arXiv:submit/7700634 [cs.LG] 10 Jun 2026

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* that integrate diverse components such as vision encoders, language backbones, diffusion and flow heads, audio codecs, action generators, and world-model predictors. Such architectures underpin a broad class of multimodal models, including unified multimodal models, omni models, speech-language models, vision-language-action policies, and world models. However, existing model serving frameworks were built on narrow assumptions about model structure, making them ill-suited to accommodate this new architectural 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 find that it achieves, on average, 20% lower end-to-end latency than vLLM-Omni for text-to-image workloads on BAGEL, while delivering up to 2.9× lower real-time factor and 2.7× higher throughput for text-to-speech workloads on Qwen3-Omni. M* also outperforms the V-JEPA 2-AC rollout baseline for robotic planning by up to 12.5×. 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 [10, 44, 8, 6, 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 [23, 50] do not cleanly capture. In text-only LLMs, all data takes essentially the same path through a simple autoregressive (AR) loop. Meanwhile, in modern multimodal LLMs, different modalities and tasks may trigger different execution paths through the same heterogeneous model.

Preprint.

For example, in UMMs such as BAGEL [10], image generation vs. image understanding tasks pass data through different components within the same heterogeneous model. Other models may contain long non-AR loops, such as diffusion transformers (DiTs) [29] or rectified flow [26, 12] 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) [18] or the pipelined Thinker–Talker architecture in Qwen3-Omni [44]. These patterns are not isolated exceptions layered onto otherwise token-centric models but are becoming the standard structure of multimodal models.

Despite rapid progress, modern LLM serving systems [23, 50] still face a fundamental abstraction mismatch when extended from AR-focused text generation to composite multimodal inference. These frameworks have been successfully adapted to multimodal models that attach a non-text encoder to a language model backbone, as in vision-language models (VLMs) or speech recognition models [5, 32]. However, they are insufficient in capturing the complex patterns described above.

Recent work has attempted to address this gap with an intermediate abstraction of a fixed chain [46] or DAG [33] of "stages", where each stage captures one or many model components. However, a gap still remains for modern multimodal models, resulting in suboptimal performance. For example, the stage abstraction cannot capture more complex patterns such as non-AR loops across stages, parallelism internal to a stage, and different tasks or modalities taking different paths through the same model (see §2). Thus, the underlying system misses key performance opportunities, such as parallelism across components or request-specific execution of components. Furthermore, physical placement can only be controlled at the granularity of a stage, which misses efficiency opportunities such as 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 same or better performance as 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 [23], CUDA-graph capture, and continuous batching [47], 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*. We demonstrate these capabilities by instantiating M* on five representative composite models: BAGEL, Qwen3-Omni, $\pi_{0.5}$, V-JEPA 2, and Orpheus. On BAGEL [10], M* delivers, on average, ~30% lower p50 end-to-end latency than vLLM-Omni [46] on text-to-image workloads and ~50% lower for image-editing workloads (with CFG parallelism enabled). It achieves comparable real-time factor (RTF) and throughput compared to VoxServe [21], a speech-optimized serving system, on Orpheus-TTS [8] Compared to vLLM-Omni, M* consistently delivers lower RTF and higher throughput on Qwen3-Omni text-to-speech workloads across batch sizes, reaching up to ~15% higher throughput for batch sizes ≥ 16.

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. We describe five representative families to exemplify the structural patterns that we target.

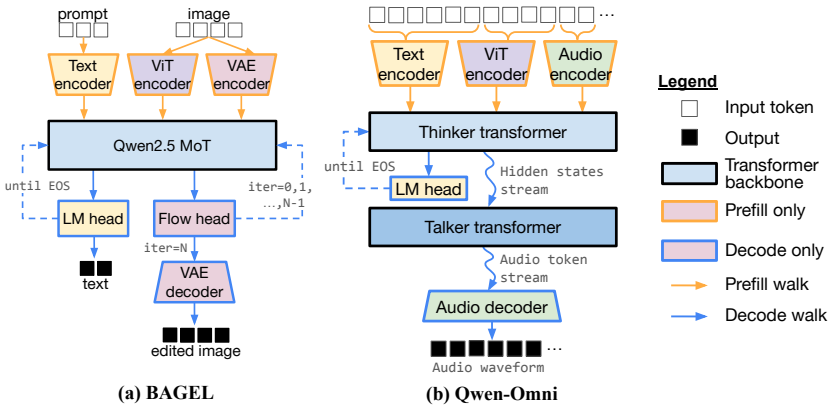


Figure 1: Example model architectures of (a) a UMM (BAGEL [10]) and (b) Omni model (Qwen3-Omni [44]). Composite models are structurally diverse, yet representable as computation graphs.

1. **Unified multimodal models (UMMs)** are designed to handle multiple types of vision tasks with a shared Transformer backbone and encoders/decoders for text and visual latents [10, 9, 42]. For example, BAGEL [10] handles vision understanding, image generation, and image editing in a single model with a shared Mixture-of-Transformers (MoT [25]) component. Each task uses 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 to serve text-to-speech or speech-to-speech applications [8, 11, 19]. The codec model needs to be invoked at different intervals to generate audio waveforms depending on the architecture. For streaming applications, output audio must be produced in real time [21].
3. **Omni models** aim to support any-to-any modalities, including real-time speech. Qwen2.5-Omni and Qwen3-Omni models [43, 44] are notable examples that combine two Transformers in a Thinker–Talker topology. The Thinker produces text and high-level hidden states from inputs in any modality, while the Talker converts those states into speech codec tokens, followed by an audio codec decoder to generate audio waveforms (Figure 1b).
4. **Vision-language-action models (VLAs)** are used in robotics applications to produce a robot action state from an image observation and a text instruction. VLAs often use various combinations of a text encoder, a ViT encoder, a Transformer backbone, and an action decoder [6, 39].
5. **World models** pair a video encoder with an iterative latent predictor that rolls forward over a variable horizon to understand the world. They are often used in applications such as robotic planning [4, 16, 1, 2].

2.2 Composite Models are Computation Graphs

Composite models pose three concrete challenges:

(C1) **Architectural diversity.** As described above, multimodal models exhibit diverse architectures with multiple distinct execution paths depending on input modality. For example, in UMMs, text-to-text chat, text-to-image generation, image-to-text understanding, and image-to-image editing all use different subsets of components with distinct computation patterns. Many computation patterns also involve non-AR loops (e.g.,classes 1, 4, and 5 in Section 2.1).

(C2) **Performant modularity.** Frameworks such as HuggingFace Transformers [40] offer broad flexibility, but often sacrifice efficiency. In contrast, specialized systems such as vLLM [23] for AR text generation and VoxServe [21] for speech generation achieve higher performance in their target domains by implementing domain-specific optimizations, but they do not generalize across modalities.

(C3) **Physical topology.** Composite models use heterogeneous components that may be executed in sequence, in a pipeline, or in parallel. Input tensors may be batched or streamed between components across diverse links, such as intra-node NVLink or inter-node Infiniband. Thus, system flexibility in the physical placement and data transport between components is critical for end-to-end performance.

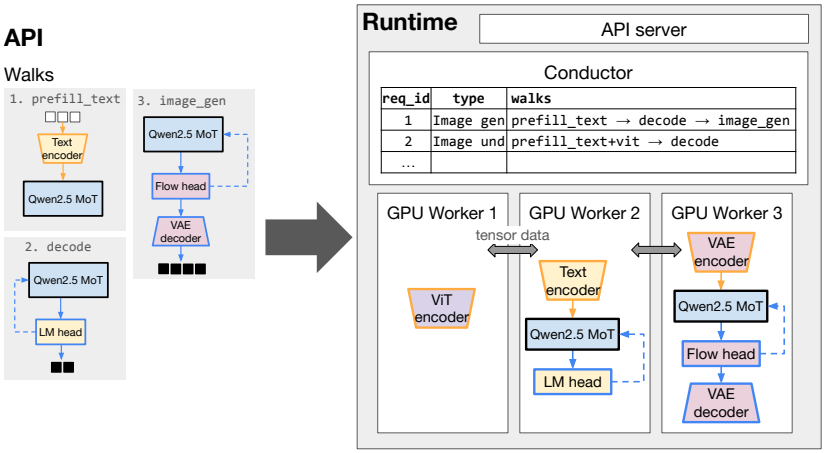


Figure 2: M* at a glance. **Left**: The model author defines the model as a computation graph (Figure 1) and a series of graph walks. **Right**: The runtime (Section 3.3) places the subgraphs of components on different GPU workers based on a user-specified placement.

Why Existing Systems Fall Short. Existing serving systems address only a subset of these challenges. vLLM [23] and SGLang [50] are highly optimized for AR text generation but treat multimodal inputs as prefill-time encoder add-ons, with no first-class support for patterns such as non-AR loops, parallel execution of nodes within a graph walk, or cross-component data streaming. vLLM-Omni [46] and SGLang-Omni [33] extend vLLM and SGLang, respectively, 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. Therefore, per-model glue code is needed for other patterns such as diffusion loops, fan-out in CFG, and custom policies for streaming data across components.

Computation Graphs in M*. Despite the architectural diversity above, we observe that every model in Section 2.1 has the same structural shape: 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. Each request traverses the graph over a small number of *walks*.

Thus, M* addresses C1 by treating multimodal inference as a graph execution, allowing individual components to be composed in arbitrary loops, chains, and parallel branches (Section 3.1). We avoid the abstraction tax by "compiling" the graph abstraction down to a high-performance serving runtime. The runtime includes an efficient request scheduler and per-component engines that integrate state-of-the-art optimizations across modalities (Section 3.3), thus addressing C2. Finally, M* addresses C3 by enabling user-defined physical placements (Section 3.2) and policies to move data from one component to another (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 1 previews how the Walk Graph relates to existing multimodal serving abstractions. Each prior system corresponds to a restricted subset of the Walk Graph.

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 label for the Walk that prefills text tokens in the input prompt in Figure 2. Each request is a series of Walks, e.g.,

Table 1: Comparing the Walk Graph against existing multimodal serving abstractions [46, 33].

	vLLM-Omni [46]	**SGLang-Omni [33]**	**M*** (ours)
Graph node	Engine-instance stage	Worker-pool stage	**Model component**
Composition primitives	Flat DAG	Flat DAG	**Seq. / Par. / Loop / Stream**
Execution paths per model	Prefill, decode	Prefill, decode	Flexible
Loops	Within a stage	Within a stage	Across any subgraph
Placement granularity	Stage	Stage	Component, w/ optional Walk

image understanding is a series of [`prefill_text` $\rightarrow$ `prefill_vit` $\rightarrow$ `decode`]. The model author provides a per-model state machine that determines each request's next Walk based on the request's modalities and the outputs of the current Walk. The model author only provides (G, W) with the state machine, and the execution of 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`, representing a unit of computation that fires when its required inputs arrive, and `GraphEdge`, representing a tensor flowing from one node to another. Nodes are composed into a computation (sub)graph using the four primitives (full description in 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" of tensors before firing. We denote this with `StreamingGraphEdge`, parameterized by a `ChunkPolicy` that decides when the consumer has accumulated enough input to fire. We observe that 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. The policy interface is extensible: new policies can be added without changes to the system.

Example: BAGEL. We make the abstraction concrete with BAGEL [10] (Figure 1a), a unified multimodal model that handles image understanding, image generation, and image editing with one Mixture-of-Transformers [25] backbone. Appendix A shows examples of other models.

Listing 1: Simplified non-CFG Walk for image generation in BAGEL.

```
image_gen = Sequential([
    Loop(section=GraphNode(name="LLM", input_ids={"latents", "time_index"},
                           outputs=[GraphEdge(next_node="LLM", name="latents"),
                           GraphEdge(next_node="LLM", name="time_index")]),
         n_iters=49, outputs=[GraphEdge(next_node="vae_decoder", name="latents")]),
    GraphNode(name="vae_decoder", input_ids={"latents"},
              outputs=[GraphEdge(next_node=EMIT_TO_CLIENT, name="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 is a `Sequential` combination that chains a 49-iteration `Loop` into a terminal `vae_decoder`. Inside the `Loop` is a `Sequential` that runs (i) a `Parallel` region containing the three CFG branches (`LLM`, `LLM_cfg_text`, `LLM_cfg_img`) and (ii) `combine_cfg` that applies the CFG formula and an Euler step and then loops the updated latents back to all three branches. After the final iteration, the resulting latents flow into the `vae_decoder`, which emits the decoded image to the client. For simplicity, the listing above shows a non-CFG version of this Walk, in which a single `LLM` node is iterated instead of three. The full CFG Walk is shown in Section F of the Appendix.

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 allows the runtime scheduler to execute requests efficiently and in a model-agnostic manner. The scheduler only needs to track each request's type, and its currently executing `GraphNode` and Walk (Figure 2). Once one Walk has finished, it then uses the state machine provided by the model author to select the next Walk. A key benefit is that the scheduler by construction executes the minimum components needed to complete each request, rather than forcing all requests to execute all components of the model.

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 [8]. The system runtime is agnostic to the specific model architecture and supports parallelism uniformly across them. By contrast, vLLM-Omni tightly couples modality-specific features such as CFG to the system runtime by adding glue code to expand a user request into multiple branches.

M* further exposes tensor parallelism (TP) as a graph-node-level abstraction. To shard a node, an author replaces its `Linear`, `MLP`, and `Attention` layers with globally-provided sharded counterparts and declares the node's TP degree in the configuration `yaml`. Everything else, e.g., scheduling, synchronization, tensor transport between nodes of differing TP world sizes, and KV-cache transfer, is handled by the system runtime.

Flexible placement. The model deployer may specify a placement mapping from `GraphNode` to GPU rank(s). This decoupled API allows for flexible placements without needing to modify the model or runtime code. Optionally, the user can specify different GPU ranks for the same logical graph node across different Walks. This API can be used to express common patterns important for maximizing hardware utilization, such as data-parallelism by specifying multiple ranks or prefill-decode disaggregation [51] by specifying different ranks for prefill vs. decode Walks. In addition, it enables independent scaling of different model components, e.g., placing encoders and decoders on many small GPUs and LLM backbones on few large GPUs.

Furthermore, this approach allows for transparent resource sharing across concurrent requests, even ones of different types or Walks. For example, in BAGEL the `LLM` node appears in all Walks (Section 3.1 and Figure 2): if the user specifies the same GPU rank(s) for the node in all Walks, then the system automatically multiplexes the same physical `LLM` replica(s) across requests from all Walks.

Loop optimizations. `Loop` and `DynamicLoop` provide first-class loop support and can be used to express iterative patterns including AR decoding until EOS, fixed-step diffusion, and per-request rollout horizon in world models. These semantics are important for enabling portability of system optimizations across different model architectures. In contrast, stage-DAG abstractions in vLLM-Omni and SGLang-Omni cannot have cycles; therefore, any loops must remain internal to a stage.

The `Loop` abstraction allows performance features such as CUDA graphs and continuous batching [47] to be agnostic to the presence of loops. Furthermore, it enables scheduling of loop iterations as if they were any other component. For instance, in the BAGEL model, the M* scheduler can seamlessly interleave flow steps and autoregressive decoding steps that use the same `LLM` node.

Flexible chunk policies. `StreamingGraphEdge(policy)` allows capture of arbitrary producer-consumer patterns within a model architecture. Three reusable `ChunkPolicy` types (§3.1) cover every streaming connection in our evaluation. For example, Qwen3-Omni uses a `FixedChunkPolicy` with chunk size 1 for the Thinker→Talker connection; this means that the Talker should consume each Thinker output as soon as it arrives. The same model uses `LeftContextChunkPolicy` for Talker→Code2Wav for causal smoothing of the audio output. Critically, the runtime is agnostic to the chunk policy used; the same infrastructure is used for a range of streaming patterns.

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 [17]; 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 [30].

Each `GraphNode` is executed by an *engine*, an inference instance selected by the model author based on the node's component type. There are currently two engines: `KVCacheEngine`, a modality-agnostic transformer engine with `FlashInfer`-based paged-attention KV-cache state and a cuda-graph-compatible sampling plugin. Stateless nodes use simpler execution paths via `StatelessEngine`. Both engines support continuous batching, CUDA-graph replay, and `torch.compile`. Each worker can host multiple engines and runs a local scheduler to drive the engines with a round-robin execution policy.

To overlap CPU scheduling with GPU execution, each worker asynchronously schedules batch $N + 1$ while batch N is still in flight. The next batch is scheduled by alternating between the following: (1) traversing the Walk Graph with the outputs of the current batch N, and determining what nodes will be ready once the current batch finishes, and (2) scheduling any unrelated batches that are ready (to avoid head-of-line blocking). The heaviest overhead is often constructing the FlashInfer attention plan; we double-buffer the attention plan and asynchronously construct the next attention plan in a separate thread and CUDA stream. For speculation across `DynamicLoop` iterations, termination checks are deferred to the next iteration, so each termination costs at most one wasted step.[1]

4 Evaluation

We evaluate M* on BAGEL-7B [10], Qwen3-Omni-30B-A3B [44], Orpheus-3B [8], and V-JEPA 2 vitg-AC [4]. As baselines, we use vLLM-Omni for supported models (BAGEL and Qwen3-Omni), SGLang-Omni for Qwen3-Omni, and modality-specific baselines for the remainder: VoxServe [21] for Orpheus and Meta's native `vjepa2` implementation for V-JEPA 2. Since $\pi_{0.5}$ and V-JEPA 2-AC are both robotic planning models, are unsupported by HuggingFace Transformers and serving frameworks such as vLLM-Omni, and only provide native repositories, we benchmark only V-JEPA 2-AC. All experiments were run on either a single 4×H100 node or a single 8×H200 node; configurations are reported inline.

Metrics. When measuring 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. Lower is better and <1 means streaming is feasible. Each configuration uses 5-10 warmup requests followed by 10–160 timed requests (at least $5\times$ the maximum concurrency). We report p50 (solid bars) and p95 (hatched extensions) where appropriate.

4.1 BAGEL: M* vs vLLM-Omni

We evaluate BAGEL on text-to-image (T2I), image editing (I2I), and image-to-text understanding (I2T) using inputs from VBench [20] for generation tasks and Food101 [7] for understanding. Both systems use the BAGEL-7B checkpoint, generating images ar 1024×1024 resolution with a 50-step flow schedule. For I2I, both M* and vLLM-Omni generate images of the same aspect ratio as the input image (scaled such that the long edge is dimension 1024).

For T2I/I2I, we benchmark two configurations of vLLM-Omni: the default configuration, which has a "Thinker" and diffusion transformer (DiT) stage (essentially replicating the BAGEL transformer), and a single-stage configuration. For I2T, the default configuration outperforms the single-stage pipeline, so we benchmark the default configuration, with the maximum number of sequences set to largest batch size tested.

[1] For models where this wasted step is inadmissible, speculative scheduling can be disabled on a per-node level.

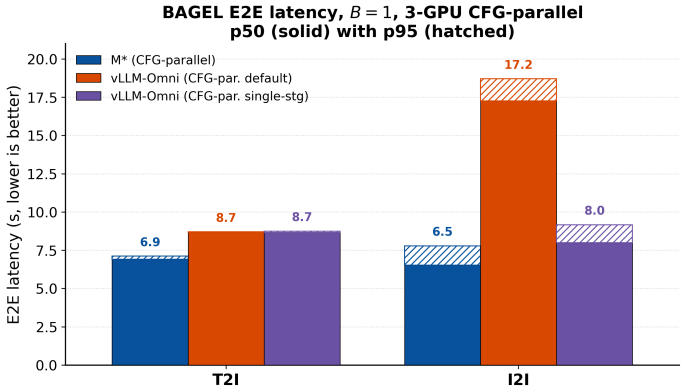


Figure 3: **BAGEL T2I/I2I E2E latency, B=1, 3-GPU CFG-parallel.**

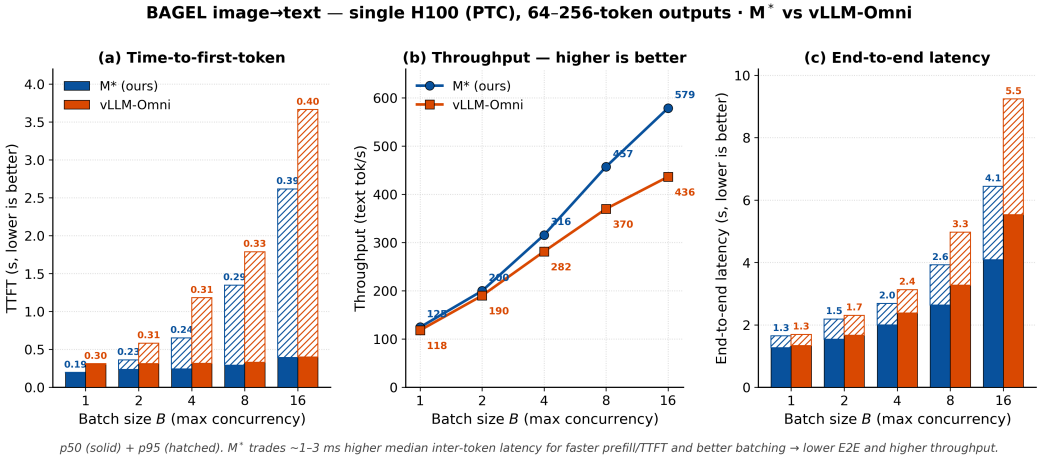


Figure 4: **BAGEL I2T, single H100,** $B \in \{1, 2, 4, \ldots, 16\}$**.**, `ignore_eos` with output lengths between 64 and 256. (a) TTFT (log y). (b) Throughput (req/s). (c) End-to-end request latency.

For T2I/I2I we use 3 H100s with classifier-free-guidance (CFG) parallelism (one rank per CFG branch). M* runs the three branches in parallel via the `Parallel` primitive (Section 3.1); vLLM-Omni uses a specialized CFG parallel plugin that ues `torch.distributed`.

For I2T we use 1 H100, since CFG only applies to image generation. To ensure exact parity in output token count, we ignore EOS in both vLLM-Omni and M*, instead generating until a benchmark-determined sequence length. All I2T results are averaged across three benchmark runs.

Image generation (T2I, I2I). On 3-GPU CFG-parallel at B=1, M* improves on single-stage vLLM-Omni's p50 end-to-end latency by $1.25\times$ on T2I and $1.22\times$ on I2I (Fig. 3). For the default configuration, which involves an expensive KV cache transfer between the Thinker and DiT, our advantage on I2I grows to $2.64\times$. The p95 advantage is similar. We also see performance gains in the single-GPU (i.e., no CFG parallelism) case, plots for which can be found in Section E.

This improvement is primarily due to M*'s KV cache management abstractions: we represent the three CFG contexts as three labels over a single paged KV pool, as opposed to vLLM-Omni's dense `NaiveCache` per CFG context. Each denoise step can then apply paged attention, reading page tables in place; vLLM-Omni concatenated key and value tensors at every layer and for every step. The label is a general cache-key axis, as such, it inherits paging, offload, by-reference transfer, and continuous batching, whereas the dense `NaiveCache` is model-specific without immediate access to such optimizations.

Image understanding (I2T). For output token lengths distributed uniformly between 64 and 256, we achieve comparable throughput to vLLM-Omni at low batch sizes, with our advantage increasing to 32.7% by B=16 (Figure 4). This is also reflected in the per-request E2E latency (25.5% improvement at B=16). M* achieves consistently lower p50 TTFT across batch sizes, ranging from 33% at

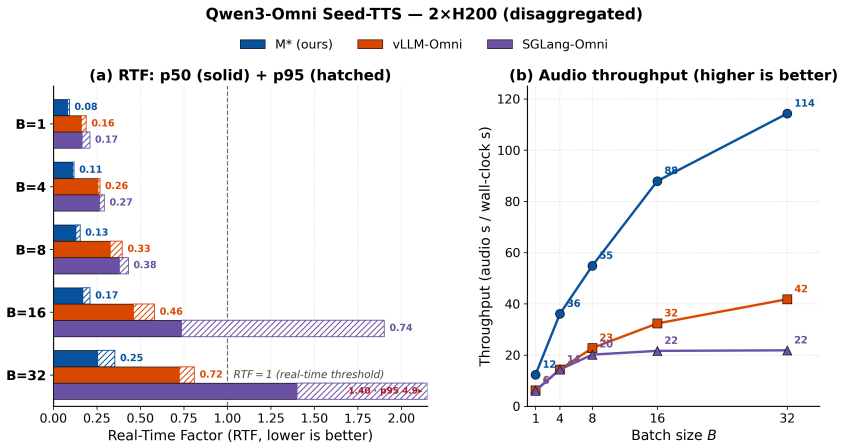


Figure 5: **Qwen3-Omni Seed-TTS, 2-GPU.** (a) RTF (lower is better). (b) Audio throughput (higher is better).

B=1 to 14% at B=16, while maintaining a tighter tail: M* p95 TTFT is 28% lower than that of vLLM-Omni at B=16, despite lower p50 gains. Figures 10 and 11 in the Appendix (varying the output token length distribution) show a similar story , with our advantage most prominent for shorter-decode workloads.

M* enables one config for all modalities. In this section, we benchmark two configurations of vLLM-Omni, and neither performs well across all three workloads at once: the default configuration performs well for T2I and I2T but poorly for I2I, whereas the single-stage configuration performs well for T2I and I2I but poorly for I2T.[2] The default config suffers on I2I because the Thinker and DiT are separate stages, requiring an expensive KV-cache transfer between prefill and the flow loop; the single-stage configuration collapses them into one process and foregoes that transfer, but no longer runs I2T on vLLM's optimized AR engine—losing continuous batching and token streaming. M*, by contrast, serves optimized T2I, I2I, *and* I2T with the same configuration, while also enabling PD disaggregation, encoder disaggregation, and tensor parallelism with minor (mainly config-level) changes.

4.2 Qwen3-Omni: M* vs vLLM-Omni and SGLang-Omni

We run Qwen3-Omni on the Seed-TTS [3] text-to-speech benchmark on 2 H200s, with the model disaggregated as Thinker on rank 0 and Talker+Code2Wav on rank 1 (Figure 5). M* significantly outperforms vLLM-Omni and SGLang-Omni's RTF across batch sizes. At B=16, M* delivers 2.7× and 4.0× higher throughput than vLLM-Omni and SGLang-Omni, respectively. We also apply degree-2 tensor parallelism to the Thinker (overall using 3 H200s), and compare against SGLang-Omni.[3] As shown in Figure 6, our RTF and throughput advantage remain consistent to the non-TP results in Figure 5.

The RTF and throughput improvement is largely due to the M*'s flexibility and modularity. As CUDA graph capture in M* is defined on a per-submodule level with customizable inputs and outputs, the entire Talker submodule, including the multi-token predictor loop, is able to run as a single CUDA graph. vLLM-Omni, on the other hand, has CUDA graphs explicitly disabled for the Code Predictor. In addition, our system places a co-located Talker and Code2Wav on the same Worker process (via our one-to-one Worker-to-GPU mapping), eliminating the need for inter-process communication of Talker codes to the Code2Wav. Both vLLM-Omni and SGLang-Omni require separate processes for each stage. The `StreamBuffer` abstraction allows clean support for both colocated and disaggregated streaming of data between graph nodes.

[2]Specifically, it achieves a throughput of 41 tokens/sec on batch size 1 (with `enforce_eager` manually set to `false` in the config file, and `max_num_seqs` increased to `16`), which is half as fast as the default config. It also appears to not support continuous batching or streaming of tokens to the API server.

[3]We were unable to get vLLM-Omni's tensor-parallel Thinker to work.

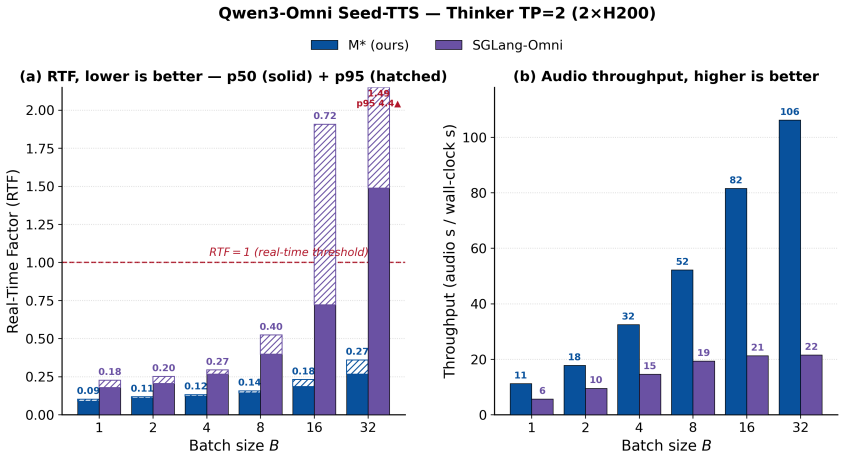


Figure 6: **Qwen3-Omni Seed-TTS, Thinker TP 2 (3-GPU.** (a) RTF (lower is better). (b) Audio throughput (higher is better).

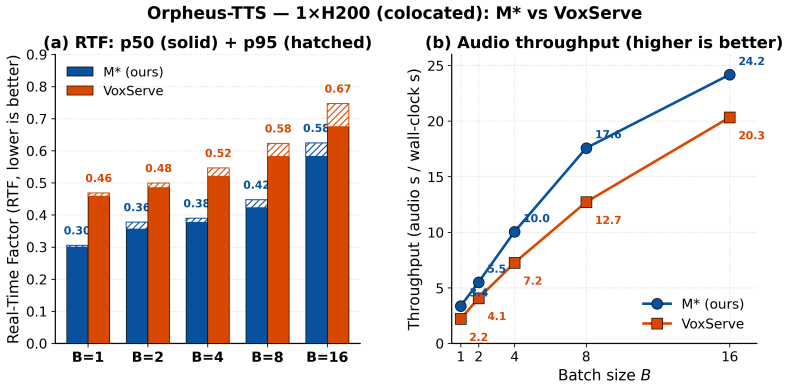


Figure 7: **Orpheus TTS, single H200.** (a) RTF (lower is better). (b) Audio throughput (higher is better).

4.3 Orpheus: M* vs VoxServe

We measure Orpheus-3B performance on a single H200 against VoxServe [21] on B=$\{1, 2, 4, 8, 16\}$ and Seed-TTS (Fig. 7), averaging results over 5 trials. For the M* implementation, we have an LLM with a `StreamingEdge` into a SNAC audio decoder. M* performance is overall better: M* delivers 13.6% lower p50 RTF than VoxServe at B=16, with throughput improvements ranging from 20% (B=8) to 52% (B=1). We attribute these gains to the following components of M*: Authors inherit model-level improvements for free, e.g., fused projections provided by M*'s `Attention` layers and cuda-graph-compatible sampling, and the Walk Graph abstraction enables the speculative `FlashInfer` planning described in §3.3 for arbitrary multimodal requestss.

4.4 V-JEPA 2: Rollout for Robotic Planning

V-JEPA 2 is a world model that supports action-conditioned (AC) rollout: each step autoregressively predicts the next video frame conditioned on an action and previous states. We compare against the Meta's `vjepa2` implementation on 1 H100 with B=1 at rollout horizons H=$\{4, 15, 30\}$, using inputs from the first 50 episodes of the DROID dataset [22]. The baseline runs a hand-written Python autoregressive loop without KV caching, forcing costly prefill over a growing sequence at every iteration.

Meanwhile, M* encodes the rollout as a `DynamicLoop` over the `AREngine`, applying paged-attention KV caching to avoid duplicate prefills. Thus, M* delivers p50 speedups of 2.08× at H=4, 3.76× at H=15, and 12.5× at H=30 (Figure 8).

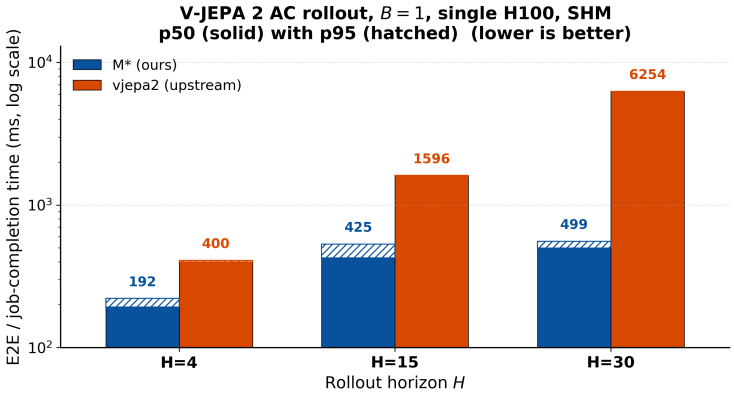


Figure 8: **V-JEPA 2 AC rollout,** $B{=}1$**, single H100** (lower is better).

5 Related Work

Token-centric LLM serving. vLLM [23], SGLang [50], and Orca [47] 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 [51], Splitwise [28], and Mooncake [30] disaggregate prefill from decode to avoid inter-phase interference. M* supports this and other disaggregation options via flexible placement policies (§3.2).

Multimodal serving. vLLM-Omni [46] and SGLang-Omni [33] represent a model as a fixed DAG of stages, where each stage executes as a separate inference engine. ModServe [31] and EPDServe [34] support flexible disaggregation but only for the image preprocessing and encoder components. Cornserve [27] supports any-to-any models by recovering a dependency graph of Python tasks during execution via record-and-replay. M*'s Walk Graph captures computation ahead of time and at a finer granularity – a forward pass of one component such as one denoising iteration in a DiT vs. a full request on a DiT engine in Cornserve – to enable compile-time optimizations across components.

Modality-specific serving. Many works focus on modality-specific inference optimizations, most of which can be integrated into M*. VoxServe [21] unifies SpeechLM serving with a streaming-aware scheduler; M* supports this through streaming graph edges. FastVideo [49, 48] introduces sparse attention techniques for video generation that can be directly integrated into M*. xDiT [14] proposes parallelism strategies for DiTs [15, 13, 36], which future versions of M* can support through flexible placement policies. Inferix [37] targets long-video world models with block-diffusion decoding, combining LLM-style KV-cache management with block-wise diffusion; M*'s Walk Graph can express such hybrid autoregressive–diffusion pipelines as a single graph. FlashDrive [24] accelerates VLA inference for autonomous driving through temporal KV-cache reuse, speculative decoding, and adaptive flow-matching steps, all complementary to M*'s graph-level scheduling.

6 Conclusion

M* is a universal serving runtime built on the observation that composite multimodal models can be captured as walks over a dataflow graph. By introducing the Walk Graph, a small set of unifying and composable graph primitives, M* decouples the model architecture from the physical placement and execution. We show that the resulting system can deliver performance on par with or better than state-of-the-art baselines across a range of model families. As models are increasingly deployed in the real world, such composite multimodal models will become increasingly critical to end applications. Thus, we expect that M* will accelerate the development of the next generation of models.

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] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] Jiarui Fang, Jinzhe Pan, Aoyu Li, Xibo Sun, and WANG Jiannan. 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`.

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

[37] 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.

[38] 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.

[39] 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.

[40] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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 [44] 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 (capabilities 3 and 4).

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 expands the simplified comparison in §3 (Table 1) along nine axes, each tracing back to a primitive or capability introduced in §3.

E Deferred Experimental Results

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

M* improves on vLLM-Omni's T2I p50 latency by $1.13\times$. For I2I, M* also improves on default config vLLM-Omni's I2I p50 latency by $1.66\times$ and $1.25\times$ for vLLM Omni's single-stage config (Figure 9).

Table 3: The Walk Graph compared with existing multimodal serving abstractions. vLLM-Omni and SGLang-Omni implement stage graphs at *engine-instance* granularity; VoxServe specializes a single `Model` interface for SpeechLMs. Each prior system corresponds to a restriction of the Walk Graph (see §3).

	vLLM-Omni	**SGLang-Omni**	**VoxServe**	**M* (ours)**
Graph granularity	Stage = engine instance	Stage = worker pool	Single `Model` class	**Node = one forward pass**
Walks per model	One pipeline (frozen) per `model_type`	One `PipelineConfig` per variant	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	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 scheduler's outer loop	Graph-level `Loop` / `DynamicLoop`
Iter loops ⇒ CUDA graphs	No: `enforce_eager: true` on Code2Wav & BAGEL DiT *(verbatim: "cudagraph-incompatible")*	Per-stage; codec executors are eager Python loops	Audio-only	Yes — per-iter `forward` is shape-static
Streaming policies	Single `async_chunk` toggle	Generic `StreamQueue` transport; chunking is per-model	Per-model `detokenize_interval` (audio only)	Three reusable `ChunkPolicy` types cover taxonomy
Placement granularity	Per-stage `devices: str`	Per-stage `gpu_placement: dict`	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	8 SpeechLM families (TTS + STS only)	BAGEL, Qwen3-Omni, V-JEPA 2 (masked + AC), $\pi_{0.5}$, Orpheus

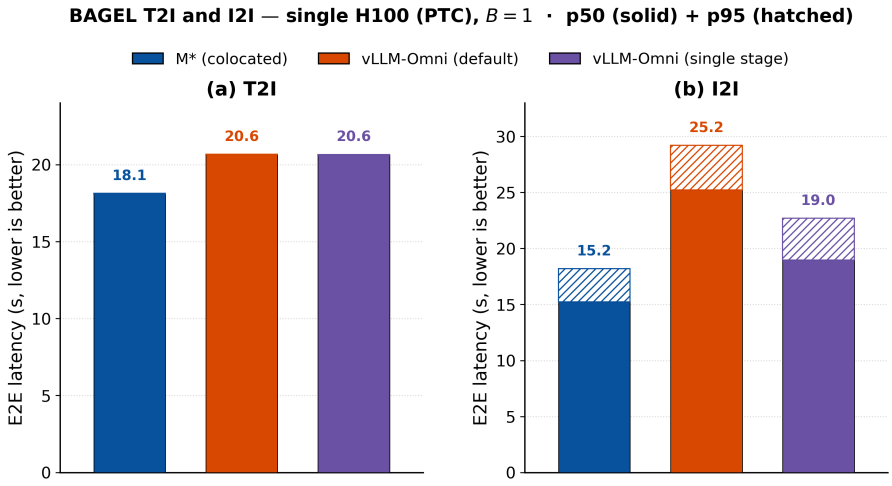


Figure 9: **BAGEL T2I and I2I, single H100,** B=1**.** M* wins on both; the gap widens for I2I.

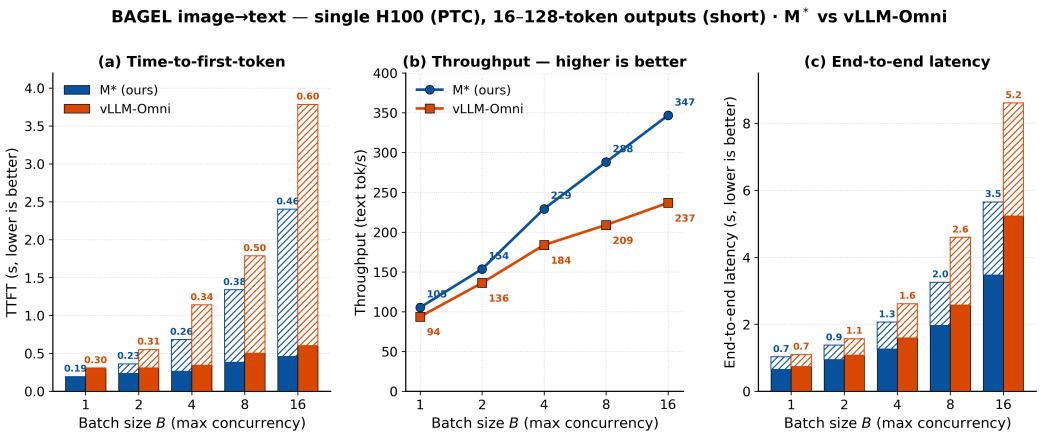


Figure 10: **BAGEL I2T, single H100,** $B \in \{1, 2, 4, \ldots, 16\}$**.**, `ignore_eos` with output lengths between 16 and 128. (a) TTFT (log y). (b) Throughput (req/s). (c) End-to-end request latency.

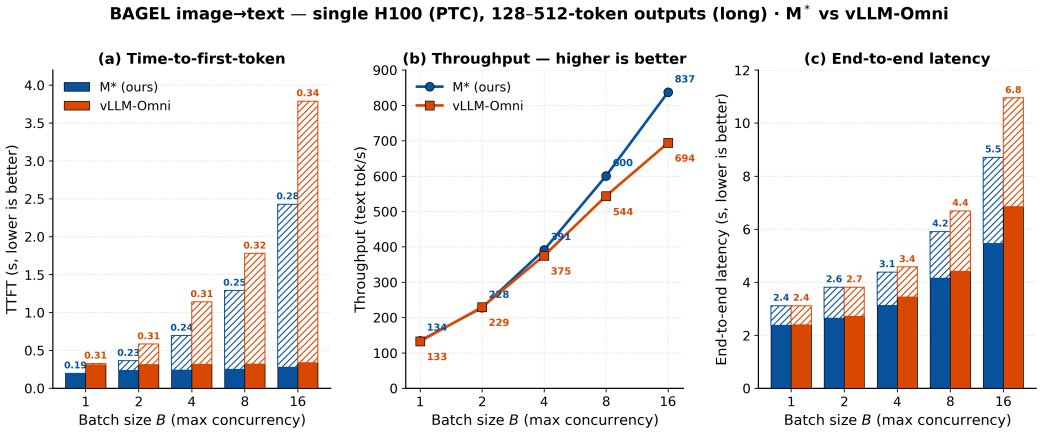


Figure 11: **BAGEL I2T, single H100,** $B \in \{1, 2, 4, \ldots, 16\}$**.**, `ignore_eos` with output lengths between 128 and 512. (a) TTFT (log y). (b) Throughput (req/s). (c) End-to-end request latency.

Figures 10 and 11 show M* vs. vLLM-Omni on BAGEL I2T for varying output length workloads: one where output lengths are randomly sampled between 16 and 64 tokens, and one where output lengths range from 128 to 512 tokens. Overall, we have the largest advantage on shorter-decode and higher-concurrency workloads, while remaining comparable in the longer-sequence, lower-concurrency setting.

F Walk Graph Primitives: Image Generation Walk with CFG Parallelism

```
image_gen_cfg = Sequential([
  Loop(
    section = Sequential([
      Parallel([ # 3-branch CFG, 1 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],
           [loopback to all three LLMs])
    ]),
    max_iters = num_timesteps - 1, # 49 Euler steps
    outputs = [Edge("vae_decoder", "latents")]),
  Node("vae_decoder", [latents], [Edge(EMIT_TO_CLIENT, "image_output")]),
])
```

G 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.

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, as stated in Section 4, many evaluation settings that we ran are not included due to significant correctness and performance issues found in baselines, which may have been due to evaluating these systems in previously untested settings. Thus, we did not include these results for fairness.

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. Configuration files, dockerfiles, and the exact reproduction commands will accompany the public source release at the camera-ready deadline. We commit to working with reviewers to ensure that all results are reproduced before the camera-ready deadline.

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

- One $4\times$NVIDIA H100 (80 GB) node, used for BAGEL on its 3-GPU CFG-parallel deployment (one rank per CFG branch, encoders / VAE colocated with rank 0) and V-JEPA 2 on a single GPU.
- One $8\times$NVIDIA H200 (141 GB) node, used for Qwen3-Omni on a 2-GPU disaggregated deployment (Thinker on rank 1, Talker+Code2Wav on rank 0) and Orpheus on a single GPU.

Software stack. M* requires Python 3.12 and PyTorch with CUDA. The engine layer uses FlashInfer [45] for paged-attention prefill and decode, HuggingFace Transformers [41] 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 so cross-system token sequences match. Each configuration uses 3–5 warmup requests followed by 10–160 timed requests (with the total number of requests being least $5\times$ the maximum concurrency); we report mean, p50, and p95 wherever appropriate.

- **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; workloads T2I, I2I, I2T at $B \in \{1, 4, 8\}$ on VBench prompts.
- **Qwen3-Omni-30B-A3B-Instruct** (`configs/qwen3omni_2gpu.yaml`): Seed-TTS evaluation set, `max_tokens=256`, `thinker/talker/cp_temperature=0`, system prompt matching vLLM-Omni's official Qwen3-Omni examples; $B \in \{1, 4, 8, 16, 32\}$.
- **Orpheus-3B** (`configs/orpheus.yaml`): LLM on rank 0, SNAC decoder on rank 1; 3-engine walk (Embeddings $\rightarrow$ LLM $\rightarrow$ SNAC), $B \in \{1, 2, 4, 8, 16\}$.
- **V-JEPA 2 ViT-g (AC)** (`configs/vjepa2_ac.yaml`): DROID episodes, 8 frames per request at 256×256, bf16, sequential (B=1), rollout horizons $H \in \{4, 15, 30\}$. The upstream Meta vjepa2 baseline reproduces the verbatim driver pattern from `notebooks/energy_landscape_example.ipynb` (Cells 5–6).

Software. Table 4 lists the third-party systems used as baselines or as direct dependencies of M*'s engine layer. Where a license could not be confirmed from the official repository, the cell is left blank.

Table 4: Software / code licenses.

Asset and version	URL	License
vLLM-Omni (vllm: v0.21.0) [46]	`https://github.com/vllm-project/vllm-omni`	Apache 2.0
SGLang-Omni (commit: 4a3960) [33]	`https://github.com/sgl-project/sglang-omni`	Apache 2.0
VoxServe v0.1.0 [21]	`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 [10]	`https://github.com/bytedance-seed/Bagel`	Apache 2.0
Orpheus-TTS reference [8]	`https://github.com/canopyai/Orpheus-TTS`	Apache 2.0
openpi ($\pi_{0.5}$ reference) [6]	`https://github.com/Physical-Intelligence/openpi`	Apache 2.0
SNAC (audio codec) [35]	`https://github.com/hubertsiuzdak/snac`	MIT
HuggingFace Transformers [41]	`https://github.com/huggingface/transformers`	Apache 2.0
HuggingFace Diffusers [38]	`https://github.com/huggingface/diffusers`	Apache 2.0
FlashInfer [45]	`https://github.com/flashinfer-ai/flashinfer`	Apache 2.0
Mooncake transfer engine [30]	`https://github.com/kvcache-ai/Mooncake`	Apache 2.0

Model weights. Table 5 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).

Table 5: Model checkpoint licenses.

Asset	URL	License
BAGEL [10]	`https://huggingface.co/ByteDance-Seed/BAGEL-7B-MoT`	Apache 2.0
Qwen3-Omni [44]	`https://huggingface.co/Qwen/Qwen3-Omni-30B-A3B-Instruct`	Apache 2.0
Orpheus [8]	`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

Datasets and benchmarks. Table 6 lists the evaluation datasets. Where a license could not be confirmed from the official source, the cell is left blank.

Table 6: Dataset / benchmark licenses.

Asset	URL	License
VBench [20]	`https://github.com/Vchitect/VBench`	Apache 2.0
Seed-TTS [3]	`https://github.com/BytedanceSpeech/seed-tts-eval`	CC BY 4.0
DROID [22]	`https://huggingface.co/datasets/lerobot/droid_100`	MIT