \sum_{v_i : \rho(v_i) = r} p_i.$$

This quantity is exactly the makespan of the corresponding assignment of jobs to machines. Therefore, there exists a deployment with

$$T_{\min}(\rho) \le B$$

if and only if the original $P||C_{\max}$ instance admits a non-preemptive schedule with makespan at most B.

Consequently, even this restricted version of deployment search (with no data dependencies, no state dependencies, no communication costs, placement-independent execution times, and only non-preemptive resource capacity constraints) is NP-hard. The general multimodal deployment problem strictly contains this special case, because it additionally allows dependency edges, placement-dependent execution times, synchronization costs, co-location, disaggregation, streaming, batching, and intra-model parallelism; as such, the general multimodal deployment problem must also be NP-hard.

B.3 Application 1: actions → DiT → VAE

We analyze a two-stage video-generation pipeline consisting of a DiT stage followed by a VAE stage, the structure shared by the WorldPlay, Krea-Realtime, and LongLive applications evaluated in Section 5.3. We compare two deployment families under a fixed budget of G GPUs: a co-located sequence-parallel deployment, where both stages share one resource, and a disaggregated pipeline-parallel deployment, where DiT and VAE run on separate resources. We evaluate both action-to-visible latency (measured by d_0^*), and throughput (measured by $T_{\min}$). We show that co-location is optimal for action-to-visible latency, while disaggregation is optimal for throughput when its best split has no larger sustainable period than the co-located deployment.

Template. The static template has:

- operations $\mathcal{V} = \{D, V\}$, where D is the DiT operation for one batch and V is the VAE decode operation;
- resources $\mathcal{R}$ introduced per deployment below;
- edges $E = \{(D, V, 0),\ (D, D, 1),\ (V, V, 1)\}$;
- entry $\mathcal{V}_{\text{in}} = \{D\}$;
- output map $o(k) = V$ for every chunk $k \in \{0, \dots, N-1\}$.

The edge $(D, V, 0)$ is the intra-batch DiT-to-VAE dependency. The inter-batch self-edges $(D, D, 1)$ and $(V, V, 1)$ encode stage-local ordering or recurrent-state dependencies across consecutive batches.

Deployment plan. We compare the following deployment families.

In the co-located deployment ρ_{col}, both stages are assigned to the same G-GPU resource r_{col}:

$$\rho_{\text{col}}(D) = \rho_{\text{col}}(V) = r_{\text{col}}, \qquad \sigma_{r_{\text{col}}} = (D, V).$$

Thus the shared resource executes DiT and then VAE for batch j before beginning DiT for batch $j+1$. Since the two stages are co-located,

$$\kappa_{D,V,0}(\rho_{\text{col}}) = 0.$$

In a disaggregated deployment $\rho_{\text{pipe}}^{X,Y}$, DiT is assigned to an X-GPU resource and VAE is assigned to a disjoint Y-GPU resource, where $X + Y = G$ and $X, Y \geq 1$:

$$\rho_{\text{pipe}}^{X,Y}(D) = r_D^X, \qquad \rho_{\text{pipe}}^{X,Y}(V) = r_V^Y,$$

with cyclic schedules

$$\sigma_{r_D^X} = (D), \qquad \sigma_{r_V^Y} = (V).$$

Thus the DiT and VAE resources can process different batches concurrently in steady state.

Proposition B.4 (Co-location is latency-optimal for DiT–VAE)**.** *For the DiT–VAE deployments above, assume the requested period T is sustainable and that assigning a stage to the full co-located resource does not make it slower than assigning that stage to a strict subset of the GPUs:*

$$\tau_D(\rho_{\text{col}}) \leq \tau_D(\rho_{\text{pipe}}^{X,Y}), \qquad \tau_V(\rho_{\text{col}}) \leq \tau_V(\rho_{\text{pipe}}^{X,Y})$$

for every split $X + Y = G$. Then

$$d_0^*(\rho_{\text{col}}) \leq d_0^*(\rho_{\text{pipe}}^{X,Y}) \qquad \textit{for every split } X + Y = G.$$

Proof. For each chunk k, the direct logical path from the entry to the output is

$$P_k : \quad D \to V.$$

This path has $\Lambda(P_k) = 0$ and weight

$$w(P_k, \rho) = \tau_D(\rho) + \kappa_{D,V,0}(\rho) + \tau_V(\rho).$$

By Lemmas B.1 and B.2,

$$d_0^*(\rho) \geq T + \tau_D(\rho) + \kappa_{D,V,0}(\rho) + \tau_V(\rho) - k\Delta.$$

The tightest constraint is for $k = 0$, and for the deployments considered here this lower bound is achieved by the earliest-start schedule whenever T is sustainable. Therefore

$$d_0^*(\rho) = T + \tau_D(\rho) + \kappa_{D,V,0}(\rho) + \tau_V(\rho).$$

For co-location, $\kappa_{D,V,0}(\rho_{\text{col}}) = 0$, so

$$d_0^*(\rho_{\text{col}}) = T + \tau_D(\rho_{\text{col}}) + \tau_V(\rho_{\text{col}}).$$

For a disaggregated split,

$$d_0^*(\rho_{\text{pipe}}^{X,Y}) = T + \tau_D(\rho_{\text{pipe}}^{X,Y}) + \kappa_{D,V,0}(\rho_{\text{pipe}}^{X,Y}) + \tau_V(\rho_{\text{pipe}}^{X,Y}).$$

Since synchronization lags are nonnegative and the co-located resource does not make either stage slower, subtracting the two expressions gives

$$d_0^*(\rho_{\text{pipe}}^{X,Y}) - d_0^*(\rho_{\text{col}}) \geq 0.$$

Thus co-location minimizes the latency-binding path $D \to V$ within the deployment family considered here. □

Proposition B.5 (Disaggregation can be throughput-optimal for DiT–VAE)**.** *For the DiT–VAE deployments above, the co-located deployment has sustainable period*

$$T_{\min}(\rho_{\text{col}}) = \max\left\{\tau_D(\rho_{\text{col}}) + \tau_V(\rho_{\text{col}}),\ \tau_D(\rho_{\text{col}}) + \kappa_{D,D,1}(\rho_{\text{col}}),\ \tau_V(\rho_{\text{col}}) + \kappa_{V,V,1}(\rho_{\text{col}})\right\},$$

whereas a disaggregated split has sustainable period

$$T_{\min}(\rho_{\text{pipe}}^{X,Y}) = \max\left\{\tau_D(\rho_{\text{pipe}}^{X,Y}) + \kappa_{D,D,1}(\rho_{\text{pipe}}^{X,Y}),\ \tau_V(\rho_{\text{pipe}}^{X,Y}) + \kappa_{V,V,1}(\rho_{\text{pipe}}^{X,Y})\right\}.$$

Consequently, any split

$$(X^*, Y^*) \in \underset{X+Y=G,\ X,Y\geq 1}{\arg\min} T_{\min}(\rho_{\text{pipe}}^{X,Y})$$

is throughput-optimal among disaggregated deployments. If

$$T_{\min}(\rho_{\text{pipe}}^{X^*,Y^*}) \leq T_{\min}(\rho_{\text{col}}),$$

then it is throughput-optimal among the compared co-located and disaggregated deployments.

Proof. For the co-located deployment, the shared resource must execute both operations once per batch. By Lemma B.3, this gives

$$T \geq \tau_D(\rho_{\text{col}}) + \tau_V(\rho_{\text{col}}).$$

The inter-batch self-edges additionally require

$$T \geq \tau_D(\rho_{\text{col}}) + \kappa_{D,D,1}(\rho_{\text{col}}), \qquad T \geq \tau_V(\rho_{\text{col}}) + \kappa_{V,V,1}(\rho_{\text{col}}),$$

which yields the stated expression for $T_{\min}(\rho_{\text{col}})$.

For a disaggregated split, the two stages execute on separate resources. Applying Lemma B.3 to each resource, together with the inter-batch self-edges, yields the stated expression for $T_{\min}(\rho^{X,Y}_{\text{pipe}})$. The intra-batch DiT-to-VAE lag $\kappa_{D,V,0}(\rho^{X,Y}_{\text{pipe}})$ affects latency because it lies on the path $D \to V$; it affects throughput only if the communication fabric is modeled as an additional bottleneck resource.

The optimality claim follows directly from the definition of (X^*, Y^*) and from comparing its sustainable period against that of co-location. If the inequality is strict, then the disaggregated deployment achieves a strictly higher peak output rate:

$$\frac{N}{T_{\min}(\rho^{X^*,Y^*}_{\text{pipe}})} > \frac{N}{T_{\min}(\rho_{\text{col}})}.$$

□

Conclusion. Propositions B.4 and B.5 capture a latency-throughput tradeoff: co-location minimizes the latency-binding path $D \to V$, while disaggregation can reduce the sustainable period by allowing DiT and VAE work from different batches to run concurrently.

B.4 Application 2: sequence parallelism vs stage pipeline parallelism

We now compare sequence parallelism (SP) and stage pipeline parallelism (PP) for DiT architectures that admit pipeline parallelism across denoising steps, such as the LiveAvatar S2V model deployed in Sections 4.1 and 5.2. The key architectural assumption is that each denoising step maintains independent per-step KV cache, thus allowing step Q_i of batch $j+1$ to depend only on step Q_i of batch j rather than on the completion of the entire denoising chain for batch j. Under this assumption, different denoising steps can be assigned to different resources and executed as a stage pipeline across batches.

Here, we do not prove global optimality over all possible deployments, but we compare SP and PP under the two metrics introduced above: action-to-visible latency (measured by d_0^*) and throughput (measured by $T_{\min}$).

Template. The static template has:

- operations $\mathcal{V} = \{Q_1, \ldots, Q_n\}$, where Q_i is the i-th denoising step;
- resources $\mathcal{R}$ introduced per deployment below;
- edges $E = \{(Q_i, Q_{i+1}, 0) : i = 1, \ldots, n-1\} \cup \{(Q_i, Q_i, 1) : i = 1, \ldots, n\}$;
- entry $\mathcal{V}_{\text{in}} = \{Q_1\}$;
- output map $o(k) = Q_n$ for every chunk $k \in \{0, \ldots, N-1\}$.

The intra-batch edges $(Q_i, Q_{i+1}, 0)$ encode the denoising chain within a batch. The inter-batch edges $(Q_i, Q_i, 1)$ encode the per-step KV cache dependency enables stage pipeline parallelism.

Let T_Q denote the single-GPU compute time for one denoising step. For an n-GPU sequence-parallel group, let $T_Q^{SP}(n)$ denote the wall-clock time for one denoising step under sequence parallelism, including any required computation, communication, and synchronization. We assume

$$0 < T_Q^{SP}(n) \leq T_Q,$$

so sequence parallelism does not make an individual denoising step slower than running it on one GPU. We do not assume exact linear speedup and instead assume

$$T_Q \leq n T_Q^{SP}(n),$$

i.e., sublinear sequence-parallel speedup.

Deployment plan. We compare two deployments using n GPUs.

In the SP deployment ρ_{SP}, all denoising steps are assigned to one n-GPU SP resource r_n^{SP}:

$$\rho_{SP}(Q_i) = r_n^{SP} \qquad \text{for every } i.$$

The resource follows the cyclic schedule

$$\sigma_{r_n^{SP}} = (Q_1, \dots, Q_n),$$

so it executes all denoising steps of batch j before beginning the DiT work for batch $j+1$. The per-step time is

$$\tau_{Q_i}(\rho_{SP}) = T_Q^{SP}(n).$$

Since all denoising steps are co-located,

$$\kappa_{Q_i,Q_{i+1},0}(\rho_{SP}) = 0.$$

In the PP deployment ρ_{PP}, each denoising step is assigned to its own dedicated GPU:

$$\rho_{PP}(Q_i) = r_i, \qquad r_i \neq r_{i'} \text{ for } i \neq i'.$$

Each resource r_i follows the cyclic schedule

$$\sigma_{r_i} = (Q_i),$$

executing step Q_i for the next batches whenever logical predecessors complete. The per-step time is

$$\tau_{Q_i}(\rho_{PP}) = T_Q.$$

Since consecutive denoising steps are assigned to different resources, the intra-batch edges may have nonzero synchronization lags

$$\kappa_{Q_i,Q_{i+1},0}(\rho_{PP}) \geq 0.$$

In both deployments, each Q_i remains on the same resource across batches, so the per-step KV cache does not move across resources:

$$\kappa_{Q_i,Q_i,1}(\rho_{SP}) = \kappa_{Q_i,Q_i,1}(\rho_{PP}) = 0.$$

Proposition B.6 (SP has no higher first-frame latency than stage PP)**.** *For the SP and PP deployments above, assume the requested period T is sustainable and*

$$T_Q^{SP}(n) \leq T_Q.$$

Then

$$d_0^*(\rho_{SP}) \leq d_0^*(\rho_{PP}).$$

Proof. For each chunk k, the direct logical path from the entry to the output is

$$P_k : \quad Q_1 \to Q_2 \to \cdots \to Q_n.$$

This path has $\Lambda(P_k) = 0$. By Lemmas B.1 and B.2,

$$d_0^*(\rho) \geq T + w(P_k, \rho) - k\Delta.$$

The tightest constraint is for $k = 0$, and for the deployments considered here this lower bound is achieved by the earliest-start schedule whenever T is sustainable.

For SP,

$$w(P_0, \rho_{SP}) = nT_Q^{SP}(n),$$

so

$$d_0^*(\rho_{SP}) = T + nT_Q^{SP}(n).$$

For PP,

$$w(P_0, \rho_{PP}) = nT_Q + \sum_{i=1}^{n-1} \kappa_{Q_i,Q_{i+1},0}(\rho_{PP}),$$

so

$$d_0^*(\rho_{PP}) = T + nT_Q + \sum_{i=1}^{n-1} \kappa_{Q_i,Q_{i+1},0}(\rho_{PP}).$$

Therefore

$$d_0^*(\rho_{PP}) - d_0^*(\rho_{SP}) = n\big(T_Q - T_Q^{SP}(n)\big) + \sum_{i=1}^{n-1} \kappa_{Q_i,Q_{i+1},0}(\rho_{PP}) \geq 0.$$

Thus SP has no higher first-frame latency than stage PP under the stated assumptions. □

Proposition B.7 (Stage PP has no larger sustainable period under sublinear SP speedup)**.** *For the SP and PP deployments above,*

$$T_{\min}(\rho_{SP}) = nT_Q^{SP}(n), \qquad T_{\min}(\rho_{PP}) = T_Q.$$

Consequently, under the throughput assumption

$$T_Q \le nT_Q^{SP}(n),$$

we have

$$T_{\min}(\rho_{PP}) \le T_{\min}(\rho_{SP}).$$

Proof. For SP, the shared resource must execute all n denoising steps once per batch. By Lemma B.3,

$$T \ge \sum_{i=1}^{n} T_Q^{SP}(n) = nT_Q^{SP}(n).$$

The inter-batch self-edges have zero synchronization lag in this deployment, and resource non-overlap already enforces the required per-step ordering. Therefore

$$T_{\min}(\rho_{SP}) = nT_Q^{SP}(n).$$

For PP, each resource executes one denoising step per batch. By Lemma B.3,

$$T \ge T_Q$$

for every stage resource r_i. The inter-batch self-edges again have zero synchronization lag because each step remains on the same resource across batches. Therefore

$$T_{\min}(\rho_{PP}) = T_Q.$$

The claimed ordering follows from $T_Q \le nT_Q^{SP}(n)$. If the inequality is strict, then PP achieves a strictly higher peak frame rate:

$$\frac{N}{T_Q} > \frac{N}{nT_Q^{SP}(n)}.$$

The intra-batch synchronization lags $\kappa_{Q_i,Q_{i+1},0}(\rho_{PP})$ affect latency because they lie on the path $Q_1 \to \cdots \to Q_n$; they affect throughput only if the communication fabric is modeled as an additional bottleneck resource. □

Conclusion. Propositions B.6 and B.7 capture the SP–PP tradeoff. SP reduces the weight of the latency-binding path $Q_1 \to \cdots \to Q_n$, while PP reduces the sustainable period by distributing the denoising steps across resources. For DiTs without the per-step inter-batch dependency structure above, the PP schedule analyzed here is not feasible under the stated framework.

B.5 Bounds on non-agentic baselines

Apart from vLLM-Omni for Qwen3-Omni, no existing system deploys the applications of Section 5, so there is no non-agentic baseline to run for them. Instead, we use the framework to bound what a non-agentic system could achieve with the strategies available to it, each of which corresponds to an existing approach discussed in Section 1: fixed placement rules, such as the static policies of multimodal serving frameworks (Appendix B.5.1); tuning the reference's placement and runtime settings, including compilation, without changing its code (Appendix B.5.2); and parallelizing the reference's computation, as an auto-parallelism framework would, without changing when it releases its output (Appendix B.5.3). For each strategy, we bound the deployments it can reach, give the corresponding bound for the deployments FlashRT found, and place the deployments measured in Section 5 against both. We omit Qwen3-Omni, which is compared against vLLM-Omni directly.

Search over the IR. A rule-based or heuristic search could choose among the strategies that the IR marks as legal, but only if each choice can be turned into a working deployment. In FLASHRT, the agent writes each deployment as new code: the processes and communication that realize its placement, the streaming between its components, and the pipelining of stages across GPUs. A non-agentic search would need such code written in advance for every strategy and application, the hand-built, per-workload coverage that limits existing systems (Section 1). Given that code, the search must still make the choices analyzed below, placing each pair of stages for each target and mixing placements within one deployment, as FLASHRT's deployments do. Given FLASHRT's own implementations, it would search only over deployments the agent wrote, and we report the best of them for each target.

Core and prefix. We split each application into a *core*, the operations that run once per batch, and a *prefix*, the operations that run once per request before the core starts. The core comprises the video model's stages, together with SAM 3 for Krea + SAM 3, while the prefix is LongLive's ASR and LiveAvatar's ASR and LLM. Because the prefix runs once, it does not affect the sustainable period, which Lemma B.3 determines from the core alone. Because the core's first batch depends on the prefix's output, the two never run concurrently before the first output, so the prefix adds a constant c_{pre} to the first-output latency and can share resources with the core at no cost. The results of Appendices B.3 and B.4 therefore apply to the core unchanged, with each first-output latency shifted by c_{pre}.

B.5.1 FIXED PLACEMENT RULES FOR THE CORE

A fixed rule places every pair of core stages the same way, either co-located on one resource and parallelized within each stage (e.g., with sequence, tensor, or expert parallelism), or disaggregated onto separate resources. Multimodal serving frameworks fix the rule through their stage definitions, disaggregating across stage boundaries and co-locating within a stage (Section 1). The best choice depends on how the two stages rely on each other's outputs and, for serial stages, on the target metric.

Serial stages. For consecutive stages that overlap across batches, such as the DiT and VAE or successive denoising steps, Propositions B.4 and B.6 show that co-location minimizes latency, while Propositions B.5 and B.7 show that disaggregation minimizes the per-batch period. Appendix B.4 is phrased for SP, but its proofs assume only that parallelizing a step does not slow it down and speeds it up at most linearly, so they hold for any intra-stage parallelism. A fixed rule therefore attains only one of the two bounds. Always co-locating keeps every stage in each batch's period, $\tau_D + \tau_V$ for the DiT and VAE, whereas a disaggregated split needs only its slower stage (Proposition B.5); always disaggregating adds the transfer lag $\kappa_{D,V,0}$, and the slowdown of running each stage on fewer GPUs, to the latency (Proposition B.4). FLASHRT instead chooses per target: its latency-optimized deployments co-locate the stages and its frame-rate-optimized ones disaggregate them (Section 5.3), so each attains the bound for its target. The measured deployments show that each fixed rule loses a large share of one metric: at every budget where both families were measured (WorldPlay at 2 and 4 GPUs and LongLive at 2 GPUs; Tables 6 and 8), always co-locating gives up 18–25% of the frame rate, and always disaggregating adds 16–27% latency. On 2 GPUs, for example, WorldPlay's co-located deployment reaches 493 ms and 25.5 FPS, and its disaggregated one 625 ms and 31.0 FPS.

Independent branches. Core stages on parallel branches that meet only at a final operation, such as SAM 3 and Krea-Realtime, can run concurrently once disaggregated, which improves both metrics; a rule that co-locates them, as the single-GPU reference does, gives up both.

Proposition B.8 (Disaggregating independent branches)**.** *Let operations A and B both feed a join operation M, with edges $(A, M, 0)$, $(B, M, 0)$, $(A, A, 1)$, $(B, B, 1)$, entry operations $\{A, B\}$, and $o(k) = M$ for all k. Let ρ_{col} place all three on one resource and ρ_{par} place A and M on one resource and B on another, with the same processing times, and let the requested period T be sustainable under ρ_{col}. If $\tau_B > 0$ and $\kappa_{B,M,0}(\rho_{\mathrm{par}}) \le \tau_A$, then $d_0^*(\rho_{\mathrm{par}}) \le d_0^*(\rho_{\mathrm{col}})$ and $T_{\min}(\rho_{\mathrm{par}}) < T_{\min}(\rho_{\mathrm{col}})$.*

Proof. Under ρ_{col} the resource runs A, B, and M in sequence, so $d_0^*(\rho_{\mathrm{col}}) = T + \tau_A + \tau_B + \tau_M$ and $T_{\min}(\rho_{\mathrm{col}}) = \tau_A + \tau_B + \tau_M$. Under ρ_{par}, A and B run concurrently, so $d_0^*(\rho_{\mathrm{par}}) = T + \max(\tau_A, \tau_B + \kappa_{B,M,0}) + \tau_M$ and $T_{\min}(\rho_{\mathrm{par}}) = \max(\tau_A + \tau_M, \tau_B)$. The two differences are $\min(\tau_B, \tau_A - \kappa_{B,M,0}) \geq 0$ and $\min(\tau_B, \tau_A + \tau_M) > 0$. □

FLASHRT's Krea + SAM 3 deployments are instances of ρ_{par}, with Krea-Realtime as A, SAM 3 as B on its own GPU, and the composition as M (Table 4). When SAM 3 is the shorter branch ($\tau_B + \kappa_{B,M,0} \leq \tau_A$), both differences in the proof equal τ_B, so disaggregation shortens the latency and the period by the same amount, SAM 3's time per batch. The measurements match (Table 1): from the one-GPU reference to the 2-GPU deployment, the period drops from 1.76 s to 1.04 s per 12-frame batch (6.82 to 11.54 FPS) and the latency from 1.715 s to 1.014 s, both by about 0.7 s. The 2-GPU deployment also compiles SAM 3 and emits frames as they are decoded, but neither changes its period, which in this regime is set by the Krea-Realtime branch alone ($\tau_A + \tau_M$).

No fixed rule is therefore best for every pair of core stages and both metrics: co-location loses the period for serial stages and both metrics for independent branches, while disaggregation loses latency for serial stages. FLASHRT chooses per pair of stages and per target, and can mix the two rules within one deployment; for example, its latency-optimized Krea + SAM 3 deployment disaggregates SAM 3 but co-locates Krea-Realtime's DiT and VAE on an SP-3 group (Table 4), and its 6-GPU LiveAvatar deployment co-locates the ASR with the VAE and the LLM with the TTS while pipelining the S2V model's denoising steps across four GPUs (Table 3). These choices need not trade one metric for the other: WorldPlay's 6-GPU deployment, which disaggregates the DiT and VAE and parallelizes each on its own GPU group, reaches its highest frame rate, 44.3 FPS, at 447 ms, within 22 ms of its lowest latency (Table 6), and LiveAvatar's 6-GPU deployment reaches its highest frame rate, 133.48 FPS, at 1.72 s, within 0.03 s of its lowest latency (Table 1). A rule that disaggregates the prefix, finally, only spends resources that the core could use.

B.5.2 CONFIGURATION-ONLY TUNING

Without changing the reference's code, a developer or an autotuner can move its models to other GPUs or change runtime settings such as compilation, CUDA graphs, or attention backends. The code still runs the core operations of each batch in a fixed order $(v_1, \ldots, v_m)$ and starts batch $j+1$ only after batch j finishes, which adds the edges $(v_i, v_{i+1}, 0)$ for $i < m$ and $(v_m, v_1, 1)$ to every deployment of the reference.

Proposition B.9 (Placement cannot improve the sequential reference)**.** *Let ρ_1 place every core operation on one resource, and let ρ place each core operation on a resource of the same kind, so that processing times are unchanged, with nonnegative lags that vanish between co-located operations. Every period sustainable under ρ is at least $T_{\min}(\rho_1) = \sum_v \tau_v$, and $d_0^*(\rho) \geq d_0^*(\rho_1)$ under earliest-start schedules.*

Proof. Following the added edges from an output $o(k)$ of batch j to the same output of batch $j+1$ passes through every core operation once, so $Y_{j+1,k} \geq Y_{j,k} + \sum_v \tau_v$. A finite display offset requires $Y_{j,k} \leq d_0 + jT + k\Delta$ for every j (Lemma B.1), which forces $T \geq \sum_v \tau_v$, and ρ_1 attains this bound. For latency, the added edges order all instances totally, so under ρ_1 each instance starts as soon as its predecessor completes or its input arrives. Under ρ, by induction over this order, the predecessor completes no earlier and lags are nonnegative, so every $Y_{j,k}$, and hence d_0^*, can only grow. □

Runtime settings rescale the processing times but keep the added edges, so the same argument bounds the period by $\sum_v \tau_v$ with the rescaled times. The same settings also apply to the deployments FLASHRT produces, which reuse the reference's model code, so they complement rather than replace its transformations.

The deployments FLASHRT produces are not bound by the added edges: they can overlap consecutive batches, so Lemma B.3 bounds their period only by the load of their busiest resource, $\max_r W_r(\rho)$, which is below $\sum_v \tau_v$ whenever more than one resource carries core work. For example, for a deployment that places a video model's DiT and VAE on one GPU each, the period is $\max(\tau_D, \tau_V)$ instead of $\tau_D + \tau_V$ when each stage keeps its state on its own GPU (Proposition B.5). The measurements show the gap: with the same runtime settings, no placement of the WorldPlay or LongLive reference exceeds its single-GPU frame rate of 20.4 or 25.8 FPS, or has lower latency

than its single-GPU 869 or 674 ms. FLASHRT's 2-GPU deployments reach 31.0 and 47.8 FPS by running the same DiT and VAE on one GPU each while overlapping consecutive batches, and 493 and 512 ms by parallelizing each stage across both GPUs (Table 1).

B.5.3 PARALLELIZATION WITHOUT STREAMING

The LiveAvatar reference emits its video only after generating the whole response, so its first output waits for all of the response's work, not just the prefix and the core's first batch. Parallelizing the reference's computation, for instance by sharding or pipelining its models as an auto-parallelism framework would, cannot remove this wait; it can only divide it across GPUs.

Proposition B.10 (Work bound)**.** *Suppose the first output requires work whose total processing time on a single GPU is W, all of which depends on the input. If running any operation on g GPUs takes at least $1/g$ of its single-GPU time, then every deployment on G GPUs releases the first output at least W/G after the input arrives.*

Proof. The required work occupies at least W GPU-seconds, all of it between the input's arrival and the first output. G GPUs supply at most G GPU-seconds per second, so this interval lasts at least W/G. □

FLASHRT's LiveAvatar deployments stream instead: they release the first frame once the prefix and the core's first batch complete, so their latency is c_{pre} plus the core's first-output latency analyzed in Appendices B.3 and B.4, which does not grow with the length of the response, while W does. The LiveAvatar reference runs sequentially on one GPU, so W is its measured latency of 117.10 s (Table 1). Any deployment that parallelizes the reference but keeps its output behavior therefore has latency at least 58.6 s on 2 GPUs and 19.5 s on 6 GPUs, whereas FLASHRT's streaming deployments reach 1.69 s and 1.72 s, respectively. The stage-PP deployment without streaming, which pipelines the S2V model's denoising steps like FLASHRT's 6-GPU deployment but emits the video only at the end, reaches 36.29 s on 6 GPUs (Table 12); this is above the bound, which assumes all 6 GPUs stay busy until the first output. Parallelism alone thus cannot approach FLASHRT's latency, which comes from streaming.

C EXTENDED DEPLOYMENT RESULTS AND SCALING ANALYSIS

In Section 5, our experiments present latency- and frame-rate-optimized deployment variants for each application across a range of GPU budgets. This appendix first gives the GPU placements of the multi-GPU B200 deployments (Appendix C.1) and then provides extended results for two of these applications, the video world model (WorldPlay) from Section 5.3 and the video narrator (LongLive) from Section 5.3. For each application, we report additional deployments the FLASHRT agent found and analyze in depth the factors that drive how latency and frame rate scale with the number of GPUs. Finally, Appendix C.4 lists the deployments found on AMD MI355X GPUs, which Section 5.4 summarizes.

C.1 PLACEMENTS OF THE B200 DEPLOYMENTS

Tables 3 to 5 give the node-to-GPU placements ρ of the multi-GPU B200 deployments reported in Table 1.

C.2 VIDEO WORLD MODEL (WORLDPLAY)

We use the WorldPlay video world model of Section 5.3, in which the user's keyboard actions drive an autoregressive DiT and a streaming VAE that render a continuous video stream. Table 6 reports the full set of deployments the agent found across GPU budgets. At each budget the agent offers a latency-optimized deployment that co-locates the DiT and VAE at a higher SP degree and a frame-rate-optimized deployment that disaggregates them onto separate GPUs so they can be pipelined, and at 6 GPUs it combines the two. Table 7 shows how the disaggregated deployment scales from 2 to 6 GPUs. We analyze how these choices trade latency against frame rate as the budget grows.

Table 3: Placement ρ for the LiveAvatar conversational-agent deployments. The sequential baseline (omitted) places all nodes on one GPU. At 6 GPUs, the S2V DiT's four denoising steps (DiT_1–DiT_4) are pipelined one per GPU (the same per-step decomposition of Appendix B.4), while the ASR shares a GPU with the VAE and the LLM shares a GPU with the TTS.

	2 GPU (streaming)					**6 GPU (streaming + S2V PP)**							
GPU	ASR	LLM	TTS	DiT	VAE	ASR	LLM	TTS	DiT_1	DiT_2	DiT_3	DiT_4	VAE
G0	X			X	X				X				
G1		X	X							X			
G2											X		
G3												X	
G4						X							X
G5							X	X					
G6													
G7													

Table 4: Placement ρ for the video background editor deployments. The latency-optimized deployment co-locates the DiT and VAE on one SP-3 group, with SAM 3 on a 4th GPU; the frame-rate-optimized one moves the VAE onto its own GPU, taking the total to 5. SAM 3 runs on its own GPU throughout.

	Latency-optimized			**Frame-rate-optimized**		
GPU	DiT	VAE	SAM 3	DiT	VAE	SAM 3
G0	X	X		X		
G1	X	X		X		
G2	X	X		X		
G3			X		X	
G4						X

Cost model. Write $d(k)$ and $v(k)$ for the DiT and VAE stage latencies at SP degree k. WorldPlay is the pure actions $\rightarrow$ DiT $\rightarrow$ VAE pipeline analyzed in Appendix B.3, and its two deployment families instantiate that analysis. Latency follows the critical path through the DiT and then the VAE. Co-locating the two on a single SP-k group both shortens this path, since a higher SP degree fits within a given budget, and avoids a cross-group transfer, so it minimizes latency. Frame rate is instead set by the interval Δ between successive output chunks. Under co-location the DiT and VAE share the GPUs and run serially,

$$\Delta_{\text{co}} \;=\; d(k) + v(k),$$

while disaggregating them onto separate groups overlaps the VAE decode of one batch with the DiT of the next,

$$\Delta_{\text{dis}} \;=\; \max\big(d(k_D),\, v(k_V)\big),$$

bounded by the slower stage. Disaggregation buys this smaller interval at the cost of an inter-group transfer that lengthens the latency-binding path.

What drives the tradeoff. Two facts organize the numbers below. First, the DiT is the heavier stage at every SP degree ($d(k) > v(k)$), so under disaggregation the period $\Delta_{\text{dis}} = \max(d, v) = d$ hides the VAE decode behind the DiT. Second, both stages speed up only sublinearly with the SP degree: FLASHRT composes deployment strategies over the given operators and does not rewrite backend kernels (Section 6), so a higher SP degree shortens a stage without ever reaching linear speedup. Latency is the serial path $d + v$, so it is set by the SP degree of each stage and is minimized by co-location, which spends the whole budget raising that degree. A useful consequence is that co-location and disaggregation at the *same* per-stage SP degree reach nearly the same latency, differing only by disaggregation's small inter-group transfer; the two families otherwise trade latency against frame rate.

Table 5: Placement $\rho : \mathcal{V} \to \mathcal{R}$ for the latency- and frame-rate-optimized WorldPlay deployments on 2 GPUs. The latency-optimized deployment co-locates the DiT and VAE on one SP-2 group (short critical path); the frame-rate-optimized one disaggregates them onto separate GPUs and pipelines across batches (same tradeoff shown in Appendix B.3).

	Latency-optimized		**Frame-rate-optimized**	
GPU	DiT	VAE	DiT	VAE
G0	X	X	X	
G1	X	X		X

Table 6: Full set of WorldPlay video-world-model deployments found by the FLASHRT agent across GPU budgets (see Table 1). The 4- and 6-GPU deployments are analyzed in Appendix C.2.

# GPUs	**Deployment**	**Latency (ms) ↓**	**Frame rate (FPS) ↑**
1	Baseline (sequential)	869	20.4
2	FLASHRT (frame-rate-optimized)	625	**31.0**
	FLASHRT (latency-optimized)	**493**	25.5
4	FLASHRT (frame-rate-optimized)	494	**39.5**
	FLASHRT (latency-optimized)	**425**	29.7
6	FLASHRT (frame-rate + latency optimized)	447	**44.3**

Two GPUs. The latency-optimized deployment co-locates the DiT and VAE at SP-2, giving each stage both GPUs, and reaches the lower latency (493 ms) at 25.5 FPS. The frame-rate-optimized deployment instead runs the DiT and VAE on one GPU each (SP-1) and pipelines them, so its period $\max(d(1), v(1))$ lifts the frame rate to 31.0 FPS; its latency (625 ms) is higher, since dropping to SP-1 lengthens each stage on the serial path. Co-location thus affords the higher SP degree and wins latency, while disaggregation pipelines and wins frame rate. This is the choice analyzed in Appendix B.3.

Four GPUs. Both strategies move to a higher SP degree. The latency-optimized deployment co-locates the DiT and VAE at SP-4 and attains the lowest latency of any budget (425 ms); its co-located period $d(4) + v(4)$ also raises the frame rate to 29.7 FPS, a clear gain over the 2-GPU co-located point (25.5 FPS). The frame-rate-optimized deployment disaggregates the DiT and VAE at SP-2 each and pipelines them, reaching 39.5 FPS through the shorter period $\max(d(2), v(2))$. Its latency (494 ms) is essentially that of the 2-GPU co-located deployment (493 ms): both run each stage at SP-2, so the serial path $d(2) + v(2)$ is the same, and splitting the two stages onto separate GPU groups adds only a small transfer. This is the clean form of the tradeoff: at a matched SP degree, disaggregation costs almost nothing in latency and buys a markedly higher frame rate.

Six GPUs. The 6-GPU deployment disaggregates the DiT at SP-4 from the VAE at SP-2 and pipelines them, reaching the highest frame rate (44.3 FPS) at a latency (447 ms) close to the 4-GPU co-located minimum (425 ms). Because the pipeline is DiT-bound, its period $\max(d(4), v(2)) = d(4)$ is lowered only by scaling the DiT, so the two GPUs added over the 4-GPU pipeline go to the DiT group (SP-2 $\to$ SP-4) while the VAE stays at SP-2. Latency stays near the 4-GPU minimum because both deployments run the DiT at SP-4. Returns still diminish: with sublinear SP speedup, each further doubling of the DiT group shortens d by less, so frame rate grows more slowly than the GPU count.

C.3 VIDEO NARRATOR (LONGLIVE)

We use the LongLive video narrator of Section 5.3, in which an ASR model transcribes the user's spoken prompts while an autoregressive DiT and a streaming VAE render a continuous video stream. Table 8 reports the full set of deployments the agent found across GPU budgets, and Table 9 shows

Table 7: Placement ρ for the disaggregated (frame-rate-optimized) WorldPlay deployments as the GPU budget grows from 2 to 6. The DiT and VAE occupy separate groups and are pipelined across batches; the GPUs added from 4 to 6 go to the DiT group (SP-2 to SP-4) while the VAE stays at SP-2.

	2 GPU		**4 GPU**		**6 GPU**	
GPU	DiT	VAE	DiT	VAE	DiT	VAE
G0	X		X		X	
G1		X	X		X	
G2				X	X	
G3				X	X	
G4						X
G5						X

Table 8: Full set of LongLive video-narrator deployments found by the FLASHRT agent across GPU budgets (see Table 1). The 4- and 8-GPU deployments are analyzed in Appendix C.3.

# GPUs	**Deployment**	**Latency (ms) ↓**	**Frame rate (FPS) ↑**
1	Baseline (sequential)	674	25.8
2	FLASHRT (latency-optimized)	**512**	36.5
	FLASHRT (frame-rate-optimized)	630	**47.8**
4	FLASHRT (latency-optimized)	**430**	38.5
8	FLASHRT (frame-rate-optimized)	462	**67.4**

the node-to-GPU placements of the two multi-GPU deployments we analyze. We focus on the 4- and 8-GPU deployments and analyze how the additional GPUs affect latency and frame rate.

Cost model. Appendix C.2 analyzed the actions $\rightarrow$ DiT $\rightarrow$ VAE pipeline of the video world model; LongLive adds an ASR stage that transcribes each new spoken prompt. The period analysis there carries over unchanged to the DiT and VAE: co-location gives $\Delta_{\text{co}} = d(k) + v(k)$ and disaggregation gives $\Delta_{\text{dis}} = \max\big(d(k_D),\, v(k_V)\big)$, with frame rate set by $\text{FPS} \propto 1/\Delta$. The one addition is the ASR latency a on the time-to-first-output path,

$$L \;=\; a + d(k_D) + v(k_V).$$

Because the ASR fires only on a new spoken prompt and runs asynchronously, it enters L but not the period Δ, so it does not affect the frame-rate analysis.

The streaming VAE decode scales sublinearly: $v(k)$ flattens after a low SP degree. FLASHRT accommodates this through placement rather than by rewriting the VAE kernel, and this constraint shapes the LongLive deployments below.

Four GPUs. The latency-optimized 4-GPU deployment co-locates the DiT and VAE on a single SP-4 group, with the ASR sharing G0. Because latency is the serial sum $a + d(4) + v(4)$, driving both components to the largest SP degree the budget allows yields the shortest critical path, and this deployment attains the lowest latency at any budget (430 ms, $1.6\times$ below the baseline). Its frame rate is set by $\Delta_{\text{co}} = d(4) + v(4)$ and lands between the two 2-GPU deployments. It necessarily beats the co-located SP-2 point, since neither d nor v increases with the SP degree and so $d(4) + v(4) < d(2) + v(2)$ (38.5 vs. 36.5 FPS); but it does not beat the disaggregated 2-GPU pipeline, whose period is $\max(d(1), v(1))$ (47.8 FPS). Because the VAE saturates, the sum of two SP-4 latencies exceeds the maximum of two SP-1 latencies: the co-located deployment must pay the VAE term that the pipeline hides beneath the DiT. A higher SP degree alone therefore cannot match the frame rate a pipeline reaches with fewer GPUs.

Eight GPUs. The frame-rate-optimized 8-GPU deployment holds the DiT at SP-4 but moves the VAE to its own SP-3 group and gives the ASR a dedicated GPU, disaggregating all three stages.

Table 9: Placement ρ for the two multi-GPU LongLive deployments analyzed in Appendix C.3. The latency-optimized deployment co-locates the DiT and VAE on one SP-4 group; the frame-rate-optimized deployment disaggregates the DiT (SP-4), VAE (SP-3), and ASR onto separate groups so the VAE decode pipelines with denoising and transcription never blocks generation. In every deployment except the 8-GPU one the ASR shares G0 with the DiT.

	4 GPU (co-located)			**8 GPU (disagg.)**		
GPU	ASR	DiT	VAE	ASR	DiT	VAE
G0	X	X	X	X		
G1		X	X		X	
G2		X	X		X	
G3		X	X		X	
G4					X	
G5						X
G6						X
G7						X

Keeping the DiT at SP-4 fixes its critical-path term $d(4)$, so latency stays essentially at the 4-GPU value: $a+d(4)+v(3)$ is only marginally higher (462 vs. 430 ms), reflecting the VAE running at SP-3 rather than SP-4 and its output now crossing a group boundary. Frame rate, in contrast, improves sharply, because disaggregation replaces the co-located sum $d(4)+v(4)$ with the pipelined maximum $\max(d(4), v(3))$, lifting the frame rate to 67.4 FPS (2.6× the baseline) at nearly unchanged latency. The four GPUs added over the latency-optimized deployment are thus spent on disaggregation rather than on a higher SP degree: once the VAE has stopped scaling, converting the serial sum into a pipelined maximum is what buys frame rate, whereas latency keeps tracking the serial sum $a + d(k_D) + v(k_V)$ and is minimized by the SP degree. Latency and frame rate therefore respond to added GPUs through different mechanisms, so the deployment that is best for one is not best for the other.

C.4 Deployments on AMD MI355X

Table 10 lists the deployments the FlashRT agent found on AMD MI355X GPUs for all five applications.

D Independent Validation of Deployment Correctness

Beyond the agent's own checks (Section 4.3), we validate agent-generated deployments with a separate test harness, written independently from the agent sessions and never visible to any agent. It drives the reference and each deployment through the frontend's interface with the same fixed inputs and compares their outputs. Validation results for the main B200 deployments are shown in Table 11.

Controlling randomness. All backends use the same fixed seed: every stochastic stage is seeded per session or per request, and the one stage that samples without a seed (the conversational agent's LLM) decodes greedily. Correctness runs pin cuDNN and PyTorch Inductor to deterministic kernel selection. Inputs are delivered so that every processing step receives the same data regardless of the backend's speed.

Metrics and thresholds. A deployment passes if its outputs meet the following thresholds against the reference's:

- **Video:** the same number of frames, and a mean per-frame PSNR of at least 35 dB on 8-bit RGB frames.
- **Audio:** a length within one output frame of the reference's, and a mean absolute sample error of at most 2×10^{-3} for samples in $[-1, 1]$.

Table 10: Deployments generated by FLASHRT on AMD MI355X GPUs, re-optimized from scratch with the same reference implementations and GPU budgets used for B200. Latency is time-to-first-output; for Qwen3-Omni, throughput is the real-time factor (RTF).

Application	Deployment	# GPUs	Latency (s) ↓	Frame rate (FPS) ↑
LiveAvatar	Baseline (sequential)	1	101.17	–
	FLASHRT (streaming)	3	1.59	37.76
	FLASHRT (streaming + S2V PP + SP)	8	**1.47**	**78.61**
			Latency (s) ↓	**RTF $<$ 1**
Qwen3-Omni	Baseline (sequential)	1	29.66	✓
	vLLM-Omni	3	0.779	✓
	FLASHRT	3	**0.276**	✓
			Latency (s) ↓	**Frame rate (FPS) ↑**
Krea + SAM 3	Baseline (sequential)	1	2.393	5.40
	FLASHRT (latency-optimized)	4	**0.612**	7.93
	FLASHRT (frame-rate-optimized)	5	0.646	**8.90**
WorldPlay	Baseline (sequential)	1	1.102	15.6
	FLASHRT (latency-optimized)	2	**0.576**	19.8
	FLASHRT (frame-rate-optimized)	2	0.723	**29.5**
	FLASHRT (latency-optimized)	4	**0.320**	31.8
	FLASHRT (frame-rate-optimized)	8	0.326	**56.8**
LongLive	Baseline (sequential)	1	0.784	18.9
	FLASHRT (latency-optimized)	2	**0.660**	20.4
	FLASHRT (frame-rate-optimized)	2	0.907	**30.5**
	FLASHRT (latency-optimized)	4	**0.343**	38.8
	FLASHRT (frame-rate-optimized)	8	0.442	**56.5**

For the autoregressive video models that exhibit high error accumulation (WorldPlay and Krea-Realtime), the independent validation harness uses teacher forcing and diffusion timestep forcing, respectively, so that errors do not compound and the comparison gives a clear correctness signal.

E ADDITIONAL ANALYSIS OF THE AGENT HARNESS

This appendix provides more detailed analysis of the experiments conducted in Sections 5.5 and 5.6. For each study, we expand on the set of deployments the agent implemented, rather than only the best ones summarized in Sections 5.5 and 5.6. As stated previously, the results from these additional studies are based on running the FLASHRT agent on the face-to-face conversational agent application of Section 5.2 on an 8-GPU B200 node.

E.1 ABLATION

Table 12 lists the deployments produced by each ablated configuration. The Naive configuration is omitted because it terminates after a single deployment, the S2V output stream reported in Table 2.

The two configurations that lack the IR pass start by implementing the same strategy, namely streaming generated video to the output buffer while leaving the TTS $\rightarrow$ S2V edge blocking. Latency is therefore at least the time required to synthesize the full response audio, so both configurations sit above 10 s. The No IR configuration escapes this bound because, through the enforced loop, the agent continues exploring after its first working variant and identifies the remaining streaming edge; without the loop, the Naive configuration does not discover this streaming opportunity.

The No Loop configuration also discovers the full streaming variant, but this happens directly during the IR phase, rather than as a result of structured iteration (as in the No IR configuration). Importantly, through the IR pass, the No Loop configuration also identifies the S2V pipeline-parallel

Table 11: Independent validation of FLASHRT deployments against the sequential references. A PSNR of ∞ or an MAE of 0 denotes bit-identical output.

Application	Deployment	# GPUs	Video PSNR (dB) ↑	Audio MAE ↓	Pass
LiveAvatar	FLASHRT (streaming)	2	∞	0	✓
	FLASHRT (streaming + S2V PP)	6	39.7	0	✓
Qwen3-Omni	FLASHRT	3	–	3.1×10^{-4}	✓
Krea + SAM 3	FLASHRT (parallel)	2	∞	–	✓
	FLASHRT (latency-optimized)	4	51.8	–	✓
	FLASHRT (frame-rate-optimized)	5	51.8	–	✓
WorldPlay	FLASHRT (latency-optimized)	2	55.6	–	✓
	FLASHRT (frame-rate-optimized)	2	75.0	–	✓
	FLASHRT (latency-optimized)	4	41.6	–	✓
	FLASHRT (frame-rate-optimized)	4	57.0	–	✓
	FLASHRT (frame-rate + latency optimized)	6	42.1	–	✓
LongLive	FLASHRT (latency-optimized)	2	62.9	–	✓
	FLASHRT (frame-rate-optimized)	2	∞	–	✓
	FLASHRT (latency-optimized)	4	69.4	–	✓
	FLASHRT (frame-rate-optimized)	8	62.0	–	✓

Table 12: All deployments measured by each ablated configuration on the face-to-face conversational agent. The pipeline-parallel-only variant produced by the full FLASHRT agent does not stream, so its output is produced as a single burst and no sustained frame rate is defined for it.

Configuration	Deployment	# GPUs	Latency (s) ↓	Frame rate (FPS) ↑
No IR	S2V output streaming	2	10.44	30.94
	Full streaming	2	1.68	30.67
No Loop	Full streaming	2	1.67	30.93
	Full streaming + S2V PP	6	2.37	44.69
FLASHRT	Full streaming	2	1.69	30.92
	S2V PP	6	36.29	–
	Full streaming + S2V PP	6	**1.72**	**133.48**

strategy, which both the Naive and No IR configurations (which did not have an IR pass) failed to identify. The No Loop configuration proceeds directly to implement a combined variant that composes full streaming with S2V pipeline parallelism; however, the agent only prototypes pipeline parallelism within a single process, where a synchronization event on any stage stalls all stages; the agent reports a suboptimal 44.69 FPS and terminates at this prototype rather than iterating further. In contrast, the FLASHRT configuration measures the pipeline-parallel strategy in isolation first; this variant carries no streaming and therefore has poor latency (36.29 s), but implementing this strategy in isolation allows the agent to develop a multi-process coordination mechanism that exchanges activations over NCCL, avoiding single-process blocking behavior. Then, the agent systematically composes that implementation with streaming to produce the final deployment, which improves on the No Loop result on both axes at the same GPU count.

E.2 SENSITIVITY TO THE UNDERLYING MODEL

Table 14 reports the two deployments produced by Codex with GPT-5.6 Sol that are relevant to the comparison in Table 13a.

The 6-GPU deployment applies pipeline parallelism but streams only the generated video to the output buffer, leaving the TTS $\rightarrow$ S2V edge blocking, so it reaches nearly the same frame rate as

Table 13: Robustness of the harness on the face-to-face conversational agent application. In both studies, the agent is instructed to optimize the application given a full 8-GPU node.

(a) Underlying coding agent and model

Agent	Best deployment	# GPUs	Latency (s) ↓	Frame rate (FPS) ↑
Claude Code + Opus 4.8	Full streaming + S2V PP	6	1.72	133.48
Codex + GPT-5.6 Sol	Full streaming + S2V PP + VAE SP	8	1.77	131.98

(b) Independent runs of the same configuration

Run	Best deployment	# GPUs	Latency (s) ↓	Frame rate (FPS) ↑
1	Full streaming + S2V PP	6	1.72	133.48
2	Full streaming + S2V PP	8	1.63	133.07
3	Full streaming + S2V PP	8	1.78	125.30

Table 14: Deployments produced by Codex with GPT-5.6 Sol on the face-to-face conversational agent application.

Deployment	# GPUs	Latency (s) ↓	Frame rate (FPS) ↑
S2V output streaming + S2V PP	6	10.30	132.49
Full streaming + S2V PP + VAE SP	8	**1.77**	131.98

the final deployment while retaining the latency of a non-streaming pipeline. Adding the TTS → S2V edge recovers the latency reduction. The final deployment also shards the VAE across three ranks, which accounts for its use of 8 GPUs: four for the denoising steps, three for the VAE, and one shared by the ASR, LLM, and TTS. This does not change either metric: frame rate under pipeline parallelism is set by the slowest stage, and a single denoising step is slower than the single-rank VAE, so accelerating the VAE cannot raise the ceiling; latency is likewise unaffected, as the reduced VAE time is offset by the coordination overhead of an additional process group.

E.3 Consistency across runs

Table 15 reports the representative trajectory of each of the three independent runs summarized in Table 13b.

Table 15: Deployments measured in each of three independent runs on the face-to-face conversational agent application. All three runs propose the same three deployments in the same order and select the composed deployment as the best.

	Run 1		Run 2		Run 3	
Deployment	Latency (s)	FPS	Latency (s)	FPS	Latency (s)	FPS
Full streaming	1.69	30.92	1.64	30.75	1.67	30.87
S2V PP	36.29	–	43.10	–	37.99	–
Full streaming + S2V PP	1.72	133.48	1.63	133.07	1.78	125.30

The three runs also agree on how they reach the final deployment: each isolates streaming, then isolates pipeline parallelism, then composes the two. This behavior is directly induced by the FlashRT workflow, since the IR pass helps the agent identify both axes and the loop requires each one to be measured systematically before they are combined. The runs differ only in placement. Run 1 pairs the ASR with the VAE and the LLM with the TTS for a total of 6 GPUs, while runs 2 and 3 give each of these components its own GPU for a total of 8. Because the ASR, LLM, and TTS complete before the S2V pipeline begins to produce frames, co-locating them changes neither the critical path nor the steady-state period, and the three runs land within 0.15 s and 8.2 FPS of one another. Since

the agent is given a full 8-GPU budget in all three runs, it is free to disaggregate these components, even though the final deployment does not require it.

F AGENT COST

Table 16 reports the token usage, monetary cost, and wall-clock time of the five B200 agent sessions that produced the deployments of Sections 5.2 and 5.3. Costs are computed at current API pricing. Because the GPUs must remain allocated to the agent for the duration of the session, the corresponding compute cost is the E2E time multiplied by the GPU budget.

Table 16: Token usage, cost, and wall-clock time for the agent session that produced each application's deployments. Cache reads dominate token usage because the agent repeatedly revisits the reference implementation, the IR, and its own benchmark records across the session.

Application	**Input**	**5 min cache**	**1 hr cache**	**Cache read**	**Output**	**Cost**	**E2E (hr)**
LiveAvatar	44,500	155,511	1,759,657	246,157,708	1,097,179	$166.98	4.99
Qwen3-Omni	31,835	88,267	1,074,911	248,093,750	701,940	$151.92	4.92
Krea-Realtime + SAM 3	22,646	51,096	747,557	110,365,086	485,026	$74.53	2.64
WorldPlay	28,722	0	1,003,877	222,005,520	631,620	$136.98	3.51
LongLive	46,251	0	972,942	204,161,864	588,008	$126.74	5.43

At $75 to $167 per application, an agent session is inexpensive relative to the expert engineering time it replaces, and the comparison extends past the direct cost. Writing efficient serving code for a new real-time pipeline otherwise depends on the availability of engineers with the relevant systems expertise, which does not scale with the number of applications; instead, agent-driven deployment scales with compute that can be provisioned per application.

These costs reflect a search in which the agent implements and measures nearly every candidate it queues. It discards a queued candidate only in limited circumstances when a candidate is deemed unviable, and the agent always provides a recorded justification. For example, we observe a small number of instances where the agent attempts a deployment that uses an invalid SP degree that does not evenly divide the number of latent frames processed by a video model; in such cases, the agent identifies the error and discards the candidate, then replaces it with a valid candidate that uses a viable SP degree.