FlashRT: Agent Harness for Guiding Agents to Deploy Real-Time Multimodal Applications

Krish Agarwal[1], Zhuoming Chen[1], Yanyuan Qin[2], Zhenyu Gu[2], Atri Rudra[3], Beidi Chen[1]

{krisha2, zhuominc, beidic}@andrew.cmu.edu, {yanyuan.qin, zhenyu.gu}@amd.com, atri@buffalo.edu

[1]Carnegie Mellon University, [2]AMD, [3]University at Buffalo

Real-time multimodal applications, including 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 achieving high performance on a new application requires hand-crafting an efficient implementation. We present FlashRT, an agent harness that guides coding agents to lift simple developer-written reference implementations into optimized multi-GPU deployments that flexibly weigh target metrics like latency and throughput. Using a new **chain-of-program** paradigm, FlashRT directs a generic coding agent through a multi-pass transformation process where an agent transforms the reference into an intermediate representation (IR) to capture data dependencies and persistent-state scopes, validates this IR via a sequential interpreter, and performs static analyses to identify candidate transformations. Then, the agent iteratively implements, verifies, and benchmarks each candidate under a measurement-gated optimization loop to produce effective deployments that span different hardware budgets. Across various applications, including video world models and multimodal LLMs, FlashRT converts reference implementations into highly efficient deployments, delivering up to ∼**70**× latency reduction and **2.8**× throughput improvement on NVIDIA B200 GPUs. On AMD MI355X GPUs, FlashRT matches the peak latency reduction while increasing peak throughput improvement to **3.6**×, demonstrating that agent-driven optimization can be more scalable on platforms with less mature expert optimization. In fact, for Qwen3-Omni text-to-audio inference, FlashRT reduces response latency by **65%** compared to the expert vLLM-Omni implementation on AMD MI355X.

Github: https://github.com/Infini-AI-Lab/FlashRT

Website: https://infini-ai-lab.github.io/flashrt-blog

1 Introduction

As strong generative models emerge across various modalities, there is increasing adoption of real-time multimodal applications (Gao et al., 2026; Ball et al., 2025; Défossez et al., 2024; Kodaira et al., 2023), naturally followed by 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 highly complex pipelines, each with its own deployment considerations, e.g., 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. These diverse applications cannot be served effectively under a single system that hosts only a limited set of deployment policies. Instead, new system efforts are required for each application, typically through manual implementation, but **steady growth in the number of new and diverse applications makes hand-crafting efficient deployments unscalable**.

Existing systems try to automate deployment to reduce human effort but fall short for three key reasons. **(1) Limited deployment strategies**: existing multi-modal frameworks like vLLM-Omni (Yin et al., 2026) and Cornserve (Ma et al., 2025) provide useful abstractions for multimodal serving, but they restrict to static deployment policies (e.g., full disaggregation around high-level stage boundaries and full co-location within a stage), limiting efficacy on general latency or throughput targets. **(2) Limited workload coverage:** auto-parallelism frameworks like FlexFlow (Jia et al., 2018), GSPMD (Xu et al., 2021), Alpa (Zheng et al.,

2022), and Unity (Unger et al., 2022) can automatically choose sharding, placement, and pipelining, but they specialize to one fixed workload and cannot generalize to diverse multimodal applications. **(3) Optimization granularity:** existing compilation frameworks, like TVM (Chen et al., 2018) and TASO (Jia et al., 2019), though highly effective in optimizing deep learning workloads, are out of scope as they target only operator-level improvements. Moreover, such methods rely on a well-defined optimization granularity level, which does not exist for arbitrary multimodal applications.

An ideal serving system for multimodal applications should allow users to flexibly write intuitive, unoptimized, single-GPU implementations and automatically derive specialized system infrastructure for each application's individual needs. Such a system would directly address the key limitations of previous works by providing unconstrained deployment scope, workload flexibility, and adaptive optimization granularity. However, this is challenging to accomplish with rule-based systems since the deployment problem is NP-hard, as we elaborate on this in Section 3. This is especially problematic if deployment is performed at the finest granularity, e.g., kernel-level, which would result in too large a search space. However, any higher level of granularity is not even properly defined for a rule-based system, as the optimal granularity may be heterogeneous between different pipeline components.

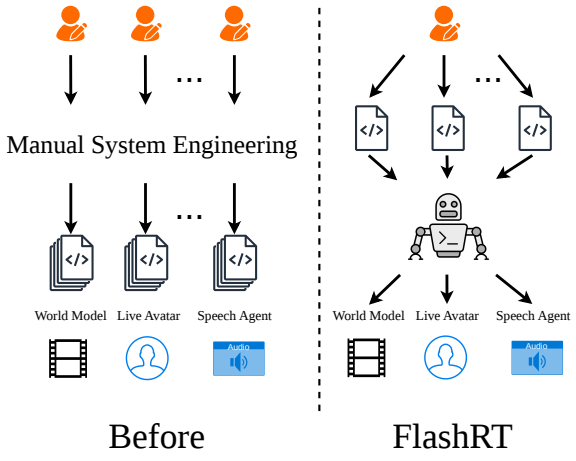


Figure 1 Previously, efficiently serving diverse multimodal applications required manual systems effort per application. FlashRT solves this by guiding a coding agent to design efficient deployments.

Fortunately, AI agents (Yang et al., 2024; Wang et al., 2024) have shown significant progress in automatically synthesizing code from simple specifications and performing low-level optimizations (Ouyang et al., 2025; Liao et al., 2025; 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**, as illustrated in Figure 1. In particular, **agents can employ higher-level reasoning to organize a target pipeline at adaptive, heterogeneous granularities, reducing the deployment search space while enabling discovery of highly efficient implementations**. However, naively prompting an agent to directly transform a baseline implementation into an efficient deployment is not effective.

- **Chain-of-program paradigm**: we identify that current agents cannot effectively perform direct translation in a single step. inspired by existing approaches like chain-of-thought prompting for LLMs (Wei et al., 2023) as well as compiler frameworks that progressively lower frontend code into a backend, we observe that agents are much more effective when required to first convert the user reference to a structured intermediate representation (IR), perform static analysis, and then translate the IR into effective deployments.
- **Application-grounded validation loop**: not only can agents fail to properly explore the diverse search space, but they may also easily produce incorrect solutions. We find that agents deliver stronger guarantees by enforcing a self-driven loop where the agent explores, implements, and measures various hypotheses; importantly, the agent can design an application-specific test harness that drives simulated user input through the backend implementation, which is critical for grounding validation and benchmarking results in the application-specific user experience.

Combining these insights, we introduce FlashRT, a system for serving multimodal applications that guides a generic coding agent through a structured workflow to automatically convert highly flexible user implementations into efficient deployments. Across a range of multimodal applications, we show that FlashRT synthesizes low-latency deployments that are competitive with expert-designed systems. Compared to third-party serving systems and other expert-written systems, FlashRT generates deployments that can flexibly tradeoff between time-to-first-output (TTFO) latency and system throughput. On the applications we evaluate, FlashRT successfully converts baseline implementations into highly efficient deployments, reaching $\sim$**70**$\times$ latency reduction and **3.6**$\times$ throughput gain. Importantly, this agentic system alone can generalize to highly diverse applications on both NVIDIA B200 and AMD MI355X hardware, while still requiring no human intervention

beyond supplying a reference implementation. As such, FlashRT proves to be an effective framework that can substantially reduce manual systems effort.

2 Related Work

Real-time multimodal serving. Many new application compose heterogeneous models into real-time pipelines. Examples include speech-to-speech voice agents (Défossez et al., 2024; Xu et al., 2025a; Seamless Communication et al., 2023), audio-driven avatar generation (Zhang et al., 2024; Tian et al., 2024; Xu et al., 2024; Peng et al., 2024; Huang et al., 2025), real-time video world modeling (Kodaira et al., 2023; Ball et al., 2025; Sun et al., 2025; Gao et al., 2026), and embodied vision-language-action policies (Black et al., 2024). Multimodal serving systems, including vLLM-Omni (Yin et al., 2026), Cornserve (Ma et al., 2025), and ModServe (Qiu et al., 2025), can express each application as a graph over a small fixed set of well-defined stages. While this provides clean abstractions for new applications, it leads to restrictive deployment policies that are not always optimal, such as required inter-stage disaggregation.

Auto-parallelism and ML compilers. Auto-parallelism systems such as FlexFlow (Jia et al., 2018), 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 and do not transfer to inference pipelines composed of heterogeneous models with disjoint runtimes. Pipeline-parallel methods like PipeDream (Narayanan et al., 2019) provide useful primitives but assume a single homogeneous training graph. Operator-level compilers such as TVM (Chen et al., 2018), TASO (Jia et al., 2019), and Halide (Ragan-Kelley et al., 2013) aggressively rewrite kernel-level computation, but their granularity for optimization does not touch important considerations like device placement or intra-model stage overlap.

Coding agents for performance optimization. Agent systems for software engineering (Yang et al., 2024; Wang et al., 2024; Hong et al., 2024; Shinn et al., 2023) have demonstrated that LLMs paired with environment feedback can navigate large codebases and iteratively repair their own outputs. For GPU kernel generation and optimization (Ouyang et al., 2025; Liao et al., 2025; Wei et al., 2025; Lange et al., 2025; Baronio et al., 2025; Li et al., 2025b,a; Yang et al., 2025b), recent work has explored using agents with traditional compiler pass selection (Tan et al., 2025), scientific-discovery search (Novikov et al., 2025; Romera-Paredes et al., 2024), and reward design (Ma et al., 2024). FlashRT differs from these in thatit optimizes is end-to-end multimodal deployment across multiple model runtimes.

3 Problem Formulation

FlashRT takes as input a synchronous single-GPU reference implementation P_{ref} of a multimodal application. P_{ref} defines the application semantics and any persistent state scopes (e.g., caches or streaming buffers). FlashRT returns a multi-GPU deployment P_{dep} that may use disaggregation, co-location, streaming, and intra-model parallelism while preserving the behavior of P_{ref}.

We model an application as a task graph $G = (\mathcal{V}, E)$, where each $v \in \mathcal{V}$ is a computation region. The application processes input in batches indexed by j, and the graph induces one operation instance (j, v) per batch. Each edge $(u, v, \lambda) \in E \subseteq \mathcal{V} \times \mathcal{V} \times \{0, 1\}$ denotes a data or state dependence, with $\lambda = 0$ an intra-batch (data) dependence of (j, v) on (j, u) and $\lambda = 1$ a cross-batch (state) dependence of (j, v) on the previous instance $(j - 1, u)$ (e.g., a KV cache that batch j inherits from batch $j - 1$). A deployment specifies a placement

$$\rho : \mathcal{V} \to \mathcal{R},$$

where each resource $r \in \mathcal{R}$ may be a single- or multi-GPU group, and a non-preemptive schedule

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

where $\varnothing$ indicates the resource is idle. Let $S_{j,v}$ be the start time of instance (j, v). Each edge $(u, v, \lambda) \in E$ imposes

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

whenever the predecessor instance $(j - \lambda, u)$ exists. Here $\tau_u(\rho)$ is the placement-dependent execution time of u, and $\kappa_{u,v}(\rho)$ captures synchronization or transfer cost. The schedule must also satisfy resource capacity, i.e., each resource non-preemptively executes at most one assigned instance at a time.

The objective is to design a deployment that minimizes application-level serving metrics, such as response latency or throughput, subject to correctness constraints. More detailed formulations are provided in Appendix A.1, which formalizes these two metrics and shows that latency is governed by the critical path from input to output, while sustainable throughput is governed by the most heavily loaded resource. Appendices A.3 and A.4 apply these bounds to representative streaming pipelines, and both reveal the same latency-throughput tradeoff: co-locating operations on a shared resource shortens the critical path, whereas disaggregating them onto separate resources and pipelining across batches reduces the load on the most heavily loaded resource, so no single deployment is best for both targets. This deployment problem is hard even in restricted cases. With a single batch, no dependencies, no communication costs, and a makespan objective (minimizing the time to finish all operations), it reduces to classical non-preemptive multiprocessor scheduling, which is NP-hard (Lenstra et al., 1977). Multimodal serving strictly generalizes this setting: operations have data and state dependencies, execution times depend on placement and parallelization, and communication costs depend on co-location. A proof is provided in Appendix A.2. Because execution and communication costs couple the deployment decisions and finding an optimum is NP-hard, an efficient deployment can neither be prescribed by a fixed policy nor solved exactly; it must instead be discovered per application by jointly reasoning about program structure, resource assignment, scheduling, and parallelism.

4 FlashRT

Converting a developer-written sequential reference for a multimodal pipeline into an efficient multi-GPU deployment requires reasoning about each application's specific structure, which is a task naturally suited to a coding agent. Unlike rule-based serving systems and auto-parallelism compilers, a coding agent can read an arbitrary reference, reason about its structure, and produce a deployment specialized to the application at hand. However, naively handing the agent a reference and directly tasking it to produce an optimized version can fail. In Section 4.1, we present a case study analysis for several failure modes when performing the naive agent deployment. Then in Sections 4.2 and 4.3, we describe how the FlashRT framework addresses these failures, namely 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 generates a talking-head video response to a user's spoken query. The application pipeline consists of the following stages: **1.** The user's speech input is processed by an automatic speech recognition (ASR) module to transcribe the audio into text. **2.** The transcribed text prompt is fed into a large language model (LLM) to generate a textual response. **3.** The generated text response is passed through a text-to-speech (TTS) model to synthesize response audio. **4.** The synthesized audio is provided to a sound-to-video (S2V) generative model, which produces a video of a person speaking the response. **5.** The generated audio and video streams are synchronized and returned to the user as the final response.

Figure 2a shows an example of what a user base implementation (the input to the system), would look like. Concretely, we use Qwen3-ASR (Shi et al., 2026) for ASR, Qwen3-4B (Yang et al., 2025a) as the LLM, Qwen3-TTS (Hu et al., 2026) for TTS, and LiveAvatar (Huang et al., 2025) for S2V.

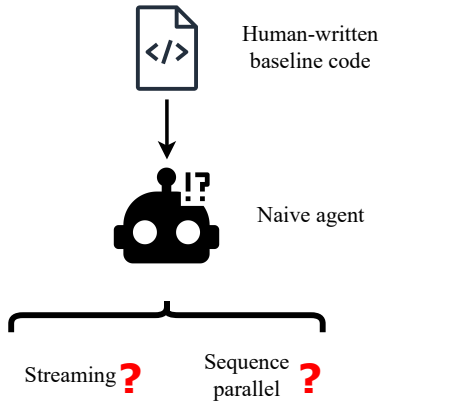
We consider the case where the response latency budget is flexible, and the agent must instead optimize for throughput (i.e., frame rate). This objective will naturally prefer deployments that use inter-module streaming and pipline parallelism. We manually identify two axes for optimization:

- *Application level*: once the LLM response is received, the TTS model can iteratively produce chunks of audio output rather than producing the whole audio at once; the LiveAvatar S2V model is autoregressive and also processes input audio in chunks, meaning that the TTS model can be streamed into the LiveAvatar model. Also, generated chunks can be displayed while future chunks are still being processed, meaning the synced audio + video output can be streamed into a display buffer.
- *Model level*: the LiveAvatar model is a DiT-based video generation model, and it consists of DiT and VAE

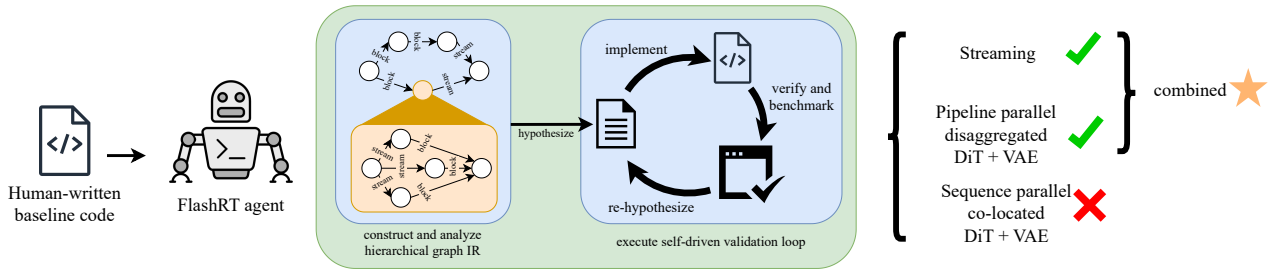
```python
def run_conversational_agent(user_prompt_audio):
    user_prompt_text = run_asr(user_prompt_audio, HF)
    response_text = run_llm(user_prompt_text, vLLM)
    response_audio = concat([
        audio_chunk for audio_chunk in
        run_tts(response_text, vLLM_Omni)
    ])
    response_video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

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

(a) Example user base implementation inputted to FlashRT.



(b) The naive agent will inconsistently find some optimization axes but not all, and it will not consider composing strategies.



(c) FlashRT transforms the baseline implementation into an IR to perform analysis. Then, it enters a self-driven validation loop to test various hypotheses. This allows the agent to consistently find diverse optimization strategies and consider compositions of strategies that improve the specified target (throughput in this case).

Figure 2 Overall view of different agent workflows. (a) shows an example input to the ideal system. (b) shows the result of naively tasking an agent to optimize a base implementation. (c) shows how failure cases are addressed by FlashRT's agent workflow.

components; since the VAE does not share any internal state (e.g., KV cache) with the DiT, the DiT can be run on an input batch in parallel with the VAE on the previous batch on separate GPUs. Additionally, LiveAvatar is trained such that each diffusion step uses an independent KV cache, so the different steps can be deployed in a pipeline parallel manner (i.e., one GPU per diffusion step). We formally analyze the latency–throughput implications of these two model-level choices in Appendix A.3 (co-locating versus disaggregating the DiT and VAE) and Appendix A.4 (sequence versus pipeline parallelism across diffusion steps).

When we provide the agent with a synchronous, single GPU baseline implementation of this pipeline and naively instruct it to find a more efficient deployment targeting throughput, we observe two key failure modes (also highlighted in Figure 2b):

- **While the agent recognizes application-level streaming opportunities (e.g., TTS → S2V), it fails to recognize any model-level pipeline parallelism opportunities.** Instead, the agent chooses to co-locate all diffusion steps as well as the VAE on the same set of GPUs and use sequence parallelism to distribute compute across GPUs; this helps reduce response latency but does not improve throughput. Ideally, the agent would recognize pipeline parallelism opportunities based on independence of KV caches between diffusion steps and no shared state with the VAE.
- **The agent does not explore diverse optimization patterns in any single run.** Instead, in some runs the agent will focus solely on application-level streaming opportunities; meanwhile, in other runs the agent will focus solely on S2V model-level sequence parallelism. Ideally, the agent should discover multiple optimization

axes in a single run, and it should also compose different axes to form the most effective deployment.

These failure modes are highly problematic because resolving them requires explicit human intervention to help the agent identify optimization opportunities it previously missed. To resolve this, Sections 4.2 and 4.3 present how we design the FlashRT agent workflow so that these failure cases are eliminated while maintaining full agent autonomy. These design principles are also visualized in Figure 2c.

4.2 Chain-of-Program Paradigm for Hierarchical Planning

We observe that failure cases where the naive agent fails to recognize important optimization opportunities (e.g., S2V pipeline parallelism) stem from the fact that **the agent is expected to directly transform a baseline implementation to an effective deployment in one monolithic step**. This differs from how a human would approach the problem: humans employ high-level reasoning to understand end-to-end workflows at varying levels of granularity to design effective implementations. This is analogous to widely adopted chain-of-thought (Wei et al., 2023) mechanisms that allow language models to reason about a prompt before providing a final response. It is also what motivates the use of intermediate representations in conventional compilers, as it is simpler (and more scalable) to design a compiler that progressively lowers a frontend implementation into a backend over multiple passes. We observe that, by instructing the agent to first convert the baseline implementation to a properly structured intermediate representation (IR) and then convert the IR to a set of candidate deployments, the agent is more more robust at discovering all possible axes for efficient implementation. We denote this paradigm as **chain-of-program**.

IR design. The IR represents a pipeline as one or more hierarchical, directed acyclic graphs. Specifically, the IR represents an application workflow using nodes to encapsulate self-contained operations and edges to denote data dependencies. These graphs instantiate the task graph $G = (\mathcal{V}, E)$ of Section 3, with nodes as the computation regions $\mathcal{V}$ and dataflow edges as the intra-batch data dependencies; the cross-batch state dependencies are supplied by the node annotations below. The IR exposes three structural properties of the pipeline that together drive the subsequent agentic analysis.

- **Graph hierarchy**: The agent must generate an IR graph in a hierarchical manner, e.g., a top-level graph modeling the overall application workflow and multiple nested graphs specifying intra-module computations. First, allowing a hierarchical graph IR allows the baseline implementation to remain flexible, e.g., it can compose custom model implementations with as opaque third-party serving systems. Second, by requiring the agent to model both high-level application graphs and nested intra-module graphs for exposed model runtimes, the agent is forced to consider various levels of granularity for optimization, preventing it from focusing only on high-level application workflow. This hierarchy also sets the granularity of the nodes $\mathcal{V}$: exposing finer nodes (e.g., a model's internal stages) is what lets a deployment place or parallelize them separately, which a flat, module-level graph would preclude.
- **Node-level state annotation:** For each IR node, the agent is required to specify the set of persistent state reads and writes corresponding to the node's operations (this excludes transient internal state). These reads and writes recover the cross-batch (state) dependencies of Section 3, i.e., the $\lambda = 1$ edges: two nodes that touch the same persistent state (e.g., a KV cache) are joined by such an edge and must be ordered across batches, whereas nodes with disjoint state carry no such edge and can be disaggregated and executed in parallel. The agent is therefore deterred from missing key opportunities related to disaggregation and pipeline parallelism.
- **Edge-level streaming annotation:** For each data dependency edge between nodes, the agent is required to explicitly mark the edge as either "blocking" or "streaming". For example, an LLM $\rightarrow$ TTS edge from the example in Section 4.1 would be "blocking", while a TTS $\rightarrow$ S2V edge would be "streaming". Similar to the previous point, these explicit annotations deter the agent from missing critical design opportunities: streaming edges mark where a consumer can begin before its producer finishes, so the two can overlap on separate resources via cross-batch pipelining, whereas blocking edges must serialize and therefore favor co-location.

Agentic IR analysis. In addition to having the agent translate the user implementation to an IR, the agent is instructed to perform static analysis on the resulting IR. This analysis critically includes several static tools

that we provide to the agent; specifically, these tools analyze the agent's generated IR graph and the data dependencies that originate from graph edges and/or shared persistent state to statically identify parallelizable nodes and streaming opportunities; this tool allows the agent's analysis workflow to be more reliable and deterministic in surfacing key deployment design principles that it can try to implement. As a side note, we also provide the agent with a IR interpreter that takes the agent's generated IR graph and executes it sequentially based on topological order; the agent can execute the baseline implementation directly and its generated IR graph with the interpreter on the same sample inputs to compare outputs and can confirm the correctness of the IR it generates.

4.3 Application-grounded validation loop

We find that even with a well-defined candidate space surfaced by IR analysis, an unguided agent can still focus on only one axis of optimization and may not consider combinations of different strategies. This failure case is shown in Section 4.1 when the agent focuses only on application-level streaming or model-level multi-GPU deployment, without considering combinations of these methods. Many existing works on agentic kernel generation (Ouyang et al., 2025; Liao et al., 2025; Wei et al., 2025) and local code optimization (Shypula et al., 2024) introduce self-driven validation loops where the agent continuously iterates on a solution, verifies correctness, and benchmarks performance. Critically, while verifying correctness and benchmarking performance is well-defined in these use cases, these are highly application-specific for multi-modal pipelines. Therefore, we propose a self-driven validation loop where, not only does the agent proposes hypotheses and implement them in every iteration, but the agent also designs a test harness that is grounded in the user application experience to evaluate key metrics.

Verification and benchmarking. In FlashRT's self-driven loop, each iteration requires the agent to verify correctness of and benchmark its implementation. Concretely, we observe that, although the agent is naturally unable to interact with an application backend via a user-facing frontend, the agent can in fact implement test cases that write simulated input to the backend's input buffers and read corresponding output from its output buffers. This is analogous to how a user-facing frontend would write user interactions to the backend's input buffers and read from its output buffers to display results to the user, effectively grounding the agents experiments in the true user experience. Through these test cases, the agent can validate correctness by driving the same simulated inputs to the baseline backend and the agent's generated backend; furthermore, the agent can measure end-to-end latency and throughput by collecting timing statistics when writing to input buffers and reading from output buffers. By requiring the agent to perform this structured verification, the agent's self-driven validation loop is strengthened since its hypotheses are grounded in application-specific measurements that mimic real user interactions.

Iteration step. In FlashRT, the agent not only performs self-driven iteration on its solution, but it facilitates the loop using a self-evolving variant queue that expands the diversity of strategies explored and improves the agent's robustness. First, the variant queue is initialized from the IR analyses. Each iteration begins with a hypothesis, i.e. naming the transformation analysis result that justifies its legality and the bottleneck it targets. The agent then implements this in an isolated environment and verifies element-wise output equivalence of its implementation against the baseline implementation over a set of sample inputs; if errors are discovered, the agent proceeds to debugging its implementation. After producing a correct implementation, the agent benchmarks its implementation on sample inputs and uses the result to update 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. Through this workflow, the agent is required to fully validate/measure its hypotheses and consider both diversity of implementation design as well as potential for strategy compositions through the reevaluation mechanism.

5 Experiments

We evaluate FlashRT on a diverse set of real-time multimodal applications and answer three questions: **(1)** Can the agent generate highly efficient implementations that deliver high throughput at low time-to-first-output (TTFO) latency? **(2)** Is the agent able to generalize to a variety of real-time multi-modal applications? **(3)** Does the agent framework generalize across hardware? We organize the experiments as follows:

Table 1 Comparison between the sequential user baseline and several deployments proposed by the FlashRT agent for face-to-face conversational agent application. Frame rate is omitted for the sequential baseline because the full output video is generated offline and becomes playable only after completion.

# GPUs	Deployment	Latency (s) ↓	Frame rate (FPS) ↑
1	Baseline (sequential, no streaming)	107.92	–
1	FlashRT (streaming)	3.94	16.26
3	FlashRT (streaming + disaggregation)	**1.57**	40.88
8	FlashRT (streaming + disagg. + S2V PP)	1.66	**173.67**

- In Section 5.2, we present a case study, matching the application studied in Section 4.1, of how the agent is able to compose several strategies to achieve ∼**70**× latency reduction from the baseline and high theoretical frame rate.
- In Section 5.3, we evaluate the agent on four additional applications, each with different compute structures. We find that the agent continues to deliver principled deployments that can jointly reduce latency while boosting throughput.
- In Section 5.4, we execute the agent pipeline separately on AMD MI355X for the same set of applications and find that it recovers the same class of efficient deployments, and the same latency and frame-rate tradeoffs, that it finds on NVIDIA B200.

5.1 Setup

Hardware. Unless otherwise noted, experiments are run on a single node with 8 NVIDIA B200 GPUs; Section 5.4 additionally evaluates the full pipeline on a node with 8 AMD MI355X GPUs. The number of GPUs allocated to the agent varies across experiments (between 1 and 8), reflecting the different scale at which we evaluate each pipeline. We report the GPU budget for each result.

Agent configuration. For every experiment, we use Anthropic Claude Code (Anthropic, 2026a) with Claude Opus 4.8 (Anthropic, 2026b), using adaptive thinking with `effort=max`. The agent runs in an isolated workspace with Claude Code's Auto permission mode enabled, which permits unattended tool use subject to Claude Code's automated action classifier. Each session starts from scratch with no memory of previous runs and no visibility into runs on other applications. The starting prompt only specifies the application to optimize and the GPU budget; it provides no hints about possible deployment strategies. Each application is supplied with a synchronous, single-GPU reference implementation that performs no inter-component streaming, and the agent is responsible for self-discovering all deployment patterns.

5.2 Case Study: Face-to-Face Conversational Agent

We evaluate the FlashRT agent on the same face-to-face conversational agent application presented in Section 4.1. We provide the agent a sequential baseline, structured like the pseudocode shown in Figure 2a.

Our results are presented in Table 1, and the multi-GPU deployments' node-to-GPU placements are shown in Table 2. We find these results to be quite impressive:

- Even with one GPU, FlashRT reduces TTFO latency by **27.4**×. This is accomplished by starting S2V inference in intermediate TTS audio chunks as soon as they become available rather than waiting for the full TTS audio to be available. However, TTS and S2V processing remains in contention on one GPU, resulting in low frame rate.
- To address low frame rate, the agent expands to using more than one GPU to disaggregate the TTS engine from the S2V engine, i.e., place them on different GPUs. By combining this with per-chunk TTS → S2V processing, the TTS and S2V models can run in parallel, reducing inter-batch latency to provide a **2.5**× frame rate improvement. This also translates to additional 2.5× latency reduction.
- The agent goes even further to use 8 GPUs: in this deployment it disaggregates TTS from S2V, and it also uses pipline parallism for the S2V (which was noted as a legal strategy for LiveAvatar S2V in Section 4.1, and

Table 2 Placement ρ for the LiveAvatar conversational-agent deployments (conventions as in Table 7). The sequential baseline and streaming deployment (omitted) place all nodes on one GPU. At 8 GPUs, the S2V DiT's four denoising steps (DiT_1–DiT_4) are pipelined one per GPU (the same per-step decomposition of Appendix A.4).

	3 GPU (disagg.)					**8 GPU (disagg. + 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				
G2			X							X			
G3											X		
G4												X	
G5													X
G6							X						
G7								X					

Table 3 Comparison between FlashRT and the hand-engineered vLLM-Omni deployment for serving Qwen3-Omni with 2 GPUs. FlashRT achieves lower latency while maintaining a Real-Time Factor (RTF) below 1, enabling real-time audio generation after the initial response latency.

Deployment	**# GPUs**	**Latency (s)** ↓	**RTF** < 1
Sequential (no streaming)	1	42.713	✓
vLLM-Omni	3	0.433	✓
FlashRT	3	**0.323**	✓

which Appendix A.4 shows sustains at least as high a frame rate as sequence parallelism when sequence-parallel speedup is sublinear). This provides an even further **4.2**× frame rate improvement with minimal impact to latency.

Ultimately, FlashRT is able to design a highly efficient implementation that reduces latency from the user's sequential baseline by ∼**70**× while sustaining an extremely high theoretical frame rate that allows real-time playability.

5.3 Generalization to other applications

We evaluate on a set of four additional applications and demonstrate that FlashRT can generalize to diverse settings while still producing efficient deployments.

5.3.1 Qwen3-Omni

Qwen3-Omni (Xu et al., 2025b) is a natively multimodal model that responds to a user prompt through three components: a Thinker LLM that produces a text response, a Talker LLM that converts the response into audio codec tokens, and a Vocoder that synthesizes the codec tokens into audio. vLLM-Omni (Yin et al., 2026) explicitly implements disaggregation and inter-stage streaming for this model. We provide the FlashRT agent sequential baseline that independently serves each component and coordinates them synchronously, and we find that the FlashRT agent automatically discovers the disaggregation structure and streaming logic on its own.

As shown in Table 3, FlashRT achieves **25% latency reduction** compared to vLLM-Omni. This is achieved by using lighter-weight inter-component data transfer compared to vLLM-Omni's more generalized implementation. Table 3 also measures Real-Time Factor (RTF), the ratio between E2E generation time and length of generated audio; FlashRT maintains RTF < 1, meaning that its output remains real-time while still delivering lower latency.

Table 4 Comparison of video background editor deployments discovered by the agent against the sequential baseline.

Deployment	# GPUs	Latency (ms) ↓	Frame rate (FPS) ↑
Baseline (sequential)	1	1715	6.82
FlashRT (parallel)	2	**1014**	**11.54**
FlashRT (latency-optimized)	4	**491**	17.18
FlashRT (frame-rate-optimized)	5	517	**19.41**

Table 5 Placement ρ for the video background editor deployments. The latency-optimized deployment co-locates DiT and VAE on 4 GPUs; the frame-rate-optimized one disaggregates the VAE onto a 5th GPU. SAM 3 runs on its own GPU.

	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

5.3.2 Video Background Editor (Krea-Realtime + SAM 3)

In this application, the user provides a continuous webcam video stream. The video is processed in two paths: Krea-Realtime (Millon, 2025), an autoregressive video model that supports video-to-video (V2V) generation, performs style transfer on the stream; Segment Anything Model (SAM) 3 (Carion et al., 2026) segments the user's body per frame. The two outputs are combined by replacing the V2V output with the original webcam pixels at all body locations, producing a video where only the background is restyled. The V2V and SAM 3 paths are independent until the final composition step and can therefore execute in parallel; within Krea-Realtime, the DiT can be sequence-parallelized; like WorldPlay, the DiT and VAE can be co-located or disaggregated.

Results are reported in Table 4, with the placements visualized in Table 5. Importantly, the agent automatically chooses to deploy SAM 3 on a separate GPU from the V2V model. The agent then produces a latency-optimized and a frame-rate-optimized variant: **(1) latency optimized**: the DiT and VAE are co-located with with SP-3, reducing latency by **3.5**× over the baseline; **(2) frame-rate-optimized** the agent uses SP-3 for DiT and places the VAE on a separate GPU to enable pipeline parallelism, resulting in **2.8**× higher frame-rate compared to the baseline.

5.3.3 Video World Model (WorldPlay)

WorldPlay is an autoregressive video world model: the user issues keyboard actions (WASD or arrow keys), and the model renders these actions as movement in a generated video stream. We use HY-WorldPlay 5B (Sun et al., 2025), which decomposes into an autoregressive DiT and a streaming VAE. The DiT and VAE share no persistent state, so they can be disaggregated on separate GPUs and pipelined across batches; alternatively, they can be co-located and sequence-parallelized across a shared group.

Results are reported in Table 6, with the node-to-GPU placements of the two 2-GPU deployments visualized in Table 7. We find that the agent employs both disaggregation of DiT and VAE as well as co-location with sequence parallelism to design deployments that tackle both latency and output frame rate. For higher frame rate, the agent chooses to disaggregate the DiT and VAE: this enables pipelining and executing them in parallel on separate GPUs to reduce inter-batch latency. For lower latency, the agent chooses to instead co-locate the DiT and VAE on the same set of GPUs: this enables a higher sequence parallel degree within the same GPU budget to reduce the critical path latency. Notably, this is also the same as reducing the inter-batch latency, so frame rate also improves. These two choices instantiate Appendix A.3, which proves that for an actions → DiT

Table 6 Deployment comparisons for WorldPlay at 1 and 2 GPUs. The agent leverages disaggregation, co-location, and sequence parallelism to construct deployments that improve latency and/or frame rate.

# 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

Table 7 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 A.3).

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

→ VAE pipeline, co-location minimizes the latency-binding critical path while disaggregation can sustain a higher frame rate.

The agent proposes further variants that scale these strategies to larger GPU budgets, reaching up to **51%** lower latency and **2.2**× the baseline frame rate; we present these deployments and their analysis in Appendix B.1.

5.3.4 `Video Narrator (LongLive)`

In this application, the user narrates a video in real time: the user speaks prompts that an automatic speech recognition (ASR) model transcribes into text, and an autoregressive video model renders a continuous video stream that updates to reflect each new prompt. We use LongLive-2.0-5B (Chen et al., 2026) for video generation, which decomposes into an autoregressive DiT and a streaming VAE. The ASR shares no persistent state with the video model and supplies a new prompt only when the user speaks, so it can be placed on a separate GPU and run asynchronously to overlap transcription with generation. Like WorldPlay, the DiT can be sequence-parallelized, and the DiT and VAE can be co-located and sequence-parallelized or disaggregated onto separate GPUs and pipelined across batches, with the VAE additionally tiled across GPUs.

Table 8 Deployment comparisons for the LongLive video narrator at 1 and 2 GPUs. At a given GPU budget the agent offers a latency-optimized deployment (co-located DiT and VAE at a higher sequence-parallel degree) and a frame-rate-optimized deployment (DiT and VAE disaggregated and pipelined across GPUs), trading the two targets against each other.

# 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**

Results are reported in Table 8. As in WorldPlay, at a given GPU budget the agent offers a latency-optimized and a frame-rate-optimized deployment that trade the two targets against each other. For lower latency it co-locates the DiT and VAE, which affords a higher sequence-parallel degree within the budget and shortens the serial per-chunk denoising critical path; for higher frame rate it instead disaggregates the DiT and VAE onto separate GPUs so that the VAE decode of one chunk overlaps the DiT computation of the next. At 2 GPUs this is the choice between co-located SP-2 and a disaggregated DiT/VAE pipeline, which trades **23**% higher latency for **31**% higher frame rate. The agent proposes further variants that scale these strategies to larger GPU budgets, reaching **1.6**× lower latency and **2.6**× the baseline frame rate; we present these deployments and their analysis in Appendix B.2.

Table 9 Deployments discovered by the FlashRT agent on AMD MI355X for all five applications, re-optimized from scratch under the same references and GPU budgets used for B200.

(a) Face-to-face conversational agent (LiveAvatar)

# GPUs	Deployment	Latency (s) ↓	Frame rate (FPS) ↑
1	Baseline (sequential, no streaming)	101.17	–
2	FlashRT (streaming)	10.58	34.2
3	FlashRT (streaming + disaggregation)	1.59	37.8
8	FlashRT (streaming + disagg. + S2V PP)	**1.47**	**78.5**

(b) Multimodal LLM (Qwen3-Omni)

# GPUs	Deployment	Latency (s) ↓	RTF < 1
1	Sequential (no streaming)	29.66	✓
3	vLLM-Omni	0.779	✓
3	FlashRT	**0.276**	✓

(c) Video background editor (Krea-Realtime + SAM 3)

# GPUs	Deployment	Latency (ms) ↓	Frame rate (FPS) ↑
1	Baseline (sequential)	2393	5.40
4	FlashRT (latency-optimized)	**612**	7.93
5	FlashRT (frame-rate-optimized)	646	**8.90**

(d) Video world model (WorldPlay)

# GPUs	Deployment	Latency (ms) ↓	Frame rate (FPS) ↑
1	Baseline (sequential)	1102	15.6
2	FlashRT (latency-optimized)	**576**	19.8
2	FlashRT (frame-rate-optimized)	723	**29.5**
4	FlashRT (latency-optimized)	**320**	31.8
8	FlashRT (frame-rate-optimized)	326	**56.8**

(e) Video narrator (LongLive)

# GPUs	Deployment	Latency (ms) ↓	Frame rate (FPS) ↑
1	Baseline (sequential)	784	18.9
2	FlashRT (latency-optimized)	**660**	20.4
2	FlashRT (frame-rate-optimized)	907	**30.5**
4	FlashRT (latency-optimized)	**343**	38.8
8	FlashRT (frame-rate-optimized)	442	**56.5**

5.4 Generalization across hardware

The experiments so far run on NVIDIA B200 GPUs. To test whether FlashRT generalizes across accelerators, we separately execute the full agent pipeline on AMD MI355X for the same set of applications. Other than the hardware, the protocol matches Section 5.1 in every respect: the agent is given the same reference implementations, the same GPU budgets, and no hardware-specific hints, and it optimizes each application from scratch (the agent is provided no information on the B200 runs). Table 9 reports the deployments it discovers across all five applications.

On MI355X, FlashRT reduces latency by up to $\sim$**70**$\times$ and improves throughput by up to **3.6**$\times$, while beating the hand-engineered vLLM-Omni baseline on Qwen3-Omni by **65%**. Across all five applications the agent

recovers the same deployment families it found on B200 and reproduces the central latency/frame-rate tradeoff, in which co-location minimizes latency and disaggregation raises frame rate. On WorldPlay, at 2 GPUs co-location reaches the lower latency (576 ms) and disaggregation the higher frame rate (29.5 FPS), the same split the agent makes on B200. On Qwen3-Omni the agent again outperforms the vLLM-Omni deployment (0.276 s versus 0.779 s) while keeping RTF below 1 (Table 9b), meaning the output streams in real-time.

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 (320 ms and 343 ms). The one application whose frame rate does not scale as on B200 is the video background editor, where the pipeline is bound by the SAM 3 segmentation path on MI355X and disaggregating the VAE yields little additional throughput. On MI355X the agent produces an efficient deployment for each application and makes the same latency and frame-rate tradeoffs it does on B200.

6 Conclusion

In this paper, we present FlashRT, an agent harness for using a coding agent to flexibly deploy real-time, multimodal interactive applications. Our agentic solution is motivated by the intractability of deriving efficient deployments from arbitrary application specifications and the promise of recent coding agents at using higher-level reasoning to perform effective code synthesis and optimization. We discover that, by instilling an agent with a properly structured workflow—specifically consisting of an IR conversion step to make data dependencies and streaming behaviour explicit, as well as a self-driven validation loop to systematically encourage diverse and principled agentic exploration—agents are capable of converting simple, human-authored application implementations into highly efficient deployments. Our results demonstrate that FlashRT is able to flexibly optimize for various targets such as throughput or latency, and the deployments it generates significantly outperform simple handwritten baselines while also being generally on par or more performant than solutions proposed by other serving systems or expert engineers. A current limitation of our system is that we have not integrated LLM kernel optimization agents. We leave this as future work to enable a broader range of optimization opportunities. Furthermore, our experiments only test one agent configuration, and future work should test how the FlashRT framework behaves under modified agent settings (e.g., different model, lower reasoning depth, modified agent scaffolding, etc.).

7 Acknowledgements

We are grateful to AMD and NVIDIA for access to computing resources. This work was partially supported by Google Research Award, Google ML & System Junior Faculty Award, Amazon Research Award, Fireworks AI, Intel, Li Auto, Moffett AI, and CMU CyLab Seed funding. This material is also based upon work supported by the National Science Foundation under Grant Nos. CCF-2504353 and CCF-2247014, and by IARPA. Any opinions, findings, conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Anthropic. Claude code overview. https://code.claude.com/docs/en/overview, 2026a. Documentation for Claude Code; accessed 2026-05-04.

Anthropic. Claude opus 4.7. https://www.anthropic.com/news/claude-opus-4-7, April 2026b. Model ID: claude-opus-4-7; accessed 2026-05-04.

Phil Ball, Jakob Bauer, Frank Belletti, Bethanie Brownfield, Ariel Ephrat, Shlomi Fruchter, Agrim Gupta, Kristian Holsheimer, Aleks 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, and Jessica Yung. Genie 3: A new frontier for world models. Google DeepMind Blog, 2025. https://deepmind.google/blog/genie-3-a-new-frontier-for-world-models/.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Cherniavskii, Janice Wang, and Fotis Iliopoulos. Kevin: Multi-turn RL for generating CUDA kernels. *arXiv preprint arXiv:2507.11948*, 2025.

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. *arXiv preprint arXiv:2410.24164*, 2024.

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, 2026. https://arxiv.org/abs/2511.16719.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018.

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. Longlive-2.0: An nvfp4 parallel infrastructure for long video generation, 2026. https://arxiv.org/abs/2605.18739.

Alexandre Défossez, Laurent Mazare, Manu Orsini, Amélie Royer, Patrick Pérez, Hervé Jégou, Edouard Grave, and Neil Zeghidour. Moshi: A speech-text foundation model for real-time dialogue. *arXiv preprint arXiv:2410.00037*, 2024.

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.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024.

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. 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. *arXiv preprint arXiv:2512.04677*, 2025.

Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks, 2018. https://arxiv.org/abs/1807.05358.

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)*, 2019.

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. *arXiv preprint arXiv:2312.12491*, 2023.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977. doi: 10.1016/S0167-5060(08)70743-X.

Shangzhan Li, Zefan Wang, Ye Tian, Yuhan Sun, Yiyi Wu, Bowen Liu, Xiao Yu, Tianyu Liu, and Yuxiang Cheng. AutoTriton: Automatic Triton programming with reinforcement learning in LLMs. *arXiv preprint arXiv:2507.05687*, 2025a.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Wang, Fei Wu, Xianzhi Han, and Tianwei Yu. CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025b.

Gang Liao, Yavuz Yetim, Ruichao Xiao, Zewei Jiang, Raghav Boinepalli, Sheela Yadawad, Liyuan Li, Nathan Yan, Ajit Mathews, Chunqiang Tang, Carole-Jean Wu, and Gaoxiang Liu. KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at Meta. *arXiv preprint arXiv:2512.23236*, 2025.

Jeff J. Ma, Jae-Won Chung, Jisang Ahn, Yizhuo Liang, Akshay Jajoo, Myungjin Lee, and Mosharaf Chowdhury. Cornserve: Efficiently serving any-to-any multimodal models. *arXiv preprint arXiv:2512.14098*, 2025.

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. In *International Conference on Learning Representations (ICLR)*, 2024.

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

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)*, 2019.

Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergei Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? *arXiv preprint arXiv:2502.10517*, 2025.

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 (CVPR)*, 2024.

Haoran Qiu, Anish Biswas, Zihan Zhao, Kunal Mahajan, Inigo Goiri Han, Chetan Bansal, Rodrigo Fonseca, Ramachandran Iyer, and Ricardo Bianchini. ModServe: Scalable and resource-efficient large multimodal model serving. *arXiv preprint arXiv:2502.00937*, 2025.

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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 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, et al. Seamless: Multilingual expressive and streaming speech translation. *arXiv preprint arXiv:2312.05187*, 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. https://arxiv.org/abs/2601.21337.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *International Conference on Learning Representations (ICLR)*, 2024.

Wenqiang Sun, Haiyu Zhang, Haoyuan Wang, Junta Wu, Zehan 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.

Hongyu Tan, Jiawei Wang, Yuxiang Liu, Wenliang Sun, and Tianyi Zhao. AwareCompiler: Agentic context-aware compiler optimization via a synergistic knowledge-data driven framework. *arXiv preprint arXiv:2510.11759*, 2025.

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 *European Conference on Computer Vision (ECCV)*, 2024.

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)*, pages 267–284, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. 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. *arXiv preprint arXiv:2407.16741*, 2024.

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. *arXiv preprint arXiv:2509.07506*, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. https://arxiv.org/abs/2201.11903.

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. *arXiv preprint arXiv:2503.20215*, 2025a.

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. 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. *arXiv preprint arXiv:2406.08801*, 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. 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, 2025a. 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 *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Zijian Yang, Junfeng Liu, Yifan Liu, Yuxuan Sun, Hao Wang, and Tianlong Chen. CudaForge: An agent framework with hardware feedback for CUDA kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025b.

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)*, pages 521–538, 2022.

Yue Zhang, Minhao Liu, Zhaokang Chen, Bin Wu, Yubin Zeng, Chao Zhan, Yingjie He, Junxin Huang, and Wenjiang Zhou. MuseTalk: Real-time high quality lip synchronization with latent space inpainting. *arXiv preprint arXiv:2410.10122*, 2024.

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)*, 2022.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, 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 *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–210, 2024.

Appendix

A 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.

A.1 Setup

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

A.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 $\Delta = \frac{T}{N}$. 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 chunk k of batch j is output 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.

A.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 each chunk k.

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.

A.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 A.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.

A.1.4 Two key bounds

Lemma A.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 Theorem A.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 A.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.

A.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 Theorem A.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

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

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.

A.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 Sections 5.3.2 to 5.3.4. 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 A.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 Theorems A.1 and A.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 A.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

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

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 Theorem A.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 Theorem A.3 to each resource, together with the inter-batch self-edges, yields the stated expression for $T_{\min}(\rho_{\text{pipe}}^{X,Y})$. The intra-batch DiT-to-VAE lag $\kappa_{D,V,0}(\rho_{\text{pipe}}^{X,Y})$ 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:

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

□

Conclusion. Propositions A.4 and A.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.

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

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

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

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

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

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:

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

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 A.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 Theorems A.1 and A.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 A.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

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

we have

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

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

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

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 Theorem A.3,

$$T \geq T_Q$$

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 $T_Q \leq nT_Q^{SP}(n)$. 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 A.6 and A.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 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 provides extended results for two of these applications, the video world model (WorldPlay) of Section 5.3.3 and the video narrator (LongLive) of Section 5.3.4. 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.

B.1 Video World Model (WorldPlay)

We use the WorldPlay video world model of Section 5.3.3, in which the user's keyboard actions drive an autoregressive DiT and a streaming VAE that render a continuous video stream. Table 10 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 sequence-parallel 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 11 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.

Cost model. Write $d(k)$ and $v(k)$ for the DiT and VAE stage latencies at sequence-parallel degree k. WorldPlay is the pure actions → DiT → VAE pipeline analyzed in Appendix A.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,

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

Table 10 Full set of WorldPlay video-world-model deployments found by the FlashRT agent across GPU budgets (cf. Table 6). The 4- and 6-GPU deployments are analyzed in Appendix B.1.

# 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**

Table 11 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

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 sequence-parallel 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.

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 A.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

Table 12 Full set of LongLive video-narrator deployments found by the FlashRT agent across GPU budgets (cf. Table 8). The 4- and 8-GPU deployments are analyzed in Appendix B.2.

# 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**

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 → 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.

B.2 Video Narrator (LongLive)

We use the LongLive video narrator of Section 5.3.4, 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 12 reports the full set of deployments the agent found across GPU budgets, and Table 13 shows 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 B.1 analyzed the actions → DiT → 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 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 sequence-parallel degree. FlashRT accommodates this through placement rather than by rewriting the VAE kernel, and it 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× 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

Table 13 Placement ρ for the two multi-GPU LongLive deployments analyzed in Appendix B.2. 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

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. 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.