M*: A Modular, Extensible Serving System for Multimodal Models

Atindra Jha[1,*] **Naomi Sagan**[1,*] **Keisuke Kamahori**[2,†] **Irmak Sivgin**[1,†]
Rohan Sanda[1] **Steven Gao**[2] **Mark Horowitz**[1] **Luke Zettlemoyer**[2]
Olivia Hsu[1,3] **Jure Leskovec**[1,‡] **Baris Kasikci**[2,‡] **Stephanie Wang**[2,‡]

[1]Stanford University [2]University of Washington [3]Carnegie Mellon University
[*]Co-first authors [†]Second authors [‡]Equal advising Correspondence to atindra@cs.stanford.edu

Abstract

We are entering a new era of *composite model architectures*, from omni models to vision-language-action policies and world models. Their components differ in structure (encoders, decoders, transformers with standard or linear attention), in computation (autoregressive prefill and decode, diffusion and flow, convolution-heavy codecs), and in how they exchange data (loop-backs of tokens and latents, encoder embeddings passed to LLMs, streaming between concurrently running components). However, existing model serving frameworks were built on narrow assumptions about model structure, making them ill-suited to accommodate this diversity. Here we present M*, a universal serving system for efficient serving of composite AI models. M* represents models as dataflow graphs, processing requests spanning diverse modalities and tasks as traversals over these graphs. The core insight is a modular abstraction that supports arbitrary composition of model components, flexible placement onto a physical cluster, and model-agnostic optimizations within a distributed runtime. We call this abstraction the **Walk Graph** and show how it can concisely capture composite models from a broad range of families. We instantiate M* on representative models and compare against released versions of vLLM-Omni and SGLang-Omni. On Qwen3-Omni text-to-speech, M* delivers 2.2× and 3.1× their audio throughput at batch size 16 and sustains the highest request rate under a first-audio SLO. On BAGEL, it serves image generation, editing, and understanding from one deployment, with 15% to 46% higher image-to-text throughput and, under mixed online load, 1.3 to 1.4× the SLO attainment of vLLM-Omni's best two-GPU stage split and 2.5 to 4× its three-GPU deployment. It also accelerates V-JEPA 2-AC robotic-planning rollouts by up to 6× over the native implementation. Thus, our work paves the road towards more efficient serving of complex models with minimal developer effort.

1 Introduction

AI is entering a new era of *composite model architectures*: multimodal models built from structurally distinct components, including vision encoders, transformer backbones, diffusion and flow heads, audio codecs, and action generators. Unlike earlier models with relatively fixed execution structures, these components are composed and executed in patterns that vary across inputs and tasks. This new generation includes unified multimodal models (UMMs), omni models, speech language models (SpeechLMs), vision-language-action models (VLAs), and world models [13, 51, 10, 7, 4]. Tasks span image and video understanding and generation, real-time speech interaction, robot interaction, and world prediction. Despite their differences in modality and task, these architectures share a key property: *inference no longer reduces to a single autoregressive forward loop*.

The diverse execution structures of multimodal models create requirements that current LLM serving stacks [27, 58] do not cleanly capture. In text-only LLMs, all data takes essentially the same path

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

through a simple autoregressive (AR) loop. In modern multimodal models, different modalities and tasks trigger different execution paths through the same heterogeneous model. For example, in UMMs such as BAGEL [13], image generation and image understanding pass data through different components. Other models contain long non-AR loops, such as diffusion transformers (DiTs) [36] or rectified flow [30, 15] for image generation and variable-horizon world-model rollouts [4]. They may also contain internal parallelism, such as the condition branches in classifier-free guidance (CFG) [21] or the pipelined Thinker–Talker architecture in Qwen3-Omni [51]. These patterns are becoming the standard structure of multimodal models rather than exceptions layered onto token-centric ones.

LLM serving systems [27, 58] have been adapted to models that attach a non-text encoder to a language backbone [5, 39], but they still face a fundamental abstraction mismatch with the patterns above. Recent work addresses this gap with an intermediate abstraction, a fixed chain [53] or DAG [40] of "stages" that each capture one or more model components. Stages still cannot capture non-AR loops across stages, parallelism internal to a stage, or different tasks taking different paths through the same model (§2). The system therefore misses parallelism across components and request-specific execution, and placement at stage granularity rules out independent scaling of individual components.

This work addresses the problem of building an efficient multimodal serving system that enables day-zero support for the next generation of composite models. Our key insight is that, despite the diversity of model architectures, *every multimodal model is a dataflow graph of heterogeneous components*, and every user request executes as a *walk* of components within the graph. We design a flexible intermediate abstraction based on this idea to decouple the model architecture from the system runtime, enabling: (1) the capture of diverse model architectures, and (2) the support for flexible placement of different components to maximize hardware utilization, while (3) achieving the same or better performance than custom-built serving engines.

Thus, we present M*, a universal multimodal serving system. After the model author declares their model architecture as a computation graph and a set of graph walks, the deployer instantiates the model by declaring a mapping of model components to physical GPU ranks. Then, the runtime is responsible for all physical execution, such as component disaggregation, request scheduling, request batching, tensor transport, and tensor streaming. The runtime also integrates well-established optimizations, including paged attention [27], CUDA-graph capture, and continuous batching [54], as well as state-of-the-art modality-specific optimizations.

By decoupling the model architecture from the system runtime, M* enables broad support for composite multimodal models spanning text, image, video, audio, and robot actions. M* maintains state-of-the-art per-component performance and further improves efficiency because *the system execution mirrors the model's component graph*. Graphs of components with control flow are not new (§G). Our contribution is the granularity and the contract. Nodes are sub-components of one model that share weights and state, and data-dependent control flow is confined to walk boundaries, so the runtime knows each walk's component sequence ahead of time and can batch, capture, and place across it. We demonstrate these capabilities by instantiating M* on five representative composite models: BAGEL, Qwen3-Omni, $\pi_{0.5}$, V-JEPA 2, and Orpheus. We evaluate against released versions of vLLM-Omni [53] and SGLang-Omni [40], against VoxServe, against Meta's native V-JEPA 2 implementation, and in a scoped setting against Cornserve. On BAGEL [13], M* serves text-to-image generation, image editing, and image understanding from one deployment. Its single-request generation latency modestly beats vLLM-Omni's best deployment per workload ($1.07\times$ to $1.12\times$ at matched CFG-parallel placement), its image-understanding throughput is 15% to 46% higher, and under a mixed online workload it sustains 1.3 to $1.4\times$ the SLO attainment of vLLM-Omni's best two-GPU stage split and 2.5 to $4\times$ its three-GPU deployment. On Qwen3-Omni text-to-speech, M* has the lowest RTF at every batch size, $2.2\times$ and $3.1\times$ the audio throughput of vLLM-Omni and SGLang-Omni at batch size 16, and the highest sustainable arrival rate under a first-audio SLO. On Orpheus-TTS [10], M* outperforms VoxServe [25], a speech-optimized serving system, by 14% in RTF at batch size 16 and 19% to 55% in throughput. On V-JEPA 2 [4], M* accelerates robotic-planning rollouts by up to $6\times$ over the native implementation. Ablations attribute the largest share of the Qwen3-Omni and V-JEPA 2 gains to one mechanism, loop-body CUDA-graph capture, and the BAGEL gains to that capture together with continuous batching, both applied by the Walk Graph uniformly to a text-decode loop, an autoregressive talker, and a world-model rollout.

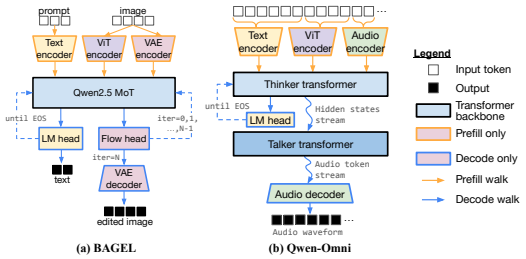

Figure 1: Example model architectures of (a) a UMM (BAGEL [13]) and (b) Omni model (Qwen3-Omni [51]). Composite models are structurally diverse, yet representable as computation graphs.

2 Background and Motivation

2.1 The Era of Composite Multimodal Models

Unlike text-only LLMs, recent multimodal models exhibit substantial architectural diversity: heterogeneous components for consuming and producing data across modalities, connected in complex structures. Five representative families exemplify the structural patterns that we target.

1. **Unified multimodal models (UMMs)** handle multiple vision tasks with a shared Transformer backbone and encoders/decoders for text and visual latents [13, 11, 49]: BAGEL [13] handles vision understanding, image generation, and image editing with a shared Mixture-of-Transformers (MoT [29]) component, each task using a different combination of encoders/decoders and transformer weights (Figure 1a).
2. **Speech language models (SpeechLMs)** pair an autoregressive Transformer backbone with neural audio codec models for text-to-speech or speech-to-speech [10, 14, 23]; the codec is invoked at architecture-specific intervals, and streaming applications must produce audio in real time [25].
3. **Omni models** support any-to-any modalities, including real-time speech. Qwen2.5-Omni and Qwen3-Omni [50, 51] combine two Transformers in a Thinker–Talker topology: the Thinker produces text and hidden states from inputs in any modality, and the Talker converts those states into speech codec tokens that an audio codec decoder turns into waveforms (Figure 1b).
4. **Vision-language-action models (VLAs)** produce a robot action from an image and a text instruction through a text encoder, a ViT encoder, a Transformer backbone, and an action decoder [7, 47].
5. **World models** pair a video encoder with a latent predictor that rolls forward over a variable horizon, e.g. for robotic planning [4, 19, 1, 2].

2.2 Composite Models are Computation Graphs

Composite models pose three concrete challenges:

(C1) **Architectural diversity.** Multimodal models have multiple execution paths depending on input modality. In UMMs, text-to-text chat, text-to-image generation, image-to-text understanding, and image-to-image editing use different subsets of components with distinct computation patterns, and many patterns involve non-AR loops (classes 1, 4, and 5 in Section 2.1).

(C2) **Performant modularity.** HuggingFace Transformers [48] offers flexibility but sacrifices efficiency, while specialized systems such as vLLM [27] for text and VoxServe [25] for speech reach high performance through optimizations that do not generalize across modalities.

(C3) **Physical topology.** Heterogeneous components execute in sequence, in a pipeline, or in parallel, with tensors batched or streamed between them over NVLink or InfiniBand, so flexibility in placement and data transport is critical for end-to-end performance.

Why Existing Systems Fall Short. vLLM [27] and SGLang [58] are highly optimized for AR text generation but treat multimodal inputs as prefill-time encoder add-ons. They have no first-class support for non-AR loops, parallel execution of nodes within a walk, or cross-component streaming. vLLM-Omni [53] and SGLang-Omni [40] extend them to DAGs of stages glued together with explicit data transfer functions. This fits two- or three-stage thinker–talker pipelines (Figure 1b), but neither exposes loops or parallel composition of stages. Per-model glue code is therefore needed for diffusion loops, fan-out in CFG, and custom streaming policies.

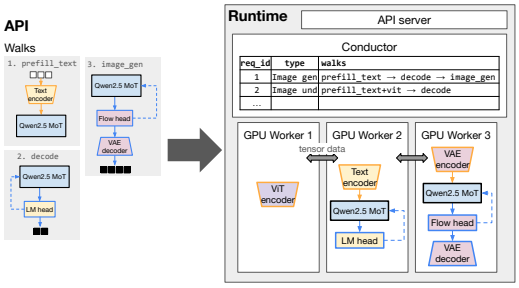

Figure 2: M* at a glance. **Left**: The model author defines the model as a computation graph (Figure 1) and a series of graph walks. **Right**: The runtime (Section 3.3) places the subgraphs of components on different GPU workers based on a user-specified placement.

Computation Graphs in M*. Despite this diversity, every model in Section 2.1 has the same structural form: a directed graph of heterogeneous *components* such as encoders, decoders, and transformer backbones (Figure 1). Intermediate tensors flow along the edges, possibly in a streaming fashion, and each request traverses the graph over a small number of *walks*. M* addresses C1 by treating multimodal inference as graph execution, with components composed in arbitrary loops, chains, and parallel branches (Section 3.1). It addresses C2 by "compiling" that abstraction down to a runtime whose scheduler and per-component engines integrate state-of-the-art optimizations across modalities (Section 3.3). It addresses C3 with user-defined physical placements (Section 3.2) and data-movement policies between components (Section 3.1).

3 The Walk Graph

M* is built on a single contract where a model is a directed *computation graph*, a request is a series of *Walks* of the graph, and the runtime is the executor of the graph. This contract is the **Walk Graph**, a model computation graph with a finite set of named Walks. We define the Walk Graph in Section 3.1, enumerate the capabilities it unlocks in Section 3.2, and describe the corresponding runtime in Section 3.3.

Table 3 in the appendix compares the Walk Graph with existing multimodal serving abstractions [53, 40, 12] along nine axes. vLLM-Omni and SGLang-Omni are flat DAGs of engine-level stages, with prefill and decode as the only execution paths, loops internal to a stage, and placement per stage. Cornserve is a task graph recorded per request over per-component executors, with static control flow and placement per executor. Each corresponds to a restricted subset of the Walk Graph, whose nodes are model components composed by `Sequential`, `Parallel`, `Loop`, and streaming edges, with flexible execution paths and placement per component and Walk.

3.1 API

A model is declared as a tuple (G, W) where $G = (N, E)$ is a directed *computation graph* of nodes and edges, and $W = \{w_1, \ldots, w_n\}$ is a finite set of named *Walks*. Each Walk is a labeled subgraph of G corresponding to one phase of model behavior, e.g., `prefill_text` is the Walk that prefills the text tokens of the prompt in Figure 2. Each request is a series of Walks, e.g., image understanding is [`prefill_text` $\rightarrow$ `prefill_vit` $\rightarrow$ `decode`]. A per-model state machine picks each request's next Walk from the request's modalities and the outputs of the current Walk. The model author provides only (G, W) and the state machine, and executing requests is the job of M*'s runtime.

Four composable primitives. The computation graph G and the Walks w_i are built from two types: `GraphNode`, a unit of computation that fires when its required inputs arrive, and `GraphEdge`, a tensor flowing from one node to another. Nodes are composed with four primitives (Appendix B):

- `Sequential`: a chain of subgraphs where the outputs of one feed into the next.
- `Parallel`: a fan-out of children that may execute concurrently.
- `Loop`: bounded iteration with per-iteration and accumulated output channels.
- `DynamicLoop`: `Loop` with per-request early-exit, such as for end-of-sequence (EOS) in AR models or rollout horizon in world models.

Streaming edges and chunk policies. In streaming-output models (e.g., real-time speech generation), the producer emits one tensor at a time and the consumer must accumulate a "chunk" before firing. We denote this with `StreamingGraphEdge`, parameterized by a `ChunkPolicy` that decides when the consumer has enough input to fire; three policies suffice for the models in our evaluation: (i) `FixedChunkPolicy(`K`)` fires every K items; (ii) `SlidingWindowChunkPolicy(`W`,`S`)` fires once the buffer holds W items and advances by S; (iii) `LeftContextChunkPolicy(`C`,`L`)` prepends L frames of left context to each C-frame chunk. New policies can be added without changes to the system.

Example: BAGEL. We make the abstraction concrete with BAGEL [13] (Figure 1a), a unified multimodal model that handles image understanding, image generation, and image editing with one Mixture-of-Transformers [29] backbone. Appendix A shows examples of other models.

Listing 1: The image-generation Walk of BAGEL with classifier-free guidance (CFG); `Node` and `Edge` abbreviate `GraphNode` and `GraphEdge`.

```
image_gen = Sequential([
  Loop(section=Sequential([
        Parallel([ # 3-branch CFG, one branch per rank
          Node("LLM", [latents, t], [Edge("combine_cfg", "v_main")]),
          Node("LLM_cfg_text", [latents, t], [Edge("combine_cfg", "v_text")]),
          Node("LLM_cfg_img", [latents, t], [Edge("combine_cfg", "v_img")])]),
        Node("combine_cfg", [v_main, v_text, v_img, latents, t],
             [Edge("LLM", "latents"), Edge("LLM_cfg_text", "latents"),
              Edge("LLM_cfg_img", "latents")])]), # loop the latents back
      n_iters=49, outputs=[Edge("vae_decoder", "latents")]),
  Node("vae_decoder", [latents], [Edge(EMIT_TO_CLIENT, "image_output")])])
```

BAGEL's computation graph consists of seven nodes (`vit_encoder`, `vae_encoder`, `LLM`, `LLM_cfg_text`, `LLM_cfg_img`, `combine_cfg`, `vae_decoder`) and six Walks across those nodes. The `image_gen` Walk chains a 49-iteration `Loop` into a terminal `vae_decoder`; each iteration runs a `Parallel` region with the three CFG branches and then `combine_cfg`, which applies the CFG formula and an Euler step and loops the updated latents back to the branches. Without CFG the same Walk iterates a single `LLM` node.

3.2 What the Walk Graph Unlocks

The key contribution of the Walk Graph is that it decouples the composite model architecture from the system runtime. We enumerate the capabilities afforded by this design.

Modality-aware scheduling. The Walk abstraction lets the runtime scheduler execute requests in a model-agnostic manner. The scheduler tracks only each request's type and its current `GraphNode` and Walk (Figure 2), and selects the next Walk through the author's state machine. By construction it executes only the components each request needs.

Flexible parallelism. Authors directly capture parallelism within their model using the graph composition primitives, e.g., using `Parallel` for CFG in BAGEL (Section 3.1). Other examples include K-way model-predictive-control for world models [4] and multi-branch sampling for AR models [10]. The runtime is agnostic to the specific model architecture and supports parallelism uniformly. By contrast, vLLM-Omni couples modality-specific features such as CFG to the runtime through glue code that expands a request into multiple branches. Tensor parallelism and Ulysses sequence parallelism are declared per node, with sharded `Linear`, `MLP`, and `Attention` layers and a parallelism degree in the `yaml`, and the runtime handles synchronization, transport between nodes of differing TP or SP world sizes, and KV-cache transfer.

Flexible placement. The model deployer specifies a placement mapping from `GraphNode` to GPU rank(s), optionally per Walk, without modifying model or runtime code. This expresses data parallelism by naming multiple ranks, prefill-decode disaggregation [59] by naming different ranks for prefill and decode Walks, and independent scaling of components, e.g., encoders and decoders on many small GPUs and LLM backbones on few large ones. It also gives transparent resource sharing across requests of different types. For instance, BAGEL's `LLM` node appears in all Walks (Figure 2), and if it is placed on the same rank(s) in every Walk, the system multiplexes one physical replica across all of them.

Loop optimizations. `Loop` and `DynamicLoop` express AR decoding until EOS, fixed-step diffusion, and per-request rollout horizons as graph structure. This is what makes system optimizations portable across architectures. Stage-DAG abstractions in vLLM-Omni and SGLang-Omni cannot have cycles, so their loops stay internal to a stage. With loops in the graph, CUDA-graph capture and continuous batching [54] are agnostic to their presence, and loop iterations are scheduled like any other component. In BAGEL, for example, the scheduler can interleave flow steps and decoding steps on the same `LLM` node.

Flexible chunk policies. `StreamingGraphEdge(policy)` captures arbitrary producer-consumer patterns within a model. The three `ChunkPolicy` types of §3.1 cover every streaming connection in our evaluation. Qwen3-Omni uses `FixedChunkPolicy(1)` for Thinker→Talker and `LeftContextChunkPolicy` for Talker→Code2Wav. The runtime is agnostic to the policy used.

3.3 Runtime

The runtime executes requests for the expressed Walk Graph. An HTTP server accepts requests; a *Conductor* (one per server) maintains per-request Walk state and dispatches work to *Workers* via ZeroMQ [20]; Workers (single-process; one per GPU rank) execute the local subgraph (Figure 2), routing tensors directly to downstream workers. Cross-rank graph edges are inter-process tensor transfers, with streaming edges instantiating a per-request input buffer at the consumer. The data plane is pluggable and supports shared memory, as well as RDMA and TCP via Mooncake [37].

Engine and resources. Each `GraphNode` is executed by the `Engine`, which handles continuous batching, CUDA-graph capture and replay, and `torch.compile`. Persistent state (e.g., the paged KV cache, recurrent state, positions) is represented as a set of `Resources`, as is any computation that accesses such state or needs a *planning* stage to be CUDA-graph-compatible (e.g., attention, GDN, sampling). Each `Resource` type defines how its state is allocated (if it has any), what is planned before CUDA-graph replay, and how its state is updated at each step. Models declare which `Resources` each node uses, and the `Engine` manages their lifecycle.

Asynchronous execution. To overlap CPU scheduling with GPU execution, each worker schedules batch $N + 1$ while batch N is in flight. It traverses the Walk Graph assuming batch N produces its declared outputs to determine which nodes will be ready next, yielding to other ready nodes to avoid head-of-line blocking. Resource plans for batch $N + 1$ are built in a separate thread and CUDA stream and double-buffered. In decode loops, the next step reads its input tokens from a GPU-side table written by the sampler, so the worker routes the loop-back edge without per-request tensors. Termination checks are deferred until the next `DynamicLoop` iteration has launched, so each termination costs at most one wasted step. Speculation can be disabled per node if inadmissible.

4 Evaluation

We evaluate M* on BAGEL-7B [13], Qwen3-Omni-30B-A3B [51], Orpheus-3B [10], and V-JEPA 2 vitg-AC [4]. As baselines, we use vLLM-Omni for BAGEL and Qwen3-Omni, SGLang-Omni for Qwen3-Omni, VoxServe [25] for Orpheus, and Meta's native `vjepa2` implementation for V-JEPA 2. We also compare against Cornserve [12] in a scoped setting (Section 4.7). $\pi_{0.5}$ is unsupported by every baseline, so of the two robotics models we benchmark only V-JEPA 2-AC. All experiments run on a single $4\times$H100 node or a single $8\times$H200 node, and we report the GPUs used inline. Baselines are pinned to released versions, vLLM-Omni v0.22.0 and SGLang-Omni 0.1.0. All requests are seeded, every system serves the same inputs, BAGEL I2T fixes the output length per request, and audio outputs are the model's own. Section I lists every commit.

Metrics and protocol. For text outputs we report time-to-first-token (TTFT) and throughput. For image generation we report end-to-end (E2E) latency. For audio we report the per-request **real-time factor (RTF)**, the ratio of processing wall time to generated audio duration, where lower is better and <1 means streaming is feasible, along with time-to-first-audio (TTFA) and audio throughput in seconds of audio per wall-clock second. Closed-loop experiments (Sections 4.1 to 4.3) keep

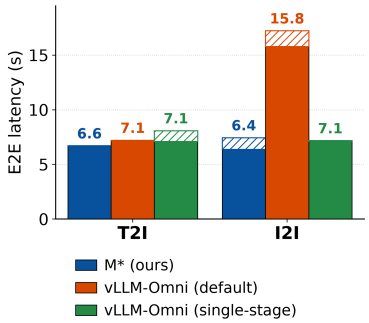

Figure 3: **BAGEL T2I/I2I E2E latency, B=1, CFG-parallel, H100.**

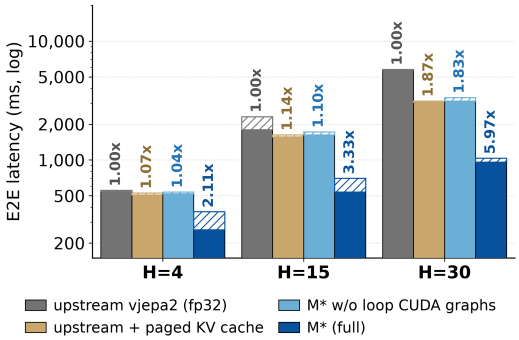

Figure 4: **V-JEPA 2 AC rollout, B=1, single H200** (p50, lower is better): upstream in its default fp32, upstream plus paged KV cache, M* without loop graphs, M* (bf16).

B requests in flight after 5 warm-up requests[1] and time at least $\max(5B, 12)$ requests for speech and text outputs, 10 requests for image outputs, and the 50 episodes of Section 4.3 for V-JEPA 2. BAGEL I2T and Orpheus cells are means of five runs, and the other closed-loop cells are single runs. Open-loop experiments (Section 4.5) use Poisson or gamma arrivals. Every near-capacity cell there is a median over n repetitions, with the spread reported where it changes the verdict, and no single-run tail number decides a claim in that study. Figures show p50 (solid) and p95 (hatched).

4.1 BAGEL: M* vs vLLM-Omni

We evaluate BAGEL on text-to-image (T2I), image editing (I2I), and image-to-text understanding (I2T), with inputs from VBench [24] and Food101 [9], on H100s. Images are 1024×1024 with a 50-step flow schedule. vLLM-Omni ships two deployments. The default one has a "Thinker" stage and a diffusion-transformer (DiT) stage that hold separate copies of the BAGEL weights, and the single-stage one collapses them into one engine. For generation both systems run CFG-parallel with one rank per CFG branch: three for M*'s `Parallel` region, and two on T2I and three on I2I for vLLM-Omni. For I2T we use one GPU and vLLM-Omni's default deployment with text-only requests, so that its DiT stage is skipped. Both systems ignore EOS and decode a seeded per-request token budget, so output lengths are identical.

Image generation (T2I, I2I). At B=1 (Figure 3), M* completes a T2I request in 6.62 s p50 and an I2I request in 6.37 s. vLLM-Omni's best deployment, single-stage with CFG parallelism, takes 7.10 s and 7.12 s. The gap is a modest $1.07\times$ and $1.12\times$. vLLM-Omni's default deployment, however, has $2.48\times$ our I2I latency, due to paying a per-request KV-cache transfer from the Thinker stage to the DiT stage. On T2I this costs little (7.14 s), but on I2I, where the cached context holds the input image, it costs $2.5\times$ (15.8 s; single-GPU results in Section E).

The "DiT" is the same Mixture-of-Transformers backbone running its generation experts, and what is cached is the request's text and ViT context KV written during prefill. In M*, the prefill walks write that KV into one paged pool and the CFG branches attend to it in place under branch labels. In vLLM-Omni, the Thinker and DiT are separate engines with separate KV managers, so the state must be moved per request, or the stages must be merged into one engine that loses per-stage batching and capture for I2T. The single-request advantage is small by design. What the shared pool buys is one deployment for every BAGEL workload, which we measure under mixed load in Section 4.5.

Image understanding (I2T). Against the corrected baseline (Figure 10 in Section E), M*'s text throughput is 15% higher at B=1, and the gap widens with concurrency to 46% at B=16 (615 vs. 420 tok/s). p50 E2E latency is 13% lower at B=1 and 34% lower at B=16 (3.84 vs. 5.82 s), and p50 TTFT is 0.19 vs. 0.30 s at B=1. The ViT encoder is a node in the walk graph, so its work interleaves with the LLM's, and the asynchronous scheduler (Section 3.3) hides the CPU work between encoding and image prefill, including attention-plan construction. Neither vLLM-Omni deployment serves all three workloads well. The default one pays the KV transfer on I2I, and the single-stage one runs I2T without continuous batching or token streaming. M* serves T2I, I2I and I2T from one configuration,

[1] V-JEPA 2 rollouts are short, so its runs use more warm-up requests. BAGEL I2T repetitions run back to back on one server, so only the first repetition is preceded by warm-up requests (two), and later repetitions start on an already-warm server.

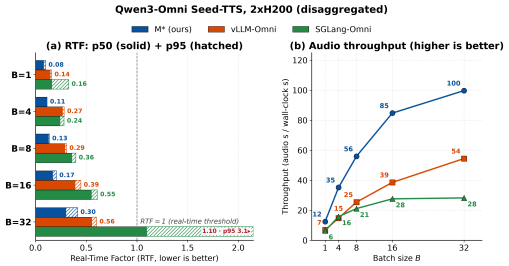

Figure 5: **Qwen3-Omni Seed-TTS, 2×H200.** (a) RTF (↓). (b) Audio throughput (↑).

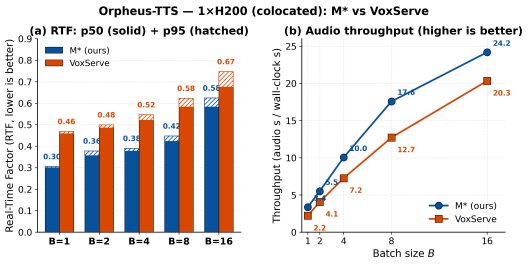

Figure 6: **Orpheus TTS, single H200.** (a) RTF (↓). (b) Audio throughput (↑).

and reaches PD disaggregation, encoder disaggregation, replication and tensor parallelism through configuration changes alone.

4.2 Qwen3-Omni and Orpheus: M* vs vLLM-Omni, SGLang-Omni, and VoxServe

We run Qwen3-Omni on the Seed-TTS [3] text-to-speech set on 2 H200s, with the Thinker on one rank and the Talker plus Code2Wav on the other (Figure 5). Both baselines use their shipped two-GPU Qwen3-Omni deployments. M* has the lowest RTF at every batch size. At B=16 it produces 84.7 seconds of audio per second, against 38.6 for vLLM-Omni and 27.6 for SGLang-Omni (2.2× and 3.1×), with p50 RTF 0.17 vs. 0.39 and 0.55. At B=32 the gap is 1.8× and 3.5×, and SGLang-Omni's p50 RTF passes 1.0. The closed-loop capacity at B=32 is 2.25 requests/s for M*, 1.07 for vLLM-Omni, and 0.62 for SGLang-Omni. M*'s first audio arrives before SGLang-Omni's at every batch size, at the median and at p99; vLLM-Omni's median is lower, but its p99 reaches 3.5 s at B=16, against 0.93 s for M*.

With the Thinker tensor-parallel over two ranks (3 GPUs), the ordering holds (Section E.2). M* also stays ahead at the November 2026 releases of both baselines, on text-to-speech and on a speech-to-speech workload, where SGLang-Omni has overtaken vLLM-Omni, and ahead of VoxServe v2 on Orpheus and Qwen3-TTS (Section E.3). The same runtime serves a plain text LLM, the one-node case, and on Qwen3.5-0.8B it leads vLLM and SGLang at every concurrency (Section E.5). Two mechanisms account for most of the gap, and Section 4.4 measures the first. CUDA-graph capture in M* is defined per submodule with custom inputs and outputs, so the entire Talker, including its multi-token-prediction loop, replays as one graph. Disabling that capture alone moves M*'s RTF at B=16 from 0.17 to 2.5. Second, placement is a configuration, so the Talker and Code2Wav share a worker and codec frames never cross a process boundary, while both baselines run every stage in its own process.

Orpheus-3B is a one-walk SpeechLM, served as an LLM node with a `StreamingGraphEdge` into a SNAC decoder node, and it gains from the runtime alone. On a single H200 at B=$\{1, 2, 4, 8, 16\}$ on Seed-TTS, M* delivers 14% lower p50 RTF than VoxServe [25], a speech-specialized serving system, at B=16 (25% to 35% at $B \leq 8$) and 19% to 55% higher throughput (Figure 6). The gains come from M*'s fused attention projections, its CUDA-graph-compatible sampler, and the speculative FlashInfer planning of Section 3.3.

4.3 V-JEPA 2: Rollout for Robotic Planning

V-JEPA 2 supports action-conditioned (AC) rollout: each step predicts the next latent frame from an action and the previous states. We compare against Meta's `vjepa2` implementation on one H200 at B=1 and horizons H=$\{4, 15, 30\}$, on the first 50 DROID episodes [26]. The upstream loop re-prefills the growing sequence at every step. M* declares the rollout as a `DynamicLoop` over the predictor node, so each step attends to paged KV state and the loop body replays as a CUDA graph. To separate the two effects, we also measure the upstream loop with a paged KV cache added (its outputs match upstream) and M* with loop CUDA graphs disabled (Figure 4). At H=30 the KV cache alone gives 1.87×. M* without graphs matches that corrected baseline at 1.83×. Graph replay of the loop body takes M* to 5.97× (960 ms vs. 5.73 s p50). The upstream runs in its default fp32 and M* in bf16, but precision is not the source of the gain: the upstream's KV-cache control is slower in bf16 (1.64× at H=30) and M* in fp32 (bf16 attention kernel) reaches 6.4×. At H=4 and H=15 the speedups are 2.11× and 3.33×, and graph replay is bit-exact against eager execution on all 150 rollouts.

Table 1: Mechanism ablation, one M* toggle at a time (BAGEL I2T 1×H100; Qwen3-Omni TTS 2×H200; V-JEPA 2 1×H200).

Family	Mechanism turned off	Metric	Effect
BAGEL I2T, B=16	loop CUDA-graph capture	throughput 615 → 312 tok/s	1.97×
BAGEL I2T, B=16	continuous batching across iterations	throughput 615 → 136 tok/s	4.52×
BAGEL I2T, B=16	asynchronous pipeline	throughput 615 → 523 tok/s	1.18×
BAGEL I2T, B=1	loop CUDA-graph capture	E2E p50 1.14 → 4.12 s	3.6×
Qwen3-Omni TTS, B=16	Talker-loop CUDA graph	RTF p50 0.167 → 2.52	15×
Qwen3-Omni TTS, B=16	asynchronous pipeline	RTF p50 0.167 → 0.260	1.56×
Qwen3-Omni TTS, B=16	placement: all nodes on one rank instead of two (config only)	RTF p50 0.167 → 0.326	1.95×
V-JEPA 2, H=30	loop CUDA graphs (KV cache kept)	speedup 5.97× → 1.83×	3.26×

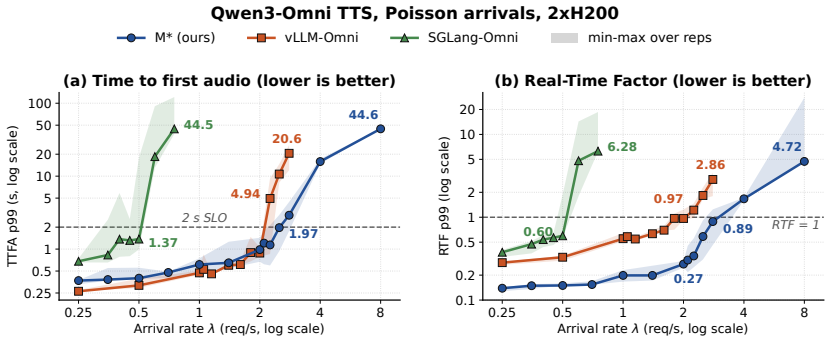

Figure 7: **Qwen3-Omni TTS, Poisson arrivals, 2×H200.** Median over 2 to 11 repetitions per rate of (a) TTFA p99 and (b) RTF p99; shading is the min–max range. Table 7 lists the rates around each knee.

4.4 Where the Gains Come From

Table 1 turns off one M* mechanism at a time, with the same M* everywhere, on the three loop families above. Loop-body CUDA-graph capture is the largest lever on Qwen3-Omni and V-JEPA 2 (15× and 3.26×) and worth 1.97× to 3.6× on BAGEL I2T, where continuous batching across iterations is the largest (4.52×). The asynchronous pipeline is worth 1.18× to 1.56×. The placement row shows what moving the whole model onto one rank costs (1.95×), a configuration change in either direction.

4.5 Online Serving

Open-loop arrivals. We drive Qwen3-Omni TTS on 2 GPUs with Poisson arrivals at increasing rate λ, 200 requests per run, under a first-audio SLO of TTFA p99 $\leq$ 2 s. We report medians over n repetitions (Figure 7), with $n \geq 6$ at each system's capacity knee (Table 7 in Section E.9), because single runs do not reproduce near capacity: TTFA p99 spreads reach 3 to 18× at the same operating point. The sustainable rate λ^* under the SLO is 2.25 req/s for M*, 2.0 for vLLM-Omni, and 0.5 for SGLang-Omni, whose tail collapses at λ=0.6. Beyond the knee M* degrades gracefully, and peak SLO-goodput, completed requests per second with RTF $<$ 1, is 2.13 to 2.16 vs. 1.38 for vLLM-Omni (1.5 to 1.6×). M* keeps $\geq$ 99.75% real-time attainment through λ=2.8, while vLLM-Omni is at 71% by λ=2.25, and at a matched λ=2.0 the RTF p99 is 0.27 vs. 0.97. At low rates vLLM-Omni's p50 first-audio latency is better than ours (0.20 vs. 0.29 s at λ=0.25). A replication on a second cluster (4×H100) moves both knees up, to λ^* of 2.8 vs. 2.5 vs. 0.5, and reproduces the relationships, with goodput 1.50× and RTF p99 at λ=2.0 of 0.20 vs. 0.82 (bursty arrivals are in Section E.9).

Mixed workloads on one deployment. We issue a seeded Poisson mix of t2i, i2t and i2i requests (1:2:1, identical sequences for every system and repetition) and score each request against a per-type SLO. Images must complete within 30 s, and i2t must reach its first token within 2 s and complete within 15 s. Attainment is the share of issued requests that complete and meet their SLO, with failures counted as misses, and goodput is SLO-meeting completions per second. All numbers are medians of three repetitions (Figure 8; Table 8 in Section E.9). On 2 H100s, M*'s replicated deployment, two data-parallel copies and a configuration change, reaches 52% and 46% attainment at λ=0.18 and 0.26 req/s. vLLM-Omni's best two-GPU deployment, its Thinker/DiT split, reaches 41% and 32%. On 3 GPUs with CFG parallelism, M* holds 89% vs. 36% and 65% vs. 16%. vLLM-Omni's CFG-parallel deployment cannot serve t2i in a mixed stream at default semantics, because its branch count is fixed

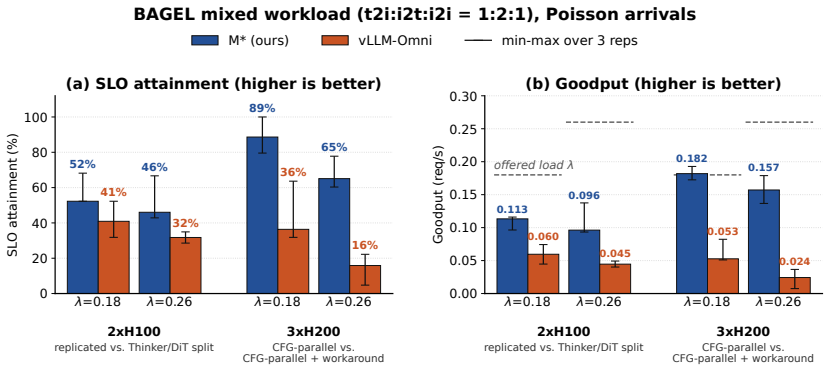

Figure 8: **BAGEL mixed online workload (t2i:i2t:i2i = 1:2:1).** Median over 3 repetitions of (a) SLO attainment and (b) goodput; whiskers are the min–max range. Labels under each group name the M* vs. vLLM-Omni deployments; the repetition protocol and full results are in Section E.9.

at deployment time, so these cells use its documented workaround (`cfg_img_scale`=2.0 on t2i), which changes the generation semantics. Below saturation vLLM-Omni's split holds text tails better than a single-GPU M* deployment, so the win is the mix.

4.6 Overheads, Transport, Multi-Host Deployment, and Placement

The built-in per-(node, walk) profiler puts the cost of fine-grained graph execution at 1 to 3 ms of scheduling and pre- and post-processing per node execution, and at 1.6 ms per step for BAGEL's cross-branch CFG combine, about 1.5% of a 100 ms step. The asynchronous pipeline overlaps most of it (Table 1, Section E.11). Forcing every cross-worker edge onto Mooncake TCP instead of shared memory keeps Qwen3-Omni's median RTF below 0.4 through B=32, costs Orpheus 5 to 24% in RTF at B=16 by placement, and costs a BAGEL prefill/decode split 5.5% (Section E.12). We also run the same deployments as two process trees joined through the TCP control plane and the Mooncake TCP tensor plane, the code path of a two-host deployment. Outputs stay byte-identical in 15 of 16 pinned-seed runs, with one near-tie token flip, a BAGEL prefill/decode split pays 1.2 to 3.0% in E2E latency, and an Orpheus streaming split pays 16 to 17% in RTF, matching the forced-TCP single-tree numbers (Section E.13). A physically multi-node deployment and the RDMA transport remain untested (Section H). Placement is a configuration edit, and under long-audio mixed load with 240 s inputs the colocated Qwen3-Omni placement's first-audio latency grows without bound within the measurement window (p50 515 $\rightarrow$ 975 ms), while the prefill/decode-disaggregated placement stays bounded. Moving the vocoder next to the Talker, one line, cuts the worst decode stall by 37%, and vLLM-Omni's stage schema cannot express this disaggregation (Section E.14).

4.7 Cornserve and Developer Effort

Cornserve [12], the nearest prior system, composes per-component executors through a task graph recorded per request, requires statically determined control flow, pins its Qwen3-Omni Talker to eager execution, and captures no CUDA graphs in its encoder and generator executors. We measured it in its own benchmark mode on one H200, with their fork and flags and no Thinker compute, both in its favor, against M* serving the full pipeline on 2 GPUs of the same node. Its talker path leaves real time between c=1 and c=4 (RTF 0.63 and 1.18 vs. 0.080 and 0.113, medians of three), and first audio equals completion (2.8 and 4.7 s vs. 0.24 and 0.33 s). Un-pinning eager execution still leaves it near RTF 1.0 at c=4, since engine-level graphs alone do not batch the loop (Section E.1). Per-model integration cost in M* is in Section E.16.

5 Conclusion

M* is a universal serving runtime built on the observation that composite multimodal models are walks over a dataflow graph. Section G places it among token-centric and multimodal serving systems and the dataflow systems whose lineage the Walk Graph inherits. The Walk Graph decouples the model architecture from placement and execution, and two runtime mechanisms, loop-body capture and continuous batching across loop iterations, account for most of the measured gains.

References

[1] Niket Agarwal, Arslan Ali, Maciej Bala, Yogesh Balaji, Erik Barker, Tiffany Cai, Prithvijit Chattopadhyay, Yongxin Chen, Yin Cui, Yifan Ding, Daniel Dworakowski, Jiaojiao Fan, Michele Fenzi, Francesco Ferroni, Sanja Fidler, Dieter Fox, Songwei Ge, Yunhao Ge, Jinwei Gu, Siddharth Gururani, Ethan He, Jiahui Huang, Jacob Huffman, Pooya Jannaty, Jingyi Jin, Seung Wook Kim, Gergely Klár, Grace Lam, Shiyi Lan, Laura Leal-Taixe, Anqi Li, Zhaoshuo Li, Chen-Hsuan Lin, Tsung-Yi Lin, Huan Ling, Ming-Yu Liu, Xian Liu, Alice Luo, Qianli Ma, Hanzi Mao, Kaichun Mo, Arsalan Mousavian, Seungjun Nah, Sriharsha Niverty, David Page, Despoina Paschalidou, Zeeshan Patel, Lindsey Pavao, Morteza Ramezanali, Fitsum Reda, Xiaowei Ren, Vasanth Rao Naik Sabavat, Ed Schmerling, Stella Shi, Bartosz Stefaniak, Shitao Tang, Lyne Tchapmi, Przemek Tredak, Wei-Cheng Tseng, Jibin Varghese, Hao Wang, Haoxiang Wang, Heng Wang, Ting-Chun Wang, Fangyin Wei, Xinyue Wei, Jay Zhangjie Wu, Jiashu Xu, Wei Yang, Lin Yen-Chen, Xiaohui Zeng, Yu Zeng, Jing Zhang, Qinsheng Zhang, Yuxuan Zhang, Qingqing Zhao, and Artur Zolkowski. Cosmos world foundation model platform for physical AI. arXiv preprint arXiv:2501.03575, 2025.

[2] Eloi Alonso, Adam Jelley, Vincent Micheli, Anssi Kanervisto, Amos Storkey, Tim Pearce, and François Fleuret. Diffusion for world modeling: Visual details matter in Atari. *Advances in Neural Information Processing Systems*, 37:58757–58791, 2024.

[3] Philip Anastassiou, Jiawei Chen, Jitong Chen, Yuanzhe Chen, Zhuo Chen, Ziyi Chen, Jian Cong, Lelai Deng, Chuang Ding, Lu Gao, Mingqing Gong, Peisong Huang, Qingqing Huang, Zhiying Huang, Yuanyuan Huo, Dongya Jia, Chumin Li, Feiya Li, Hui Li, Jiaxin Li, Xiaoyang Li, Xingxing Li, Lin Liu, Shouda Liu, Sichao Liu, Xudong Liu, Yuchen Liu, Zhengxi Liu, Lu Lu, Junjie Pan, Xin Wang, Yuping Wang, Yuxuan Wang, Zhen Wei, Jian Wu, Chao Yao, Yifeng Yang, Yuanhao Yi, Junteng Zhang, Qidi Zhang, Shuo Zhang, Wenjie Zhang, Yang Zhang, Zilin Zhao, Dejian Zhong, and Xiaobin Zhuang. Seed-TTS: A family of high-quality versatile speech generation models, 2024. URL `https://arxiv.org/abs/2406.02430`.

[4] Mido Assran, Adrien Bardes, David Fan, Quentin Garrido, Russell Howes, Mojtaba Komeili, Matthew Muckley, Ammar Rizvi, Claire Roberts, Koustuv Sinha, Artem Zholus, Sergio Arnaud, Abha Gejji, Ada Martin, Francois Robert Hogan, Daniel Dugas, Piotr Bojanowski, Vasil Khalidov, Patrick Labatut, Francisco Massa, Marc Szafraniec, Kapil Krishnakumar, Yong Li, Xiaodong Ma, Sarath Chandar, Franziska Meier, Yann LeCun, Michael Rabbat, and Nicolas Ballas. V-JEPA 2: Self-supervised video models enable understanding, prediction and planning. *arXiv preprint arXiv:2506.09985*, 2025.

[5] Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025.

[6] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ML. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 4, 2022.

[7] Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: A vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.

[8] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision (ECCV)*, 2014.

[9] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision*, 2014.

[10] Canopy Labs. Orpheus-TTS: Towards human-sounding speech. GitHub repository, 2025. URL `https://github.com/canopyai/Orpheus-TTS`.

[11] Xiaokang Chen, Zhiyu Wu, Xingchao Liu, Zizheng Pan, Wen Liu, Zhenda Xie, Xingkai Yu, and Chong Ruan. Janus-Pro: Unified multimodal understanding and generation with data and model scaling. *arXiv preprint arXiv:2501.17811*, 2025.

[12] Jae-Won Chung, Jeff J. Ma, Jisang Ahn, Yizhuo Liang, Akshay Jajoo, Myungjin Lee, and Mosharaf Chowdhury. Cornserve: A distributed serving system for any-to-any multimodal models. *arXiv preprint arXiv:2603.12118*, 2026.

[13] Chaorui Deng, Deyao Zhu, Kunchang Li, Chenhui Gou, Feng Li, Zeyu Wang, Shu Zhong, Weihao Yu, Xiaonan Nie, Ziang Song, Guang Shi, and Haoqi Fan. Emerging properties in unified multimodal pretraining. arXiv preprint arXiv:2505.14683, 2025.

[14] Zhihao Du, Yuxuan Wang, Qian Chen, Xian Shi, Xiang Lv, Tianyu Zhao, Zhifu Gao, Yexin Yang, Changfeng Gao, Hui Wang, Fan Yu, Huadai Liu, Zhengyan Sheng, Yue Gu, Chong Deng, Wen Wang, Shiliang Zhang, Zhijie Yan, and Jingren Zhou. CosyVoice 2: Scalable streaming speech synthesis with large language models. *arXiv preprint arXiv:2412.10117*, 2024.

[15] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, Kyle Lacey, Alex Goodwin, Yannik Marek, and Robin Rombach. Scaling rectified flow transformers for high-resolution image synthesis. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[16] Jiarui Fang and Shangchun Zhao. USP: A unified sequence parallelism approach for long context generative AI. *arXiv preprint arXiv:2405.07719*, 2024.

[17] Jiarui Fang, Jinzhe Pan, Xibo Sun, Aoyu Li, and Jiannan Wang. xDiT: an inference engine for diffusion transformers (DiTs) with massive parallelism. *arXiv preprint arXiv:2411.01738*, 2024.

[18] Jiarui Fang, Jinzhe Pan, Aoyu Li, Xibo Sun, and Jiannan Wang. PipeFusion: Patch-level pipeline parallelism for diffusion transformers inference. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL `https://openreview.net/forum?id=5xwyxupsLL`.

[19] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. arXiv preprint arXiv:2301.04104, 2023.

[20] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[21] Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance. *arXiv preprint arXiv:2207.12598*, 2022.

[22] Zicong Hong, Yuyan Chen, Haoyue Zhang, Peng Li, Wuhui Chen, Song Guo, and Xiaowei Shen. Efficient multimodal serving via module multiplexing. In *Proceedings of the Twenty-First European Conference on Computer Systems (EuroSys)*, 2026. doi: 10.1145/3767295.3769389.

[23] Ailin Huang, Boyong Wu, Bruce Wang, Chao Yan, Chen Hu, et al. Step-Audio: Unified understanding and generation in intelligent speech interaction. *arXiv preprint arXiv:2502.11946*, 2025.

[24] Ziqi Huang, Yinan He, Jiashuo Yu, Fan Zhang, Chenyang Si, Yuming Jiang, Yuanhan Zhang, Tianxing Wu, Qingyang Jin, Nattapol Chanpaisit, et al. VBench: Comprehensive benchmark suite for video generative models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21807–21818, 2024.

[25] Keisuke Kamahori, Wei-Tzu Lee, Atindra Jha, Rohan Kadekodi, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. VoxServe: Streaming-centric serving system for speech language models. *arXiv preprint arXiv:2602.00269*, 2026.

[26] Alexander Khazatsky, Karl Pertsch, Suraj Nair, Ashwin Balakrishna, Sudeep Dasari, Siddharth Karamcheti, Soroush Nasiriany, Mohan Kumar Srirama, Lawrence Yunliang Chen, Kirsty Ellis, Peter David Fagan, Joey Hejna, Masha Itkina, Marion Lepert, Yecheng Jason Ma, Patrick Tree Miller, Jimmy Wu, Suneel Belkhale, Shivin Dass, Huy Ha, Arhan Jain, Abraham Lee, Youngwoon Lee, Marius Memmel, Sungjae Park, Ilija Radosavovic, Kaiyuan Wang, Albert Zhan, Kevin Black, Cheng Chi, Kyle Beltran Hatch, Shan Lin, Jingpei Lu, Jean Mercat, Abdul Rehman, Pannag R Sanketi, Archit Sharma, Cody Simpson, Quan Vuong, Homer Rich Walke, Blake Wulfe, Ted Xiao, Jonathan Heewon Yang, Arefeh Yavary, Tony Z. Zhao, Christopher Agia, Rohan Baijal, Mateo Guaman Castro, Daphne Chen, Qiuyu Chen, Trinity Chung, Jaimyn Drake, Ethan Paul Foster, Jensen Gao, David Antonio Herrera, Minho Heo, Kyle Hsu, Jiaheng Hu, Donovon Jackson, Charlotte Le, Yunshuang Li, Kevin Lin, Roy Lin, Zehan Ma, Abhiram Maddukuri, Suvir Mirchandani, Daniel Morton, Tony Nguyen, Abigail O'Neill, Rosario Scalise, Derick Seale, Victor Son, Stephen Tian, Emi Tran, Andrew E. Wang, Yilin Wu, Annie Xie, Jingyun Yang, Patrick Yin, Yunchu Zhang, Osbert Bastani, Glen Berseth, Jeannette Bohg, Ken Goldberg, Abhinav Gupta, Abhishek Gupta, Dinesh Jayaraman, Joseph J Lim, Jitendra Malik, Roberto Martín-Martín, Subramanian Ramamoorthy, Dorsa Sadigh, Shuran Song, Jiajun Wu, Michael C. Yip, Yuke Zhu, Thomas Kollar, Sergey Levine, and Chelsea Finn. DROID: A large-scale in-the-wild robot manipulation dataset. *arXiv preprint arXiv:2403.12945*, 2024.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pages 611–626, 2023.

[28] Zekai Li, Yihao Liang, Hongfei Zhang, Jian Chen, and Zhijian Liu. FlashDrive: Flash Vision-Language-Action Inference For Autonomous Driving. *ES-Reasoning Workshop at ICLR*, 2026.

[29] Weixin Liang, Lili Yu, Liang Luo, Srinivasan Iyer, Ning Dong, Chunting Zhou, Gargi Ghosh, Mike Lewis, Wen tau Yih, Luke Zettlemoyer, and Xi Victoria Lin. Mixture-of-transformers: A sparse and scalable architecture for multi-modal foundation models. *arXiv preprint arXiv:2411.04996*, 2024.

[30] Xingchao Liu, Chengyue Gong, and Qiang Liu. Flow straight and fast: Learning to generate and transfer data with rectified flow. In *International Conference on Learning Representations (ICLR)*, 2023.

[31] Jeff J. Ma, Jae-Won Chung, Jisang Ahn, Yizhuo Liang, Akshay Jajoo, Myungjin Lee, and Mosharaf Chowdhury. Cornfigurator: Automated planning for any-to-any multimodal model serving. *arXiv preprint arXiv:2512.14098*, 2025.

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 561–577, 2018.

[33] NVIDIA. Triton inference server: Ensemble models and business logic scripting. `https://github.com/triton-inference-server/server`, 2026. Accessed 2026-11-06.

[34] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015.

[35] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[36] William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4195–4205, 2023.

[37] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A KVCache-centric disaggregated architecture for LLM serving. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST)*, 2025.

[38] Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ramachandran Ramjee, and Rodrigo Fonseca. ModServe: Modality- and stage-aware resource disaggregation for scalable multimodal model serving. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2025.

[39] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International conference on machine learning*, pages 28492–28518. PMLR, 2023.

[40] sgl-project. SGLang-Omni: High-performance multi-stage pipeline framework for omni models. GitHub repository, 2026. URL `https://github.com/sgl-project/sglang-omni`. Accessed 2026-03-20.

[41] Gursimran Singh, Xinglu Wang, Yifan Hu, Timothy Yu, Linzi Xing, Wei Jiang, Zhefeng Wang, Xiaolong Bai, Yi Li, Ying Xiong, et al. Efficiently serving large multimodal models using EPD disaggregation. *arXiv preprint arXiv:2501.05460*, 2025.

[42] Hubert Siuzdak, Florian Grötschla, and Luca A Lanzendörfer. SNAC: Multi-scale neural audio codec. *arXiv preprint arXiv:2410.14411*, 2024.

[43] Xibo Sun, Jiarui Fang, Aoyu Li, and Jinzhe Pan. Unveiling redundancy in diffusion transformers (DiTs): A systematic study. *arXiv preprint arXiv:2411.13588*, 2024.

[44] Inferix Team, Tianyu Feng, Yizeng Han, Jiahao He, Yuanyu He, Xi Lin, Teng Liu, Hanfeng Lu, Jiasheng Tang, Wei Wang, et al. Inferix: A block-diffusion based next-generation inference engine for world simulation. *arXiv preprint arXiv:2511.20714*, 2025.

[45] The Ray Project. Ray serve: Scalable and programmable serving. `https://docs.ray.io/en/latest/serve/`, 2026. Accessed 2026-11-06.

[46] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, Dhruv Nair, Sayak Paul, William Berman, Yiyi Xu, Steven Liu, and Thomas Wolf. Diffusers: State-of-the-art diffusion models. `https://github.com/huggingface/diffusers`, 2022.

[47] Yan Wang, Wenjie Luo, Junjie Bai, Yulong Cao, Tong Che, Ke Chen, Yuxiao Chen, Jenna Diamond, Yifan Ding, Wenhao Ding, et al. Alpamayo-r1: Bridging reasoning and action prediction for generalizable autonomous driving in the long tail. *arXiv preprint arXiv:2511.00088*, 2025.

[48] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, 2020.

[49] Jinheng Xie, Zhenheng Yang, and Mike Zheng Shou. Show-o2: Improved native unified multimodal models. *arXiv preprint arXiv:2506.15564*, 2025.

[50] 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*, 2025.

[51] 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. Qwen3-Omni technical report. *arXiv preprint arXiv:2509.17765*, 2025.

[52] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems (MLSys)*, 2025.

[53] Peiqi Yin, Jiangyun Zhu, Han Gao, Chenguang Zheng, Yongxiang Huang, Taichang Zhou, Ruirui Yang, Weizhi Liu, Weiqing Chen, Canlin Guo, et al. vLLM-Omni: Fully disaggregated serving for any-to-any multimodal models. *arXiv preprint arXiv:2602.02204*, 2026.

[54] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 521–538, 2022.

[55] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[56] Peiyuan Zhang, Yongqi Chen, Haofeng Huang, Will Lin, Zhengzhong Liu, Ion Stoica, Eric Xing, and Hao Zhang. VSA: Faster video diffusion with trainable sparse attention. *arXiv preprint arXiv:2505.13389*, 2025.

[57] Peiyuan Zhang, Yongqi Chen, Runlong Su, Hangliang Ding, Ion Stoica, Zhengzhong Liu, and Hao Zhang. Fast video generation with sliding tile attention. *arXiv preprint arXiv:2502.04507*, 2025.

[58] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

[59] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

Appendix

A Two More Walk Graphs at a Glance

The same four primitives express qualitatively different model families. Two examples make the range concrete.

Qwen3-Omni: three partitions, two streaming edges. Qwen3-Omni [51] is an omni-modality LLM in a three-partition topology: a Thinker (text-out AR LLM), a Talker (codec-token AR LLM), and Code2Wav (audio-codec vocoder), each on its own rank. The Thinker streams hidden states to the Talker via a `FixedChunkPolicy(`K`=1)` `StreamingGraphEdge`, and the Talker streams codec frames to Code2Wav via a `LeftContextChunkPolicy`. Eight Walks span the three partitions, including separate prefill Walks per input modality and separate prefill / last-prefill / decode Walks for the Talker.

V-JEPA 2: five Walks selected per request. V-JEPA 2 [4] is a video world model whose predictor is reused across distinct tasks. Its action-conditioned variant declares five Walks: a single-shot `prefill_video`; an `encoder_only` Walk for cross-Walk pre-encoding; a batched `rollout` Walk built around a `DynamicLoop` with per-request horizon H; a `streaming_rollout` variant that emits each iter's prediction immediately; and an MPC Walk that runs the predictor with K candidate action sequences in one batched forward, scored by an `mpc_scorer` node — K-way model-predictive control as a 3-node `Sequential`. The same predictor weights serve all five Walks. The masked-predictor variant declares the same set minus the MPC Walk.

B Walk Graph Primitives: Full Signatures and Semantics

Table 2 gives the full signatures and semantics of the four composable primitives summarized in §3.1, plus the streaming edge variant `StreamingGraphEdge`.

Table 2: The four composable primitives that generate a Walk Graph, plus the streaming variant of `GraphEdge`. `GraphNode` and `GraphEdge` are the atomic types; the composite primitives below close under nesting.

Primitive	Signature	Semantics
`Sequential`	`list[Section]` $\rightarrow$ `Section`	Chain — outputs of one section feed the next.
`Parallel`	`list[Section]` $\rightarrow$ `Section`	Fan-out — children execute concurrently on (possibly distinct) ranks.
`Loop`	`Section`$\times$`int` $\rightarrow$ `Section`	Bounded iteration. Two output channels: `outputs` (cache wiped per iter — only the last iter's tensor flows downstream) and `accumulated_outputs` (cache persists across iters — every iter's contribution is concatenated and emitted en bloc). Disjoint by name.
`DynamicLoop`	`Section`$\times$`int`$_{\max}\times$`str` $\rightarrow$ `Section`	`Loop` with per-request early-exit. A submodule signals stop via `request_info.register_loop_stop(name)`; on the next iter boundary the runtime advances to terminal outputs. Used for EOS and rollout horizon.
`StreamingGraphEdge`	`str` $\times$ `ChunkPolicy` $\rightarrow$ `Edge`	Streaming variant of `GraphEdge`. Producer emits one tensor at a time; the consumer's `StreamBuffer` accumulates and gates dispatch via the policy.

C YAML Placement Listings

The two listings below ground the disaggregation patterns described in §3.2 (the Flexible parallelism and Flexible placement capabilities).

Listing 2: Per-Walk placement: Qwen3-Omni Thinker prefill/decode disaggregation.

```
node_groups:
  - node_names: [Thinker]
    graph_walks: [prefill_text, prefill_audio, prefill_vision]
    ranks: [0]
  - node_names: [Thinker]
    graph_walks: [thinker_decode]
    ranks: [1]
```

Listing 3: Declarative parallelism: BAGEL CFG-parallel placement (one rank per branch).

```
node_groups:
  - node_names: [LLM]
    ranks: [0]
  - node_names: [LLM_cfg_text]
    ranks: [1]
  - node_names: [LLM_cfg_img]
    ranks: [2]
  - node_names: [combine_cfg, vae_decoder]
    ranks: [0]
```

D Detailed Subsumption Table

Table 3 compares the Walk Graph with existing abstractions (§3) along nine axes, each tracing back to a primitive or capability introduced in §3.

Table 3: The Walk Graph compared with existing multimodal serving abstractions. vLLM-Omni and SGLang-Omni implement stage graphs at *engine-instance* granularity; Cornserve composes per-component executors through a task graph recorded per request; VoxServe specializes a single `Model` interface for SpeechLMs. Each prior system corresponds to a restriction of the Walk Graph (see §3). Baseline facts are as of the pinned versions in Section I (Cornserve: commit `6d04df90`, July 2026).

	vLLM-Omni	**SGLang-Omni**	**Cornserve**	**VoxServe**	**M* (ours)**
Graph granularity	Stage = engine instance	Stage = worker pool	Executor per component type (encoder, LLM, generator)	Single `Model` class	**Node = one forward pass**
Walks per model	One pipeline (frozen) per `model_type`	One `PipelineConfig` per variant	Task graph recorded per request (record / replay)	Single forward path	**First-class**; e.g. BAGEL: 6 walks share one node set
Cross-walk node sharing	—	—	—	N/A	Yes; `LLM` node is in every BAGEL walk
Typed primitives	Flat DAG + binary `async_chunk` flag	Flat DAG + sources / aggregators	Python tasks; control flow must be statically determined	None	`Sequential` / `Parallel` / `Loop` / `DynamicLoop` / `StreamingGraphEdge`
Iteration as graph structure	Hidden in engine / model	Hidden in executor (e.g. `chunked_decode` `while`-loop)	Hidden in executors	Hidden in scheduler's outer loop	Graph-level `Loop` / `DynamicLoop`
Iter loops ⇒ CUDA graphs	No CUDA-graph path for diffusion stages (compile is the only lever); Code2Wav graphs since v0.21	Per-stage; codec executors are eager Python loops	No: encoder and generator executors eager; Qwen-Omni Talker pinned eager	Audio-only	Yes — per-iter `forward` has fixed tensor sizes
Streaming policies	Single `async_chunk` toggle	Generic `StreamQueue` transport; chunking is per-model	Per-executor streaming	Per-model `detokenize_interval` (audio only)	Three reusable `ChunkPolicy` types cover taxonomy
Placement granularity	Per-stage `devices: str`	Per-stage `gpu_placement: dict`	Per executor (Kubernetes pod; GPUs exclusively owned)	AR / detokenizer split (fixed cuda:0 / cuda:1)	Per-(node, walk)
Multimodal models supported	Qwen2.5/3-Omni, BAGEL, MiMo-Audio, + DiTs via Diffusers	Qwen3-Omni, Ming-Omni, FishAudio S2-Pro, Voxtral-TTS	Qwen-Omni family, VLMs, diffusion generators	8 SpeechLM families (TTS + STS only)	BAGEL, Qwen3-Omni, V-JEPA 2 (masked + AC), $\pi_{0.5}$, Orpheus, Cosmos3, Whisper, Higgs-Audio, Qwen3-TTS, Wan2.2

E Additional Experimental Results

Figure 9 compares vLLM-Omni with M* for T2I and I2I job completion time, on a single H100 (no CFG-parallelism).

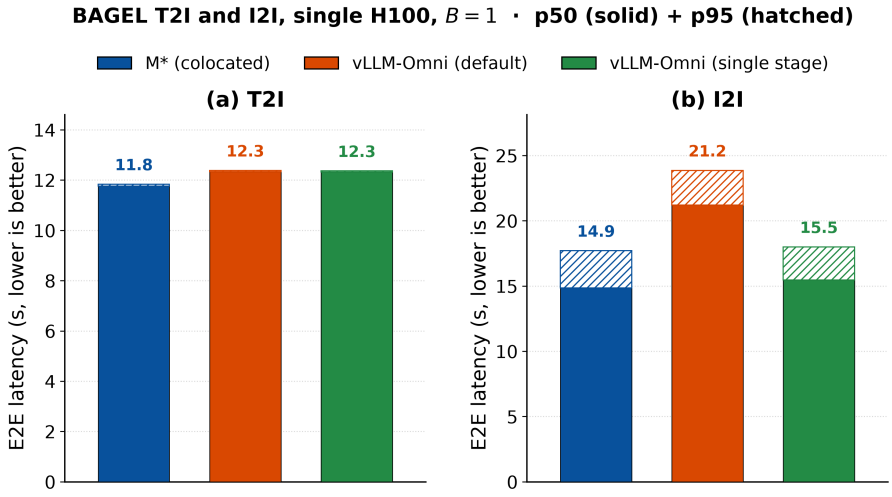

Figure 9: **BAGEL T2I and I2I, single H100,** B=1**.** M* vs. vLLM-Omni's default and single-stage deployments.

On one GPU M*'s T2I p50 latency is 1.05× faster than either vLLM-Omni deployment (11.8 vs. 12.3 s). For I2I, M* is 1.43× faster than the default deployment, which pays the Thinker-to-DiT KV transfer, and 1.04× faster than the single-stage one (14.9 vs. 21.2 and 15.5 s; Figure 9).

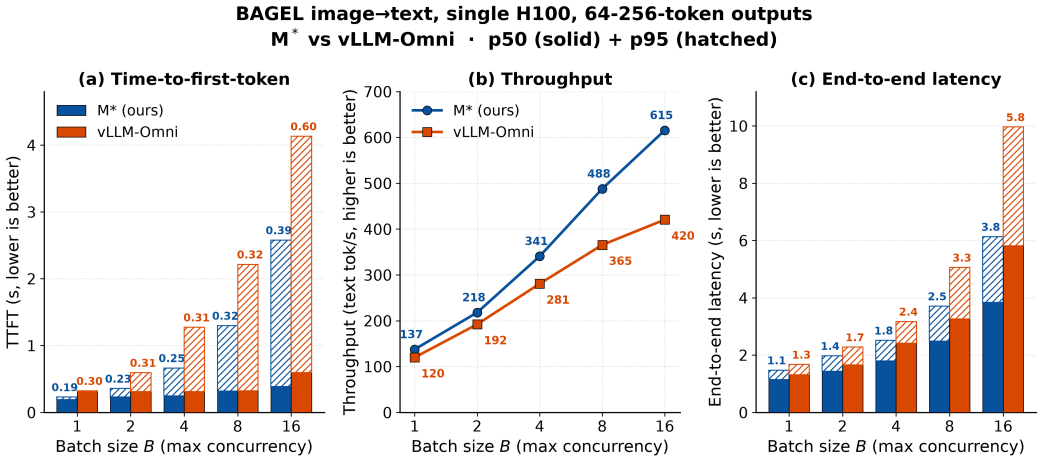

Figure 10: **BAGEL I2T, single H100,** $B \in \{1, 2, 4, \ldots, 16\}$**,** `ignore_eos` with output lengths between 64 and 256 (the setting of Section 4.1). (a) TTFT. (b) Throughput (text tok/s). (c) End-to-end request latency. Means of five runs.

Figure 10 is the I2T sweep discussed in Section 4.1; Figures 11 and 12 repeat it for output lengths sampled between 16 and 128 tokens and between 128 and 512 tokens. Overall, we have the largest advantage on shorter-decode and higher-concurrency workloads (61% higher throughput at B=16 for 16 to 128 tokens), while the advantage narrows to 33% in the longer-sequence setting.

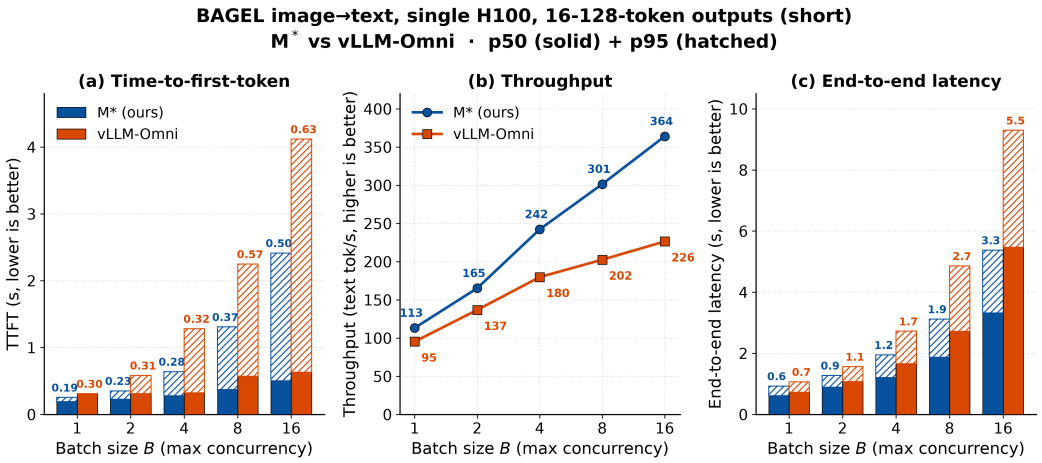

Figure 11: **BAGEL I2T, single H100,** $B \in \{1, 2, 4, \ldots, 16\}$**,** `ignore_eos` with output lengths between 16 and 128. (a) TTFT. (b) Throughput (text tok/s). (c) End-to-end request latency. Means of five runs.

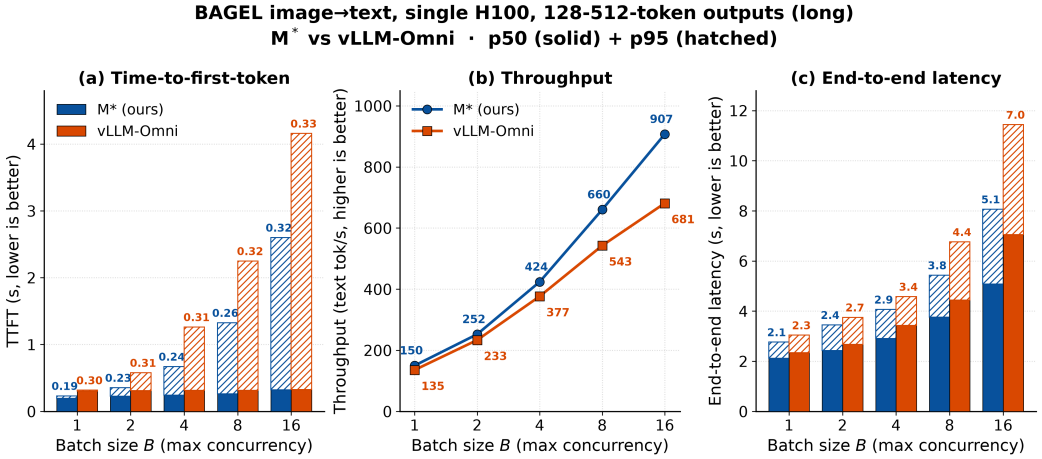

Figure 12: **BAGEL I2T, single H100,** $B \in \{1, 2, 4, \ldots, 16\}$**,** `ignore_eos` with output lengths between 128 and 512. (a) TTFT. (b) Throughput (text tok/s). (c) End-to-end request latency. Means of five runs.

E.1 Cornserve: scoped measurement

Table 4 lists the Cornserve measurement of Section 4.7. We ran their talker-plus-vocoder executor from their fork at commit `6d04df90`, with their descriptor's launch flags and their dummy-talker benchmark mode, on Seed-TTS-length workloads on one H200. That arm has no Thinker compute and no gateway hops. M* serves the full Qwen3-Omni pipeline, Thinker included, on 2 GPUs of the same node.

Table 4: Qwen3-Omni TTS: Cornserve's talker+vocoder executor (1×H200) vs. M* (full pipeline, 2×H200). RTF p50 and first-audio p50, medians of 3 repetitions. The un-pinned arm is a single run.

c	**Cornserve (as shipped, eager)**	**Cornserve (eager un-pinned, 1 run)**	**M* (full pipeline)**
1	RTF 0.63, first audio 2.8 s	RTF 0.30	RTF 0.080, 0.24 s
4	RTF 1.18, 4.7 s	RTF 0.98	RTF 0.113, 0.33 s
16	RTF 3.35, 12.8 s	–	RTF 0.168, 0.48 s

E.2 Qwen3-Omni with a tensor-parallel Thinker

Figure 13 repeats the Seed-TTS sweep of Section 4.2 with the Thinker sharded over two ranks, 3 GPUs in total, for all three systems. Our preprint reported that vLLM-Omni's tensor-parallel Thinker did not start. At v0.22.0 it does, with `--stage-init-timeout` raised to 1800 s, so the vLLM-Omni series is new in this version. At B=16 the audio throughput is 78.9 vs. 43.9 vs. 27.7 audio s/s for M*, vLLM-Omni and SGLang-Omni, and at B=32 it is 104.0 vs. 57.2 vs. 27.9. The ordering matches the non-TP deployment and the margins are similar.

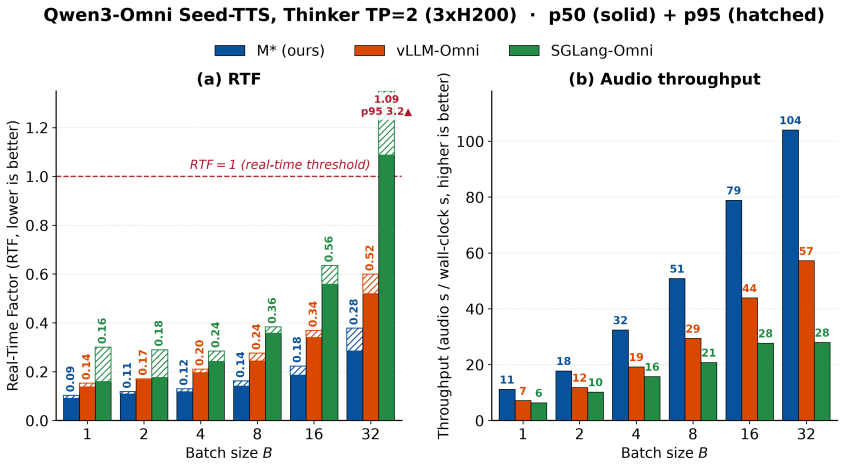

Figure 13: **Qwen3-Omni Seed-TTS, Thinker TP 2 (3 GPUs).** (a) RTF (lower is better). (b) Audio throughput (higher is better).

E.3 Current releases, speech-to-speech, and Qwen3-TTS (November 2026)

The results of record in Section 4.2 use the baseline releases pinned in July 2026, and both baselines have had new releases since. In November 2026 we re-ran Qwen3-Omni on 2×H200 with M* `main` (commit `305e80ef`), vLLM-Omni 0.32.0 and SGLang-Omni 0.2.0. The client and request format are those of Section 4.2. Each cell is one server session with $5c$ warm-up requests before the first repeat and three repeats of $\max(5c, 24)$ timed requests; we report medians across repeats. Figure 14(a, b) gives the text-to-speech results. All three systems improved since July. M* produces 2.0 to 2.2× vLLM-Omni's audio throughput at every concurrency and 1.2 to 1.5× SGLang-Omni's, whose new release closed much of the July gap (132 vs. 28 audio s/s at c=16). Figure 14(c, d) adds a speech-to-speech workload: LibriSpeech [34] validation-clean utterances as input audio, with the prompt asking the model to transcribe the clip, and text plus speech as output. M* delivers 1.9 to 2.0× vLLM-Omni's and 1.3 to 1.6× SGLang-Omni's audio throughput, with the lowest end-to-end latency at every concurrency.

Every speech-to-speech cell passes a check of the audio against the system's own output text (whisper-large-v3, mean WER at most 0.033). vLLM-Omni returns no text on this path, so its audio was checked against M*'s text for the same request, at 2.0% to 5.6% WER. In the text-to-speech protocol the model answers the Seed-TTS sentence rather than reading it, as in Section 4.2, and 3 to 7 answers per cell in every system run away, either repeating one syllable or continuing past the end of the text. Answers longer than 90 s of audio number 21 of 984 for M*, 28 for vLLM-Omni and 50 for SGLang-Omni, and all of them are included in the numbers.

We also measured Qwen3-TTS-1.7B, a SpeechLM family that an external contributor added to M* after submission, on one H200 against SGLang-Omni 0.2.0 and vLLM-Omni 0.32.0 (Figure 15). All three systems serve the same checkpoint (Qwen3-TTS-12Hz-1.7B-CustomVoice, voice `vivian`) through their OpenAI-style speech endpoints, on the first 200 Seed-TTS English sentences (256 at c=64), with $5c$ warm-up requests and three repeats per cell in one server session. Here the model reads the request text, and every cell passes a whisper-large-v3 WER check against it, with means of 1.0% to 2.6%. In audio throughput M* matches SGLang-Omni at c=4 and 8 (0.98× and 1.01×) and leads at c=1, 16, 32 and 64 (1.17×, 1.17×, 1.33× and 1.55×); against vLLM-Omni it leads by 1.5× to 3.6×. Its end-to-end latency is the lowest at c=1 and from c=16 up, and within 3% of SGLang-Omni's at c=4 and 8. SGLang-Omni has the lowest first-byte latency at every concurrency (12 to 29 ms against 25 to 71 ms for M*). Part of that difference is configuration: M*'s shipped Qwen3-TTS configuration opens with a 4-frame first chunk, 320 ms of audio, where vLLM-Omni's starts with one frame. M*'s first byte is below vLLM-Omni's at c=32 and 64.

VoxServe v2. VoxServe released version 2 in November 2026 (0.2.0, commit `b194d640`), with a Rust serving runtime, and it now serves Qwen3-TTS as well as Orpheus. We measured both against M* `main` on one H200 each (Table 5), with the same client, sentences, warm-ups and three repeats as above, and every cell passes the whisper-large-v3 WER check (0.8% to 3.2%). For Qwen3-TTS we report two VoxServe rows, because its published configuration (the flags of its v2 announcement, with the deadline scheduler and the urgent lane) collapses under a closed loop at c=64, deferring prefills until first audio takes 5 s, while its defaults hold at c=64 but are slower below. Against the better VoxServe row at each concurrency, M* delivers 1.65 to 2.57× the audio throughput with lower first-audio latency at every c. On Orpheus, under the closed-loop protocol of Section 4.2 with the LLM and the SNAC decoder on one GPU in both systems, M* delivers 1.41 to 1.56× VoxServe v2's audio throughput, with RTF p50 0.21 to 0.32 against 0.34 to 0.45. VoxServe's Qwen3-TTS was driven through

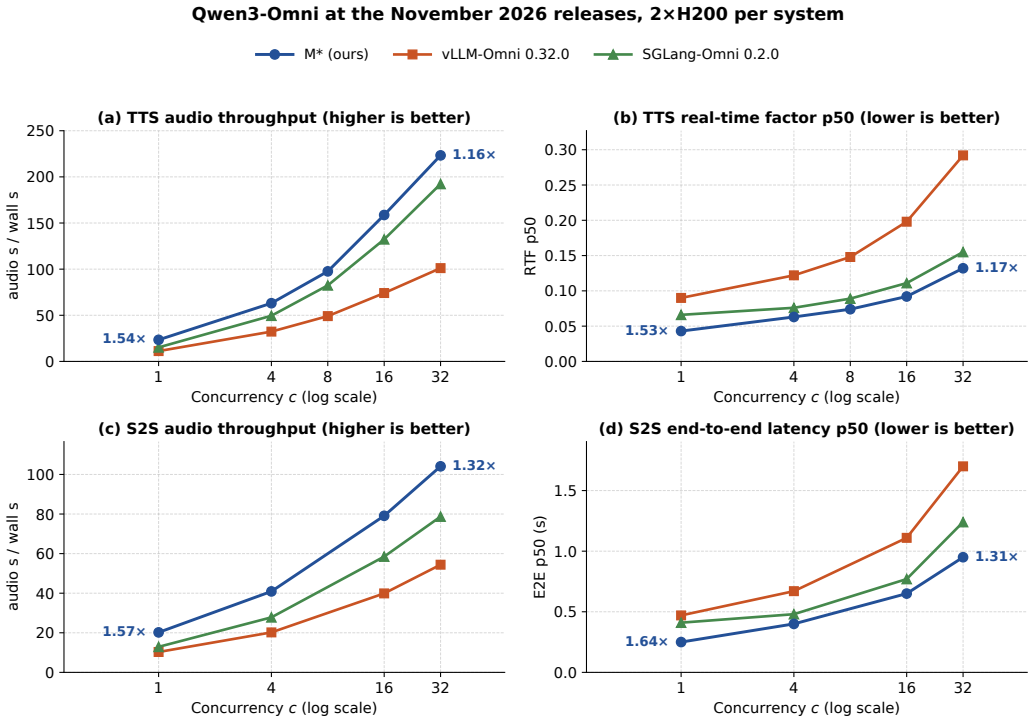

Figure 14: Qwen3-Omni at the November 2026 releases, 2×H200 per system, medians of three repeats. (a, b) Text-to-speech (TTS) on Seed-TTS: audio throughput (higher is better) and RTF p50 (lower is better). (c, d) Speech-to-speech (S2S) on LibriSpeech validation-clean: audio throughput and end-to-end latency p50. Labels give M*'s ratio to the better baseline at the ends of each curve. Audio-throughput spreads across repeats, (max−min)/median, are 0.1 to 7.5%.

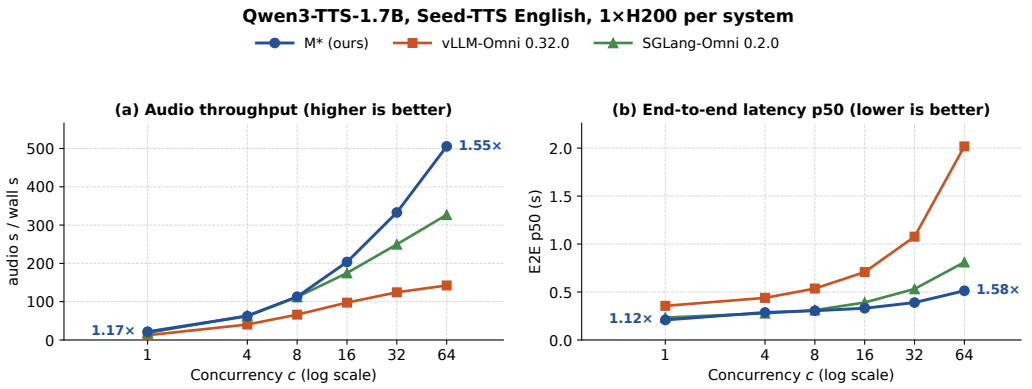

Figure 15: Qwen3-TTS-1.7B text-to-speech at the November 2026 releases, 1×H200 per system, Seed-TTS English, medians of three repeats: (a) audio throughput (higher is better) and (b) end-to-end latency p50 (lower is better). Labels give M*'s ratio to the better baseline at c=1 and 64. Throughput spreads across repeats are 0.1 to 3.4%; every cell passes the whisper-large-v3 WER check against the request text (means 1.0 to 2.6%).

its native endpoint, since its OpenAI-style route does not pass the speaker, with the client timing the same first and last audio bytes.

Table 5: M* vs. VoxServe v2 (0.2.0), 1×H200, medians of three repeats. Top: Qwen3-TTS-1.7B on Seed-TTS English, VoxServe with its published v2 flags and with its defaults; the ratio is against the better VoxServe row. Bottom: Orpheus-3B on Seed-TTS, closed loop. TTFA is first-audio p50 in ms.

Qwen3-TTS-1.7B c	M* audio s/s	VoxServe v2 flags	VoxServe defaults	M* / better	TTFA M*	TTFA VoxServe (flags / defaults)
1	21.6	10.5	7.5	2.06×	25	80 / 116
4	62.3	32.0	29.0	1.95×	40	83 / 124
8	112.6	51.6	54.5	2.07×	43	84 / 131
16	204.8	120.8	88.5	1.70×	45	101 / 157
32	328.1	198.3	134.2	1.65×	54	120 / 197
64	500.7	60.8	194.7	2.57×	73	4976 / 352

Orpheus-3B B	M* audio s/s	VoxServe v2	M* / VoxServe	RTF p50 M* / VoxServe	TTFA M*	TTFA VoxServe
1	4.7	3.0	1.56×	0.213 / 0.337	80	112
2	7.7	5.4	1.43×	0.251 / 0.361	99	123
4	14.5	9.9	1.46×	0.266 / 0.384	105	131
8	27.2	18.0	1.51×	0.278 / 0.418	111	144
16	42.3	30.0	1.41×	0.322 / 0.454	128	158

E.4 Qwen3-Omni audio-to-text and image-to-text (November 2026, B300)

Section E.3 covers the speech outputs of Qwen3-Omni. Its text outputs take the other path through the same model: the audio or vision encoder, then the Thinker, with no Talker. We measured both on two B300s per system against vLLM-Omni 0.32.0 and SGLang-Omni 0.2.0, greedy, 256 output tokens at most, prefix caching off in every compared system, closed loop at $c \in \{1, 4, 16\}$ with three repeats of 200 requests (Figure 16). Audio-to-text transcribes 200 LibriSpeech test-clean clips, scored by corpus WER; image-to-text describes 200 Food101 images, scored by whether the answer names the dish on a 20-image check (13 of 20 for M*, 14 of 20 for each baseline). M* leads every cell: 1.27×, 1.09× and 1.08× the best competitor's audio throughput and 1.26×, 1.14× and 1.15× its text throughput at c=1, 4 and 16, with the lowest first-token latency except at audio c=16, where SGLang-Omni's first token is earlier. The checkpoint answers some clips instead of transcribing them, which is why WER sits at 4 to 7% for every system; M*'s is the lowest (4.3 to 5.4% against 6.5 to 7.2%). SGLang-Omni's default radix cache was turned off for the compared rows, and its radix-on rows (never the best) are in the lane report.

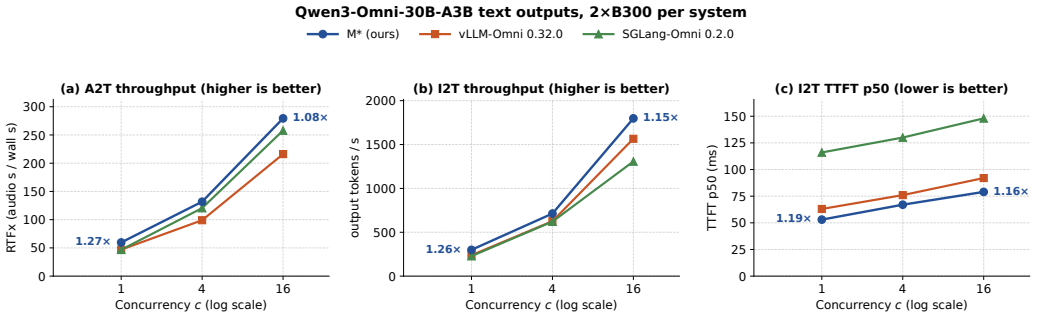

Figure 16: Qwen3-Omni-30B-A3B-Instruct text outputs, 2×B300 per system, medians of three repeats of 200 requests. (a) Audio-to-text (A2T) on 200 LibriSpeech test-clean clips, throughput in audio seconds per wall second (higher is better). (b, c) Image-to-text (I2T) on 200 Food101 images, output tokens per second (higher is better) and time to first token p50 (lower is better). Labels give M*'s ratio to the better baseline at c=1 and 16. Throughput spreads across repeats are 0.7 to 19.9% (M* A2T at c=16 is the widest).

E.5 A text-only model: Qwen3.5-0.8B (November 2026)

The Walk Graph is not only for models with many components. A plain text LLM is the one-node, one-walk case, and the same runtime serves it with the same scheduler, graphs and frontend as the composite models above. To check that this simplest case costs nothing, we measured Qwen3.5-0.8B against vLLM 0.33.0 and SGLang 0.5.23 on one H200, each with its shipped text configuration and the flags each project recommends (Figure 17). The workload is 400 fixed prompts of 512 tokens from the benchmark harness, 256 output tokens per request with EOS ignored, greedy decoding, prefix caching off in every system, closed loop at $c \in \{1, 16, 64, 128\}$.

Each system ran two server sessions, alternated with the others, with two timed passes per session after untimed warm-up passes at c=64 and 128; a cell is the median of the four runs. No request failed, and M* reproduces its own greedy outputs across sessions on 32 of 32 check prompts, while it agrees with each competitor about as often as they agree with each other (15 to 16 of 32). M* delivers the highest output throughput at every concurrency, 1.42×, 1.17×, 1.05× and 1.31× the best competitor at c=1, 16, 64 and 128, and the lowest median time to first token (9 to 45 ms against 21 to 217 ms). The c=64 margin is small, one of M*'s four runs there fell below SGLang's median, and M*'s tail inter-token latency is higher at $c \geq 64$ (p95 4.8 and 9.7 ms against 2.5 and 4.9 for SGLang). The same protocol on a B300 three days earlier gave 1.38×, 1.10×, 1.27× and 1.44×.

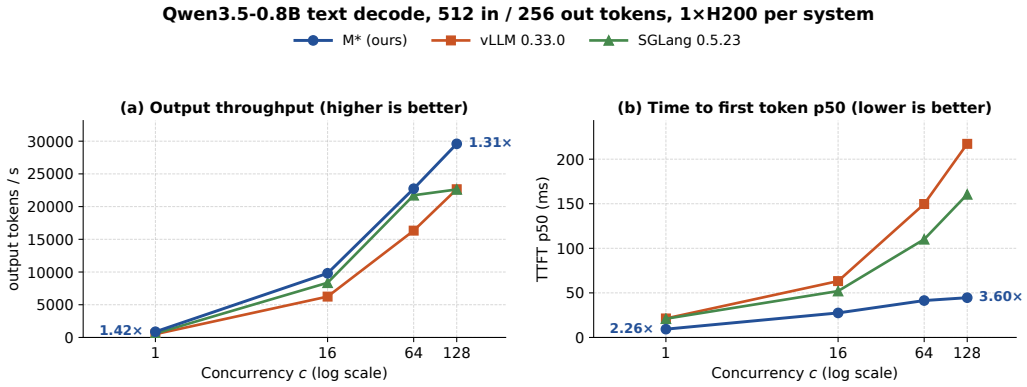

Figure 17: Qwen3.5-0.8B text decode, 1×H200 per system, 512 input and 256 output tokens, medians of four runs over two server sessions: (a) output tokens per second across the closed loop (higher is better) and (b) time to first token p50 (lower is better). Labels give M*'s ratio to the better baseline at c=1 and 128. Throughput spreads across runs are 0.7 to 9.8% for M* and 0.9 to 3.7% for the baselines.

E.6 Speech recognition: whisper-large-v3 (November 2026)

Whisper is an encoder-decoder with one audio encoder and one text decoder, a two-node walk. We measured whisper-large-v3 on the first 200 LibriSpeech test-clean clips (1,564 s of audio) against vLLM 0.33.0 and SGLang-Omni 0.2.0 on one H200, each with its shipped configuration, closed loop at $c \in \{1, 8, 32\}$ with $5c$ warm-ups and three repeats per cell in one server session (Figure 18). Every system transcribes at 1.35 to 1.37% WER, and no request failed. M* delivers 1.5 to 1.7× vLLM's throughput at every concurrency with lower latency (p50 66 to 360 ms against 112 to 552 ms). Against SGLang-Omni, not shown, it is 1.52× and 1.15× at c=1 and 8 and level at c=32 (0.98× on medians, with overlapping repeats). An in-process faster-whisper reference with the same greedy decoding reaches 40 to 61 audio seconds per second and does not scale past c=8. The same lane on one B300 gives 1.5 to 1.6× vLLM, and 0.99×, 1.08× and 1.05× SGLang-Omni, a tie at c=1 inside both systems' spreads and a lead above.

E.7 A single-component diffusion model: FLUX.2-klein-4B (November 2026)

At the other end from the composite models is a text-to-image model with one DiT loop and no second component. We measured FLUX.2-klein-4B at 1024^2, 4 steps and guidance 1.0 on 100 PartiPrompts against SGLang 0.5.23, which serves it through a diffusion-specific path with request batching (batch 8) and without, and vLLM-Omni 0.32.0, which does not batch it, on one B300 (Figure 19) and, with the same client and seeds, on one H200. Every image is seeded per prompt and checked against a diffusers 0.42 eager reference: M* reproduces the reference at 57 dB PSNR on both GPUs, while SGLang and vLLM-Omni draw different noise for the same seed (23 to 24 dB on B300, 11 dB on H200), so their PSNR measures sample divergence rather than fidelity, and CLIP scores are equal across systems (32.8). On the B300, M* leads at every point: 1.16× SGLang and 1.84× vLLM-Omni at B=1, and 1.17 to 1.24× SGLang's batched throughput and 1.9 to 2.1× vLLM-Omni's at c=4 to 16. On the H200, not shown, M* is level with SGLang at B=1 (0.331 s both), 6 to 10% below its batched throughput, and 1.4× vLLM-Omni. On the B300, vLLM-Omni refuses its FlashAttention backend and falls back to cuDNN attention, and SGLang's batched path runs eager with TF32 and reduced-precision reductions on, as it does on the H200.

E.8 Video generation: Wan2.2-TI2V-5B (November 2026, B300)

Wan2.2-TI2V-5B is a text-to-video model that an external contributor added to M* after submission (Section E.16). We measured it at 832×480×33 frames, 20 steps, guidance 5.0 and flow shift 5 on 10 VBench

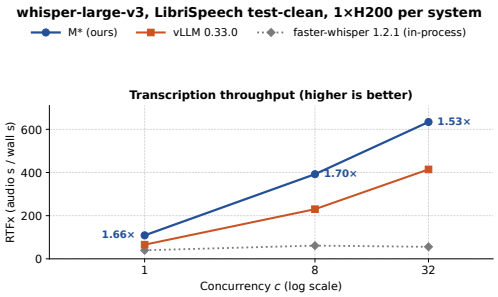

Figure 18: whisper-large-v3 ASR, 200 LibriSpeech test-clean clips, 1×H200 per system, medians of three repeats: audio seconds transcribed per wall second (↑), with M*'s ratio to vLLM. Repeat spreads 0.2 to 2.6%; corpus WER (Whisper normalizer) 1.35 to 1.37% for every system, 1.22% for faster-whisper.

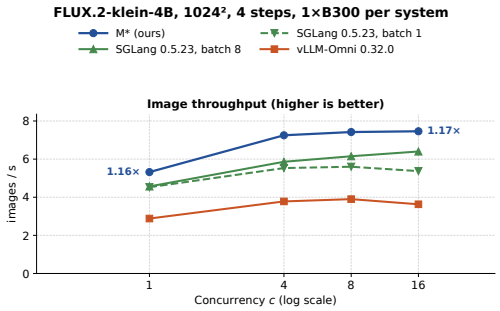

Figure 19: FLUX.2-klein-4B text-to-image, 1024^2, 4 steps, 100 PartiPrompts, 1×B300 per system (Section I), medians of three repeats (80 images each at $c \geq 4$): images per second (↑), with M*'s ratio to the better SGLang configuration. PSNR against a same-seed diffusers 0.42 eager reference: M* 57.3 dB, SGLang 22.9, vLLM-Omni 24.2; CLIP 32.8 for every system.

prompts against SGLang 0.5.23 and vLLM-Omni 0.32.0, one B300 per system, closed loop at $c \in \{1, 2, 4\}$ with three repeats of 10, 10 and 20 videos (Table 6). Outputs are checked against a diffusers 0.42 eager reference with the same seed: M* and SGLang sit at 20 to 23 dB PSNR, which is where diffusers eager itself lands across torch versions on this workload, while vLLM-Omni draws its noise on the GPU and lands at 9 dB, a different sample. M* is 1.10× SGLang and 1.56× vLLM-Omni in single-request latency and 1.18 to 1.19× SGLang and 1.24 to 1.27× vLLM-Omni in videos per minute at c=2 and 4, with the tightest spreads and an even request tail at c=4 (p95 within 2% of p50). On an H200 we have only an older round, from 3 November with M* `main 7ed765ed`, in which M* reached 0.65× the best competitor; the Blackwell-specific step kernels that account for the B300 lead do not apply there, and we did not re-measure.

Table 6: Wan2.2-TI2V-5B text-to-video, 832×480×33, 20 steps, 10 VBench prompts, 1×B300 per system, medians of three repeats. PSNR is mean over frames against a diffusers 0.42 eager reference; the spread is (max−min)/median of the throughput metric.

System	c=1 p50 (s)	videos/min c=2	videos/min c=4	spread	PSNR (dB)
M*	1.71	39.4	39.6	0.2 to 1.4%	22.6
SGLang 0.5.23	1.88	33.3	33.4	3 to 10%	20.9
vLLM-Omni 0.32.0	2.67	31.8	31.3	1.5 to 11%	9.4

E.9 Online serving: full tables

Table 7 is the open-loop Poisson sweep of Section 4.5 on the first cluster (2×H200), and Table 8 is the mixed-workload study.

The repetition protocol of Table 8 favors the baseline. vLLM-Omni gets a fresh server per repetition because its shared-server session degrades: its multimodal receiver cache raises assertion errors from the second repetition on, streaming i2t responses stop reporting first-token times, and by the seventh consecutive load point requests fail outright. M*'s medians come from one server across all repetitions.

Replication on a second cluster. To rule out host- and build-specific artifacts, we re-ran the open-loop sweep of Section 4.5 on a second cluster with 4×H100. vLLM-Omni ran as the official v0.22.0 wheel instead of a source build of the same tag, with the same client, workload and methodology. M*'s Thinker KV pool was set to 1024 pages instead of 2048 to fit next to the weights on 80 GB. The host was GPU-idle but CPU-shared, and the host load was recorded per cell. Table 9 shows the knee region. Absolute knees are host dependent, and both systems sit higher on this quieter host, but the relationships reproduce (Table 10). Peak SLO-goodput counts completions with RTF < 1 and is reached past the first-audio knee, at λ=2.5 to 2.8 for M* on the first cluster and λ=4.0 on the second, and for vLLM-Omni at λ=2.0 on the first cluster and λ=2.5 on the second. SGLang-Omni replicates as well, with λ^* about 0.5 and a marginal 0.45 to 0.5 band whose 4 to 5× spreads flip n=3 verdicts until n=6 resolves them. One of six vLLM-Omni repetitions at λ=1.8 on this cluster, below the knee region of Table 9, collapsed transiently (171 of 200 requests completed, TTFA p99 61 s, connection resets) and recovered by the following cell. The five neighboring repetitions were clean, which is why single-run tables cannot distinguish such an episode from steady-state behavior.

Table 7: Qwen3-Omni TTS, Poisson arrivals, 2×H200: median TTFA p99 (s) / RTF p99 over n repetitions. In brackets, the repetitions above the 2 s SLO. Blank cells were not run.

λ (req/s)	**M***	**vLLM-Omni**	**SGLang-Omni**
0.25	0.37 / 0.14 (n=3)	0.26 / 0.28 (n=3)	0.68 / 0.38 (n=3)
0.35	0.38 / 0.15 (n=3)		0.83 / 0.47 (n=6; 1/6)
0.45			1.32 / 0.57 (n=6; 2/6)
0.5	0.40 / 0.15 (n=3)	0.32 / 0.33 (n=3)	1.37 / 0.60 (n=6; 2/6)
0.6			18.5 / 4.81 (n=3; 3/3)
1.05		0.53 / 0.58 (n=3)	
1.4	0.65 / 0.20 (n=6)	0.60 / 0.63 (n=3)	
1.8		0.90 / 0.97 (n=6)	
2.0	0.98 / 0.27 (n=6)	0.88 / 0.97 (n=6; spread 1.3×)	
2.1	1.21 / 0.30 (n=11; 0/11)		
2.25	1.14 / 0.34 (n=11; 1/11)	4.94 / 1.22 (n=6; 4/6; spread 11×)	
2.5	1.97 / 0.59 (n=6; 3/6)	10.7 / 1.83 (n=3; 3/3)	
2.8	2.91 / 0.89 (n=11; 10/11)	20.6 / 2.86 (n=6; 6/6)	
4.0	15.9 / 1.67 (n=3; 3/3)		

Table 8: BAGEL mixed online workload (t2i:i2t:i2i = 1:2:1, Poisson): SLO attainment [min–max over 3 repetitions] and goodput (req/s).

Deployment (GPUs)	λ	**M***	**vLLM-Omni**
replicated vs. Thinker/DiT split (2×H100)	0.18	52% [52–68]; 0.113	41% [32–52]; 0.060
replicated vs. Thinker/DiT split (2×H100)	0.26	46% [43–67]; 0.096	32% [29–35]; 0.045
CFG-parallel vs. CFG-parallel + workaround (3×H200)	0.18	89% [80–100]; 0.182	36% [32–64]; 0.053
CFG-parallel vs. CFG-parallel + workaround (3×H200)	0.26	65% [60–78]; 0.157	16% [5–22]; 0.024

Table 9: Second cluster (4×H100): median TTFA p99 (s) / RTF p99 over n repetitions. In brackets, the repetitions above the 2 s SLO.

λ (req/s)	**M***	**vLLM-Omni**
2.0	0.66 / 0.20 (n=3)	1.08 / 0.82 (n=6)
2.25	0.72 / 0.24 (n=3)	1.29 / 0.94 (n=6; 1/6)
2.5	0.77 / 0.25 (n=6; 0/6)	1.58 / 1.00 (n=6; 1/6)
2.8	1.64 / 0.44 (n=6; 0/6)	12.7 / 1.54 (n=3; 3/3)
4.0	5.23 / 0.98 (n=3; 3/3)	

Table 10: The same relationships on both clusters.

	First cluster (2×H200)	**Second cluster (4×H100)**
λ^*, M* / vLLM-Omni / SGLang-Omni	2.25 / 2.0 / 0.5	2.8 / 2.5 / 0.5
λ^* ratio, M* over vLLM-Omni	1.13×	1.12×
peak SLO-goodput, M* vs. vLLM-Omni	2.13–2.16 vs. 1.38 (1.56×)	2.49 vs. 1.66 (1.50×)
RTF p99 at matched λ=2.0	0.27 vs. 0.97 (3.6×)	0.20 vs. 0.82 (4.0×)

Bursty arrivals. With gamma arrivals at λ=1.55 req/s on the first cluster, we measured M* at CV 1, 2 and 4 with 3, 6 and 6 repetitions. The medians are TTFA p50 0.42, 0.58 and 1.24 s, TTFA p99 1.63, 2.05 and 3.18 s, and RTF p99 0.24, 0.52 and 0.89. Every repetition completed all requests, and 14 of 15 held RTF p99 below 1.0 with at least 99% real-time attainment. The one exception (70% attainment) was measured immediately after a cold server start and is included in the medians.

E.10 Knob sensitivity

M*'s tunables are the chunk policies of streaming edges (Orpheus `SlidingWindow` W=28, S=7; Qwen3-Omni Thinker→Talker `Fixed` K=1; Talker→Code2Wav `LeftContext` C=25, L=25), the CUDA-graph capture buckets (autoregressive nodes $\{1, 2, 4, 8, 16, 32, 64\}$, stateless nodes $\{1, 2, 4, 8, 16\}$, overridable per configuration), the KV pool (`page_size`=128, `max_num_pages`, `cpu_offload_pages`), the scheduler (`max_concurrent_requests` and the speculation knobs `MSTAR_NUM_SLOTS`, `MSTAR_PRE_PLAN_SPEC`, `MSTAR_MAX_CONSECUTIVE_SPEC_STEPS`), and the per-edge transport (shared memory, TCP or RDMA). We swept the ones that Table 1 does not cover at B=8 on Seed-TTS, with two repetitions per cell (Table 11). Vocoder-chunk granularity is the one knob that moves first-audio latency. It does so monotonically in both speech models and at unchanged RTF. Coarsening the capture buckets to $\{1, 8, 64\}$ moves TTFA p50 by $+3\%$ on Orpheus and $+9\%$ on Qwen3-Omni and RTF p50 by $+5\%$ and $+3\%$, comparable to the 5 to 9% spread between two repetitions of the default configuration. The Thinker→Talker chunk is fixed at 1 by the streaming cadence the Talker's KV extension is built around. The shipped defaults sit mid-curve with headroom in both directions and were not tuned per benchmark.

Table 11: Knob sensitivity (B=8, Seed-TTS, two repetitions per cell).

Model	Knob	Change	Effect
Orpheus	vocoder window W	28 → 14 tokens	TTFA p50 0.19 → 0.11 s; RTF unchanged
Qwen3-Omni	Code2Wav chunk C	25 → 15 frames	TTFA p50 0.40 → 0.30 s; RTF flat
Qwen3-Omni	Code2Wav chunk C	25 → 40 frames	TTFA p50 0.40 → 0.65 s; RTF flat
Orpheus	capture buckets	default → $\{1, 8, 64\}$	TTFA p50 $+3\%$; RTF p50 $+5\%$
Qwen3-Omni	capture buckets	default → $\{1, 8, 64\}$	TTFA p50 $+9\%$; RTF p50 $+3\%$

E.11 Overheads of fine-grained graph execution

These numbers come from the built-in per-(node, walk) profiler under the settings of Section 4. In BAGEL's CFG-parallel I2I loop the branch forwards take 86 to 106 ms per step and the cross-branch combine step 1.6 ms, about 1.5%, which is the price of CFG as graph structure. Scheduling plus pre- and post-processing add 1 to 3 ms per node execution on these 100 ms steps. On small steps the overhead is visible but bounded: Orpheus decode takes 6.1 ms of wall time for a 4.8 ms forward, and the Qwen3-Omni Talker 9.8 ms for 8.4 ms. The asynchronous pipeline overlaps most of it. Turning it off costs 1.18 to 1.56× end to end (Table 1), an upper bound on the residual dispatch cost. Disaggregating Qwen3-Omni's Thinker prefill from its decode adds 14 to 20 ms of first-audio latency unloaded (Section E.14).

E.12 Transport: shared memory vs. TCP

Table 12 forces every cross-worker edge of the same deployment onto Mooncake TCP and compares against shared memory on one H200 node, with single runs per cell. The Qwen3-Omni shared-memory column is the deployment of Figure 5; under TCP its median RTF stays below 0.4 through B=32, while its RTF p99 at B=32 is 1.49. For Orpheus the cost at B=16 depends on the placement: $+4.8\%$ RTF with the LLM and the SNAC decoder on one rank (0.603 to 0.632), $+10\%$ with a prefill/decode split, and $+24\%$ with the decoder on its own rank. This study ran at commit `72d53f76`, later than the sweep of Figure 6, so its single-rank RTF differs from the 0.58 there. On 2 H100s a BAGEL prefill/decode split pays $+5.5\%$ E2E p50 and -5.5% throughput at B=4 under forced TCP, with outputs byte-identical across transports (8/8).

E.13 The multi-host code path

We run the same deployments as two process trees, each with its own node agent, joined through the TCP control plane and the Mooncake TCP tensor plane. This is the code path a physically two-host deployment takes. We compare against a single-tree shared-memory twin on the same GPUs. A BAGEL prefill/decode split across the two trees keeps pinned-seed outputs byte-identical, 8/8 on H200 and 7/8 on H100, including two 2 MB generated images whose tensors cross the tree boundary twice. The one H100 difference is a near-tie token flip from kernel autotuning. The split costs $+1.2\%$ E2E p50 at B=1, $+3.0\%$ at B=4, and -4.7% throughput, with TTFT p50 moving from 20 to 21 ms. An Orpheus streaming split places the decode LLM in one tree and the prefill and vocoder in the other, so the token stream crosses the boundary 48 to 55 times per utterance. It costs $+22$ to

Table 12: Shared memory vs. forced TCP on the same deployment (RTF p50; audio throughput in audio s/s).

Deployment	B	**SHM**	**TCP**
Qwen3-Omni, 2 GPUs	1	0.079; 12.5	0.083; 11.9
Qwen3-Omni, 2 GPUs	16	0.169; 84.7	0.234; 61.8
Qwen3-Omni, 2 GPUs	32	0.299; 99.7	0.393; 74.4
Orpheus, LLM / SNAC on separate ranks	1	0.283; 3.5	0.333; 3.0
Orpheus, LLM / SNAC on separate ranks	16	0.773; 19.6	0.961; 15.2
Orpheus, prefill / decode split	1	0.306; 3.3	0.326; 3.1
Orpheus, prefill / decode split	16	0.624; 21.8	0.686; 21.2

30 ms TTFA, $+4$ ms per chunk of inter-token latency and $+16$ to 17% RTF, and it stays real-time at RTF 0.38 to 0.42. The forced-TCP single-tree numbers match the two-tree twin's (RTF 0.374 vs. 0.377), so the transport is the cost and the process-tree split adds close to nothing. Replicating Qwen3-Omni across the two trees through the conductor's data-parallel routing restores real-time streaming at $B{=}32$ (RTF p99 1.03 $\rightarrow$ 0.77, TTFA p99 -57%) with balanced replicas at 51% and 54% SM utilization. Raw request throughput rises only 13%, because the shared streaming ingress, not the GPUs (57% SM single-replica), is the ceiling. Our shared clusters' firewall allows arbitrary TCP connections in one direction only, which blocked a physically two-box deployment. The code path exercised above (node-agent join, TCP control plane, cross-tree Mooncake transfers) is identical to the physical case, and the transport cost is what Table 12 measures.

E.14 Placement as a configuration knob

We compared three configuration-only placements of Qwen3-Omni under a long-audio mixed load: 20 to 240 s speech-to-text prefills interleaved with streaming TTS, seeded Poisson arrivals, identical sequences across arms. The arms are colocated, the shipped prefill/decode-disaggregated config, and that config with the vocoder moved next to the Talker (one line). Below saturation ($\lambda \leq 1.2$ req/s) the TTS tails are comparable across arms. At $\lambda{=}1.8$ with 240 s inputs, the colocated placement's first-audio latency grows without bound within the window (p50 515 $\rightarrow$ 975 ms between the two halves, p99 3.6 s). The disaggregated placements keep the decode rank at about 50% utilization with bounded worst stalls (inter-token maximum 1.0 to 1.6 s), and pressure moves to the prefill rank instead. Moving the vocoder cut the worst stall by 37% and the inter-token p99 by 33%. The sustainable rate under a 2 s TTFA SLO depends on prefill length. At 20 s of input audio every placement sustains the full rate grid, up to the top tested rate of 2.4 req/s. At 66 s the colocated arm sustains 2.1 req/s but not 2.4, the shipped disaggregated config sustains 1.2 but not 1.5, and the vocoder-moved one sustains 2.4, the top rung. At 127 s all arms sit near 1.5 to 1.8. At about four-minute prefills the ordering flips, as the colocated arm develops multi-second decode stalls that the disaggregated placement avoids. Unloaded, the disaggregated placements cut speech-to-text TTFT by 2.2 to $2.8\times$ (290 and 227 ms vs. 630 ms), because the encoder and prefill share a rank, and add 14 to 20 ms to first-audio latency.

E.15 Breadth: the Cosmos3 video world model

After submission we integrated NVIDIA's Cosmos3 video world model (Nano and Super) and compared against vLLM-Omni's official Cosmos3 support, which first shipped in their v0.24 line (we used v0.24.1). Both ran on the same H100 node with the same prompts, steps, guidance and seed, at $B{=}1$, with medians of 10 requests for images and 3 for video. Nano text-to-image p50 is 0.56, 1.18 and 2.41 s for M* against 1.62, 1.69 and 2.45 s at $320{\times}192$, 480p and 720p. Super text-to-image is 1.35, 3.04 and 5.92 s against 3.98, 4.14 and 6.48 s. At matched two-GPU deployments M* serves Nano text-to-video at 480p (189 frames) in 32.9 s against 39.1 s, and at 720p in 112.7 s against 124.0 s. Part of that comes from serving the model sequence-parallel, a partitioning their integration does not expose (tensor-parallel only), and a TP 2 $\times$ SP 2 deployment is a configuration change. The walk-graph program also expresses windowed autoregressive video serving and the DROID action-policy variant of the same backbone under the same runtime.

E.16 Developer effort

Per-model integration cost on `main`, counting the graph, state machine and adapters, with the roughly 10,000-line runtime reused: BAGEL 4,700 lines, Qwen3-Omni 5,911, Orpheus 1,535, V-JEPA 2 4,189, $\pi_{0.5}$ 2,115, Whisper 942, Higgs-Audio 1,029, Cosmos3 5,182. External contributors added Qwen3-TTS (3,311 lines, both commits on the same day, CI green) and Wan2.2 text-to-video (6,143 lines, a second video family, about 25 hours between first and last commit) without runtime changes. Ten model families are on `main` today, eight in-house and two contributed, against the five in the preprint.

F Broader Impacts

Our work has several potential positive impacts. By improving the efficiency, scalability, and accessibility of multimodal model deployment, the framework can enable broader access to advanced AI capabilities in applications such as accessibility technologies, scientific computing, education, healthcare, and real-time decision support systems. Improved serving efficiency may also reduce infrastructure cost and energy consumption per inference, lowering barriers for smaller organizations and researchers to deploy multimodal AI systems. At the same time, these capabilities may introduce negative societal impacts if misused. More efficient deployment of multimodal models could enable large-scale surveillance, automated misinformation generation, deepfakes, or invasive analysis of visual and audio data. Incorrect model outputs in high-stakes applications may also propagate harmful or biased decisions, particularly if deployed without appropriate human oversight. To mitigate these risks, we emphasize that model authors should deploy such instances with appropriate safeguards, including privacy protections, usage restrictions for sensitive applications, monitoring and auditing mechanisms, and careful human evaluation before deployment in safety-critical settings.

G Related Work

Token-centric LLM serving. vLLM [27], SGLang [58], and Orca [54] target autoregressive text generation with optimizations such as continuous batching, paged attention, and radix caching. M* reuses these optimizations and generalizes them to other modalities (§3.3). DistServe [59], Splitwise [35], and Mooncake [37] disaggregate prefill from decode, a placement M* expresses through configuration (§3.2).

Multimodal serving. vLLM-Omni [53] and SGLang-Omni [40] represent a model as a fixed DAG of stages, where each stage executes as a separate inference engine. ModServe [38] and EPDServe [41] support flexible disaggregation but only for the image preprocessing and encoder components. Cornserve [12] serves any-to-any models by recovering a dependency graph of Python tasks via record-and-replay and plans their placement [31]. M*'s Walk Graph captures computation ahead of time at the granularity of one denoising iteration rather than a full request, which enables compile-time optimizations across components (Section 4.7).

Dataflow and composition in serving and training systems. Ray Serve [32, 45] composes independently deployed models through Python calls between actors. Triton [33] composes static ensembles, with control flow in Python outside the optimized graph. TensorFlow's `cond` and `while_loop` [55] made control flow a first-class graph operation, and Pathways [6] runs dataflow for training. The Walk Graph inherits this lineage and differs in granularity and contract (§1): nodes are sub-components of one model rather than deployed services, and control flow is confined to walk boundaries, so the runtime itself schedules, batches, captures, and places a walk's components. EEVEE [22] multiplexes the modules of understanding models on one GPU with independently tuned batching, which could run inside a single Walk Graph node, and it does not address generative loops or streaming.

Modality-specific serving. Most modality-specific inference optimizations can be integrated into M*. VoxServe [25] unifies SpeechLM serving with a streaming-aware scheduler, which M* expresses through streaming edges. FastVideo [57, 56] introduces sparse attention for video generation. xDiT [17] proposes DiT parallelism strategies [18, 16, 43] that flexible placement can host. Inferix [44] combines KV-cache management with block-wise diffusion for long-video world models, a hybrid pipeline the Walk Graph expresses as one graph. FlashDrive [28] accelerates VLA inference with KV-cache reuse and adaptive flow-matching steps.

H Limitations

First, although we evaluate our framework across a diverse set of multimodal models and workloads, it is infeasible to exhaustively test all existing and emerging multimodal architectures, modalities, and deployment settings. As a result, performance characteristics and system behavior may differ for models, serving patterns, or hardware configurations not included in our evaluation. Second, some evaluation settings that we ran are not included because of correctness or performance issues found in baselines, which may have been due to evaluating these systems in previously untested settings. Third, every measurement in this paper is single-host. The multi-host code path (per-host node agents, a TCP control plane, and the Mooncake TCP tensor plane) was exercised as two process trees on one host, because our clusters do not share a network fabric, and the RDMA transport is untested. Fourth, baselines are pinned to the released versions named in Section I. Both baselines have had further releases since. Section E.3 re-measures Qwen3-Omni and adds Qwen3-TTS at the November 2026 releases, and BAGEL, Orpheus and V-JEPA 2 were not re-measured.

In addition, we plan to explore several aspects of the system design in future work. For example, the set of supported runtime engines could be expanded or modified as new models and multimodal tasks emerge, and our current streaming edge policy and edge abstractions represent only a limited number of strategies in a broader design space. Similarly, although M* supports tensor-parallel sharding of graph components, our implementation does not yet explore additional forms of parallelism (e.g., sequence, context, or pipeline parallelism), more

advanced asynchronous worker scheduling and modality or model-specific optimization strategies such as sparse attention techniques, or other potential system-level optimizations beyond those described in the paper. These limitations suggest opportunities for future work and further evaluation across a wider range of models, hardware platforms, and distributed serving configurations.

I Artifact and Code Reproducibility Details

This appendix consolidates the information needed to reproduce the results in §4: hardware, software stack, per-experiment configurations, and the licenses of all third-party code, model weights, and datasets used. M* is open source under the Apache 2.0 license at `https://github.com/mstar-project/mstar`, with documentation at `https://m-star.org/mstar/`. The repository carries the model packages, the placement configurations named below, the benchmark harness and clients under `benchmark/`, and the commits listed here are the versions of record for each experiment.

Hardware. Experiments run on two single-node clusters with intra-node NVLink and shared-memory (SHM) tensor transport between worker processes (TCP where stated); no inter-node communication is exercised.

- One $4\times$NVIDIA H100 (80 GB) node, used for BAGEL (3-GPU CFG-parallel generation with one rank per CFG branch and encoders / VAE colocated with rank 0; single-GPU generation and I2T; the 2-GPU mixed online workload), for the BAGEL multi-host and transport twins, and for the second online-serving cluster.
- One $8\times$NVIDIA H200 (141 GB) node, used for Qwen3-Omni on a 2-GPU disaggregated deployment (Thinker on one rank, Talker+Code2Wav on the other) and its Thinker-TP 2 variant, for the online, bursty and placement studies, for Orpheus on a single GPU, for V-JEPA 2 on a single GPU, for the 3-GPU BAGEL mixed online workload, and for the Cornserve comparison.
- One $2\times$NVIDIA B300 SXM6 pod (RunPod; Intel Xeon 6767P host, driver 595.91.07, CUDA 13.2), used only for the November 2026 measurements of Sections E.4, E.7 and E.8 and the B300 panel of Section E.6, one GPU per system (two for Qwen3-Omni), with the server pinned to 8 physical cores and the client to 4.

Versions. All baseline numbers in this version use released baseline versions: vLLM-Omni v0.22.0 (commit `963ba1ab`, vLLM core v0.22.0) and SGLang-Omni 0.1.0 (commit `8e272bcb`); the Cosmos3 comparison in Section E.15 uses vLLM-Omni v0.24.1, where their Cosmos3 support first shipped, and the Cornserve measurement uses their commit `6d04df90` (July 2026) with their shipped Qwen3-Omni descriptor. The November 2026 measurements in Section E.3 use vLLM-Omni 0.32.0 (vLLM core 0.32.0), SGLang-Omni 0.2.0 (sglang 0.5.22), and M* `main` at `305e80ef`, with the repo's `qwen3omni_2gpu_large_kv.yaml` and `qwen3tts_1p7b.yaml` configurations and the baselines' shipped deployments; the text-model measurement in Section E.5 uses vLLM 0.33.0, SGLang 0.5.23 and the same M* commit with `qwen3_5_0.8b.yaml` (prefix caching off) and the Rust frontend; the VoxServe v2 comparison uses vox-serve 0.2.0 (commit `b194d640`) and, for Orpheus, M*'s one-GPU `orpheus_colocated.yaml`; the FLUX.2-klein measurement in Section E.7 uses sglang 0.5.23, vLLM-Omni 0.32.0 and diffusers 0.42 references, with M*'s shipped `flux2_klein.yaml`; the Whisper measurement in Section E.6 uses vLLM 0.33.0, SGLang-Omni 0.2.0 and faster-whisper 1.2.1 with M*'s shipped `whisper_large.yaml`; the Wan2.2 measurement in Section E.8 uses sglang 0.5.23, vLLM-Omni 0.32.0 and diffusers 0.42 references, with M*'s shipped `wan22.yaml` and the two request sizes listed under `compile_dit_sizes`; the Qwen3-Omni text-output measurement in Section E.4 uses vLLM-Omni 0.32.0 and SGLang-Omni 0.2.0 with its radix cache off, and M*'s shipped two-GPU configuration. M* commits: Qwen3-Omni closed-loop `0566c3e7`; the TTS ablation `713df964` (full and single-rank arms), `32b0fe04` (Talker graphs off) and `633abbd4` (asynchronous pipeline off); the knob sweeps `73093d4e`, `aa192aec` and `115fb8e5`; BAGEL generation, I2T and ablations `11f7d4b0` (branch `bagel-perf-tuning-2`, since merged); online and mixed workloads `c0faa987` / `454477f5` (the 3-GPU mixed-workload cells on H200 ran vLLM-Omni `963ba1ab` on vLLM core 0.22.1rc2); transport and two-process-tree twins `72d53f76` / `071e43f4`; V-JEPA 2 harness `629cba65`; Cosmos3 `6408b50c`. The preprint measured vLLM-Omni at commit `c08959ee` (core v0.21.0rc2), SGLang-Omni at `e3e9413a`, and M* at `4a3960be`; Orpheus and VoxServe numbers are unchanged from the preprint (VoxServe v0.1.0, M* `d023ee7d`). Two preprint numbers were measurement errors and are gone from this version: the $12\times$ BAGEL I2T gap came from vLLM-Omni's out-of-the-box behavior of running the diffusion stage for every request, and the $12.5\times$ V-JEPA 2 speedup at $H{=}30$ compared a rollout silently clamped to 16 steps against the baseline's 30.

Software stack. M* requires Python 3.12 and PyTorch with CUDA. The engine layer uses FlashInfer [52] for paged-attention prefill and decode, HuggingFace Transformers [48] for tokenization and weight loading, and `torchaudio` / `torchcodec` for audio and video decoding.

Workload configurations. Every experiment is driven by our harness: `num_warmup` warm-up requests at sequential concurrency, followed by `num_requests` timed requests in the workload's configured concurrency mode (default: `offline`, sized waves of B requests). Defaults are `num_warmup`=3 and the per-model settings

below; greedy decoding (temperature = 0) is forced on every sub-model. Token sequences then match across systems up to numerical differences, which show up as the per-system runaway counts in Section E.3. Closed-loop configurations use 5 warm-up requests (more for V-JEPA 2, whose rollouts are short, while BAGEL I2T repetitions share one warm server; see Section 4) followed by at least $\max(5B, 12)$ timed requests for speech and text outputs and 10 for image outputs; the open-loop Poisson sweep uses 200 requests per run, repeated as stated in Section 4.5.

- **BAGEL-7B-MoT** (`configs/bagel_cfg_parallel.yaml`): 1024^2 output, 50-step rectified-flow schedule, `cfg_img_scale=2` / `cfg_renorm_type=text_channel` on I2I; T2I and I2I at $B{=}1$ on VBench prompts, I2T at $B \in \{1, 2, 4, 8, 16\}$ on Food101 images with seeded per-request output budgets (16–128, 64–256 or 128–512 tokens) and `ignore_eos` on both systems; vLLM-Omni I2T requests carry `"modalities": ["text"]` so that its diffusion stage is skipped.
- **Qwen3-Omni-30B-A3B-Instruct** (`configs/qwen3omni_2gpu.yaml`): Seed-TTS evaluation set (English), `max_tokens`=256, `thinker/talker/cp_temperature`=0, system prompt matching vLLM-Omni's official Qwen3-Omni examples; $B \in \{1, 2, 4, 8, 16, 32\}$; vLLM-Omni started with `--stage-init-timeout 1800`.
- **Orpheus-3B** (`configs/orpheus.yaml`): LLM and SNAC decoder colocated on one H200, the placement of Figure 6; 3-node walk (Embeddings $\rightarrow$ LLM $\rightarrow$ SNAC), $B \in \{1, 2, 4, 8, 16\}$. The two-rank and prefill/decode placements appear only in the transport study (Section E.12).
- **V-JEPA 2 ViT-g (AC)** (`configs/vjepa2_ac.yaml`): DROID episodes (first 50), 8 frames per request at 256×256, bf16, sequential ($B{=}1$), rollout horizons $H \in \{4, 15, 30\}$ with `max_rollout_horizon` raised above 30. The upstream Meta vjepa2 baseline reproduces the verbatim driver pattern from `notebooks/energy_landscape_example.ipynb` (Cells 5–6) in its default fp32; the KV-cache control patches only its rollout loop and reproduces upstream's outputs.

Software. Table 13 lists the third-party systems used as baselines or as direct dependencies of M*'s engine layer.

Table 13: Software / code licenses.

Asset and version	URL	License
vLLM-Omni v0.22.0 (vllm core v0.22.0) [53]	`https://github.com/vllm-project/vllm-omni`	Apache 2.0
SGLang-Omni 0.1.0 (commit 8e272bcb) [40]	`https://github.com/sgl-project/sglang-omni`	Apache 2.0
vLLM-Omni v0.24.1 (Cosmos3) and 0.32.0 (Section E.3)	`https://github.com/vllm-project/vllm-omni`	Apache 2.0
SGLang-Omni 0.2.0 (Section E.3)	`https://github.com/sgl-project/sglang-omni`	Apache 2.0
vLLM 0.33.0 [27] (Section E.5)	`https://github.com/vllm-project/vllm`	Apache 2.0
SGLang 0.5.23 [58] (Section E.5)	`https://github.com/sgl-project/sglang`	Apache 2.0
Cornserve (commit 6d04df90) [12]	`https://github.com/cornserve-ai/cornserve`	Apache 2.0
VoxServe v0.1.0 [25]	`https://github.com/vox-serve/vox-serve`	Apache 2.0
VoxServe v2 (0.2.0, commit b194d640; Section E.3)	`https://github.com/vox-serve/vox-serve`	Apache 2.0
Meta vjepa2 (upstream baseline) [4]	`https://github.com/facebookresearch/vjepa2`	Apache 2.0
BAGEL reference implementation [13]	`https://github.com/bytedance-seed/Bagel`	Apache 2.0
Orpheus-TTS reference [10]	`https://github.com/canopyai/Orpheus-TTS`	Apache 2.0
openpi ($\pi_{0.5}$ reference) [7]	`https://github.com/Physical-Intelligence/openpi`	Apache 2.0
SNAC (audio codec) [42]	`https://github.com/hubertsiuzdak/snac`	MIT
HuggingFace Transformers [48]	`https://github.com/huggingface/transformers`	Apache 2.0
HuggingFace Diffusers [46]	`https://github.com/huggingface/diffusers`	Apache 2.0
FlashInfer [52]	`https://github.com/flashinfer-ai/flashinfer`	Apache 2.0
Mooncake transfer engine [37]	`https://github.com/kvcache-ai/Mooncake`	Apache 2.0
faster-whisper 1.2.1 (Section E.6)	`https://github.com/SYSTRAN/faster-whisper`	MIT

Model weights. Table 14 lists pretrained checkpoints used in the evaluation, with HuggingFace mirrors (or upstream URLs) and licenses. Checkpoints are downloaded from these locations on first use; M* performs no further fine-tuning. The V-JEPA 2 AC checkpoint is the encoder+action-conditioned predictor bundle published on FAIR S3 (HuggingFace does not host an AC variant).

Datasets and benchmarks. Table 15 lists the evaluation datasets.

Table 14: Model checkpoint licenses.

Asset	URL	License
BAGEL [13]	https://huggingface.co/ByteDance-Seed/BAGEL-7B-MoT	Apache 2.0
Qwen3-Omni [51]	https://huggingface.co/Qwen/Qwen3-Omni-30B-A3B-Instruct	Apache 2.0
Orpheus [10]	https://huggingface.co/canopylabs/orpheus-3b-0.1-ft	Apache 2.0
V-JEPA 2 [4]	https://huggingface.co/facebook/vjepa2-vitg-fpc64-256	Apache 2.0
Qwen3-TTS 1.7B (Section E.3)	https://hf.co/Qwen/Qwen3-TTS-12Hz-1.7B-CustomVoice	Apache 2.0
Cosmos3 Nano, Super (Section E.15)	https://hf.co/nvidia/Cosmos3-Nano, -Super	OpenMDW 1.1
whisper-large-v3 [39] (judge; Section E.6)	https://hf.co/openai/whisper-large-v3	Apache 2.0
Qwen3.5-0.8B (Section E.5)	https://hf.co/Qwen/Qwen3.5-0.8B	Apache 2.0
FLUX.2-klein-4B (Section E.7)	https://hf.co/black-forest-labs/FLUX.2-klein-4B	Apache 2.0
Wan2.2-TI2V-5B (Section E.8)	https://hf.co/Wan-AI/Wan2.2-TI2V-5B-Diffusers	Apache 2.0

Table 15: Dataset / benchmark licenses.

Asset	URL	License
VBench [24]	https://github.com/Vchitect/VBench	Apache 2.0
Seed-TTS [3]	https://github.com/BytedanceSpeech/seed-tts-eval	CC BY 4.0
DROID [26]	https://huggingface.co/datasets/lerobot/droid_100	MIT
Food-101 [8]	https://huggingface.co/datasets/ethz/food101	Foodspotting terms (research)
LibriSpeech [34]	https://www.openslr.org/12	CC BY 4.0
PartiPrompts (Section E.7)	https://github.com/google-research/parti	Apache 2.0

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and §1 state the two contributions, the Walk Graph abstraction and the M* runtime, and every measured gain is quoted with the baseline version, batch size or concurrency, and placement at which it was taken (§4). Section H states the scope of the evaluation.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section H discusses model coverage, the baseline settings that were excluded, the single-host scope of every measurement, the pinned baseline versions, and the parallelism strategies the implementation does not yet cover.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.

- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper has no theoretical results.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: §4 states the metrics and the closed-loop and open-loop protocols. Section I lists the hardware, the baseline versions and the M* commits of the main experiments, the placements, the workloads and prompts, and the datasets. Two of the placement files appear in Section C.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.

(b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: M*, its model packages, placement configurations, and benchmark harness are public under the Apache 2.0 license at the repository named in Section I, which also lists the commit of record for each experiment. The baselines are public releases pinned by version and commit, and every dataset is public and listed with its license in Section I.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: No model is trained. The serving settings that determine each result, that is placements, batch sizes, concurrency levels, arrival processes, SLO targets, warm-up and request counts, resolutions, and audio lengths, are given in §4, Section E, and Section I.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Open-loop results are medians over n repetitions with the spread reported and n stated per cell (Section E.9), and figures show p50 and p95 of per-request latency. §4 states which closed-loop cells are means of five runs and which are single runs, and the November 2026 measurements in Section E.3 are medians of three repetitions with the relative spread across repeats reported for text-to-speech.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section I names the two nodes (one with four H100 80 GB GPUs and one with eight H200 141 GB GPUs), the GPUs each experiment uses, and the request counts per cell. Each cell completes in minutes on one to three GPUs, so the reported evaluation is a small fraction of the compute spent on development and on preliminary runs that are not reported.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work is a serving system evaluated on public models and public datasets, and it conforms to the Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Section F discusses both.

    Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper releases a serving runtime, not models or data. The checkpoints it serves are third-party releases that stay under their own licenses and safeguards.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Section I cites and lists every third-party system, checkpoint, and dataset with its version, URL, and license.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The M* source, model packages, configurations, and benchmark harness are released under the Apache 2.0 license with documentation at the URLs in Section I.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper involves neither crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your institution or country) were obtained?

    Answer: [N/A]

    Justification: The paper involves neither crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs are not a component of the method. Language and multimodal models appear only as the workloads that the system serves.

    Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to the NeurIPS Main Track Handbook (`https://neurips.cc/Conferences/2026/MainTrackHandbook`) for what should or should not be described.