



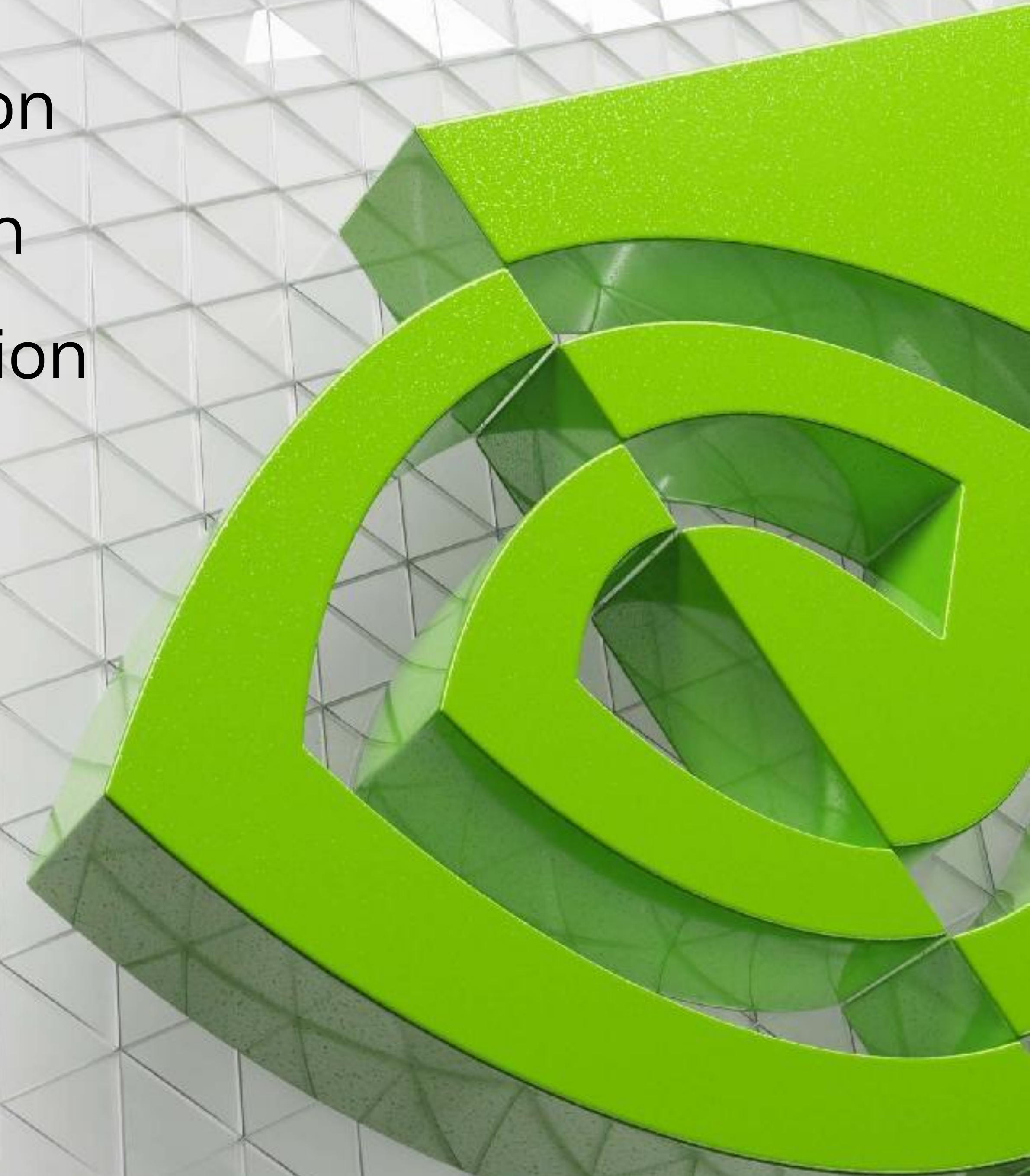
From Pixels to Physical Motion: Building World Action Models (WAM) with PyTorch & NVIDIA Cosmos



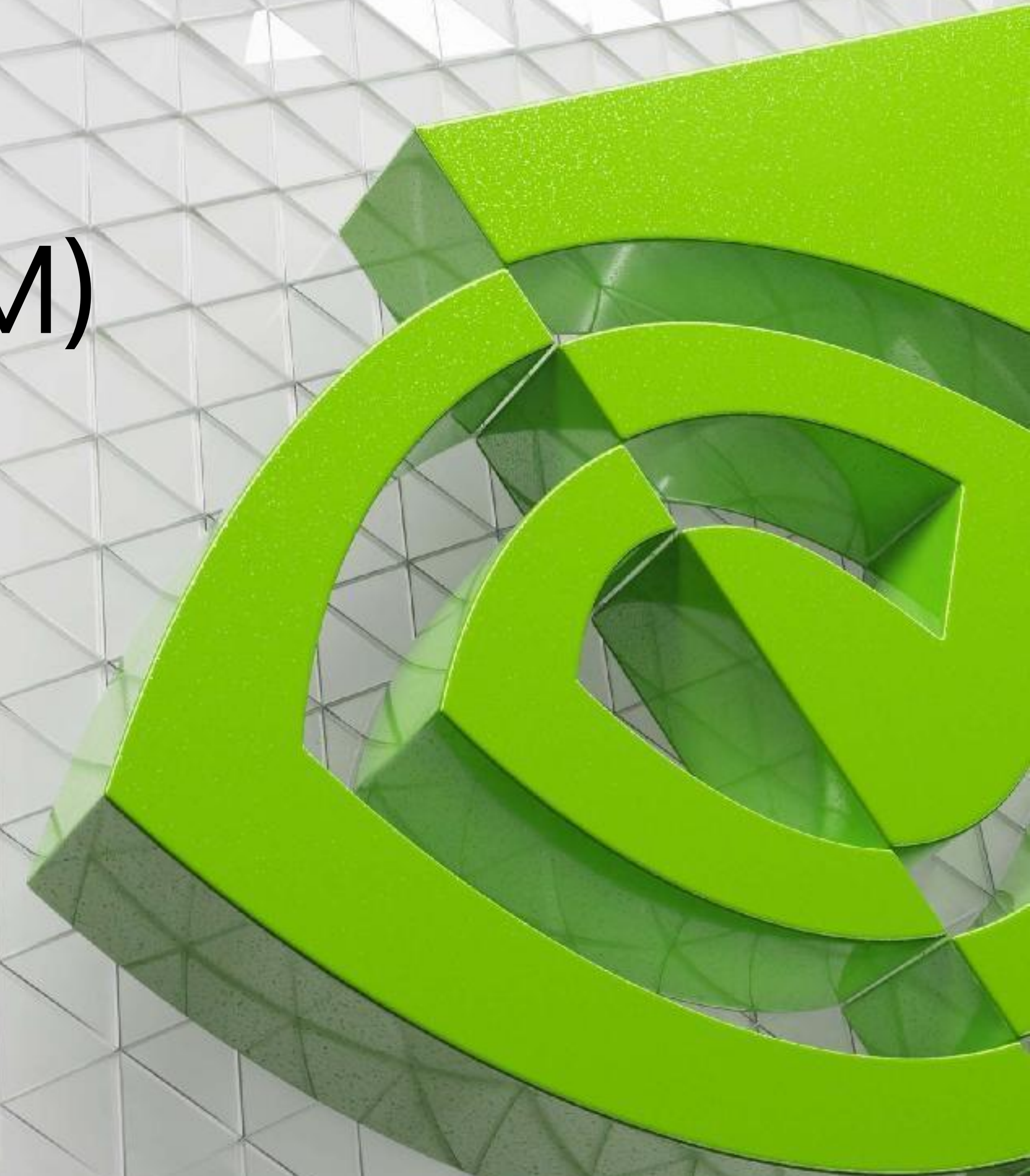
Susie Xia, Ruijie Zheng, George Kurian
PyTorch Conference | Oct 21, 2026

Agenda

- WAM: From World Models to Robot Action
- Scaling Multimodal Training with PyTorch
- From Fast Prediction to Continuous Motion
- Takeaways



World Action Model (WAM)



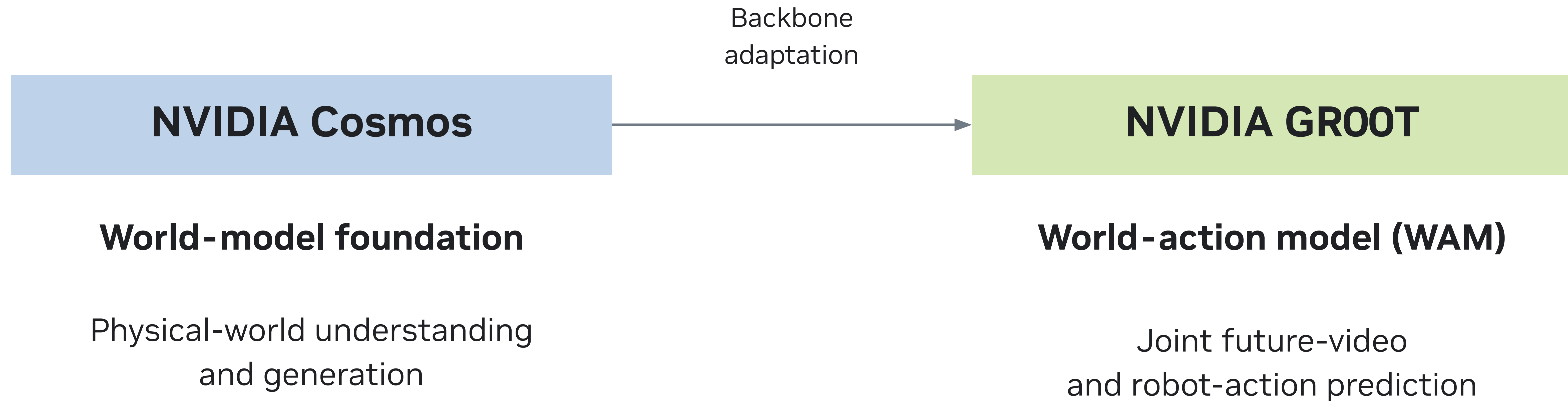


What Is a World-Action Model (WAM)?

One model jointly predicts future video and robot actions

- **Vision-first:** built on a video-generation backbone, not a language model.
- **Learning to act by learning to imagine:** jointly predicts the near future and the robot actions that realize it.
- **Strong generalization:** predicting the future encourages the model to learn physics and dynamics.
- **Interpretable:** watch the imagined future to understand and debug behavior.

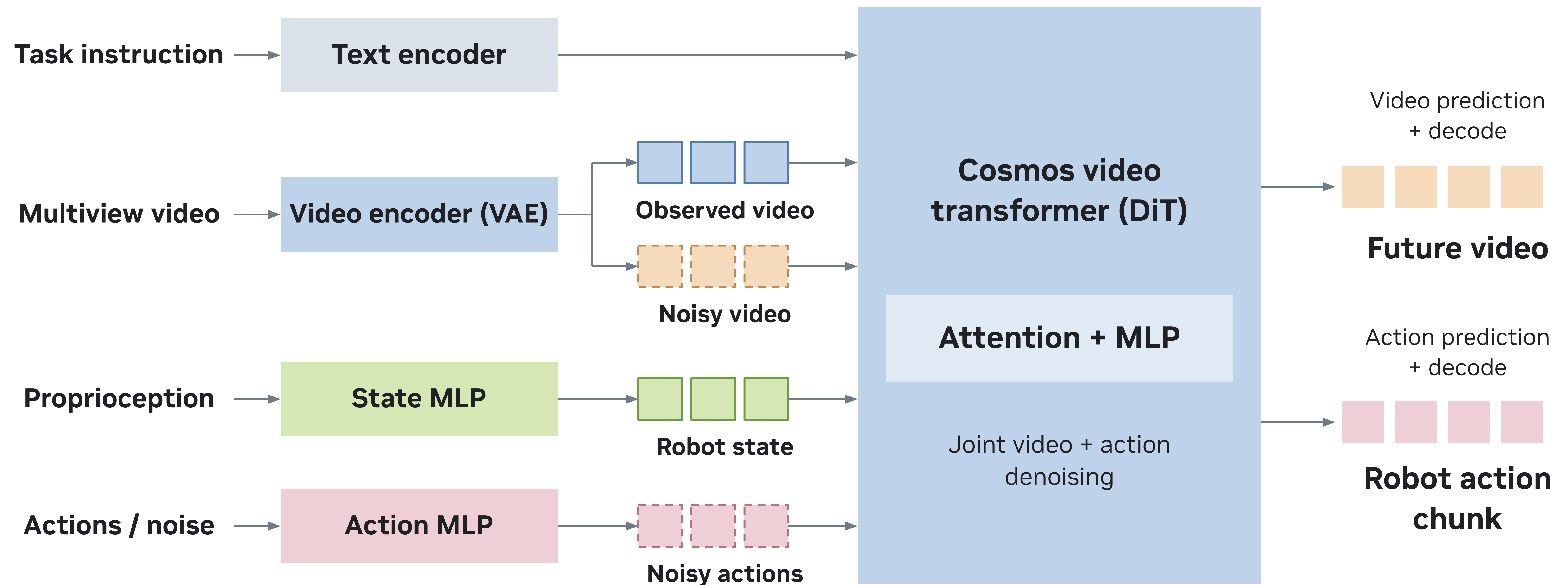
From World Models to Robot Action



Our examples span Cosmos and GR00T for training and robot deployment.

GROOT WAM: Joint Video and Action Prediction

A Cosmos world-model transformer backbone extended toward robot action

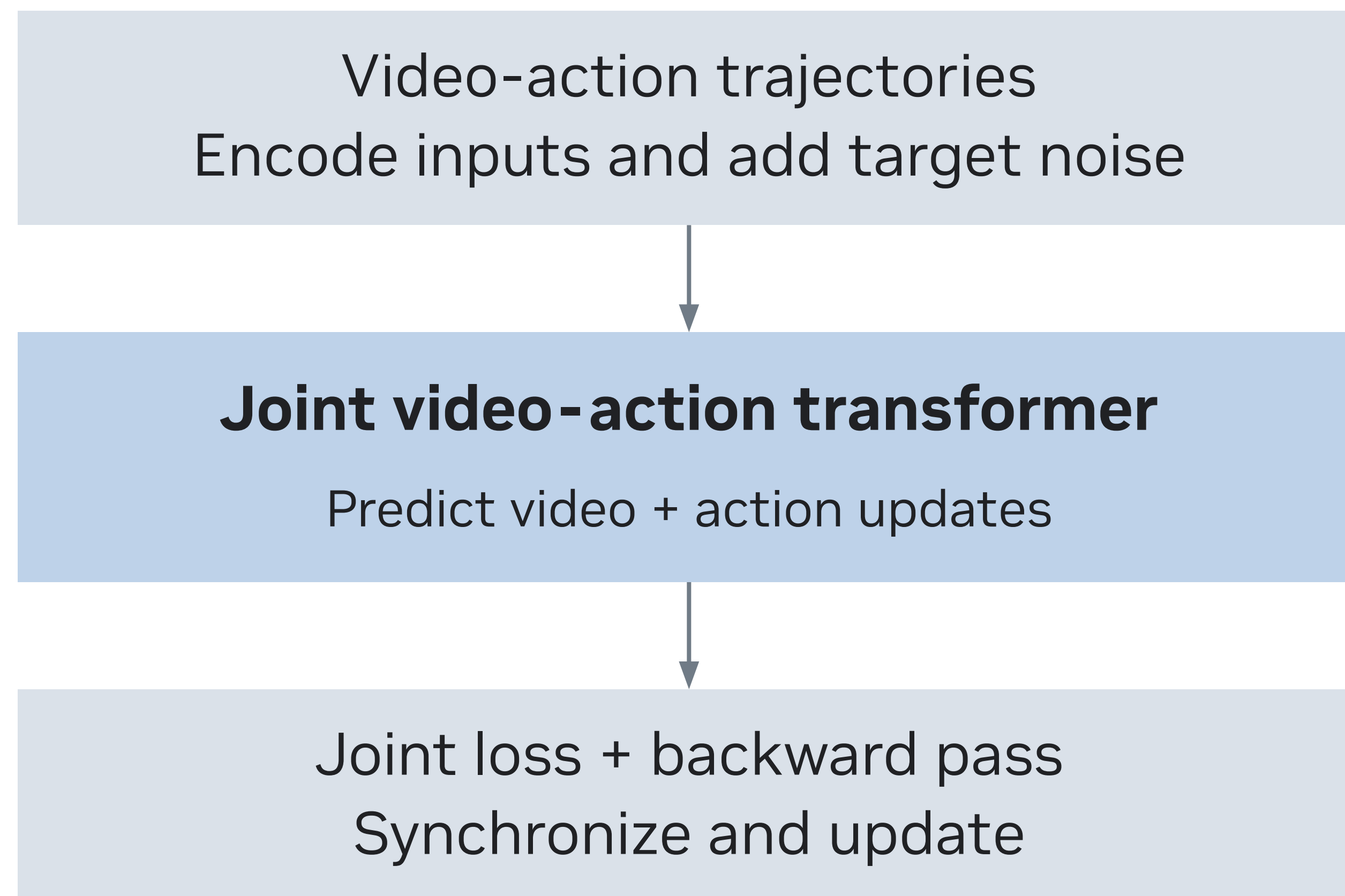


Input tokens: solid = observed conditioning dashed = noisy denoising targets

Systems Challenges

Shared lessons from Cosmos and GROOT for efficient training and inference

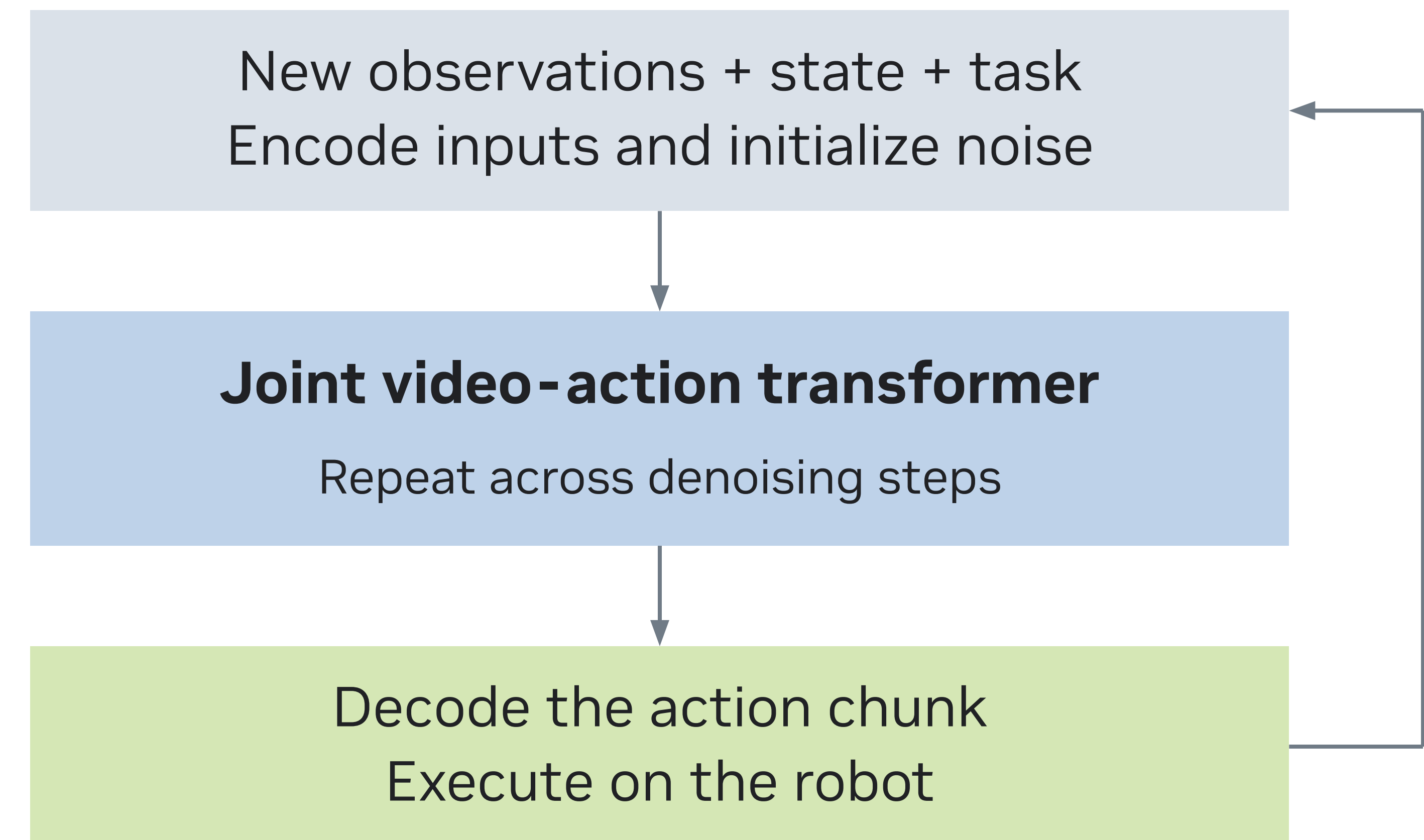
Training



**Scale efficiently across
1,000+ datacenter GPUs**

Improve MFU and reduce
distributed stalls.

Inference



Fresh observations for the next prediction

**React at 10 Hz end to end
on edge hardware**

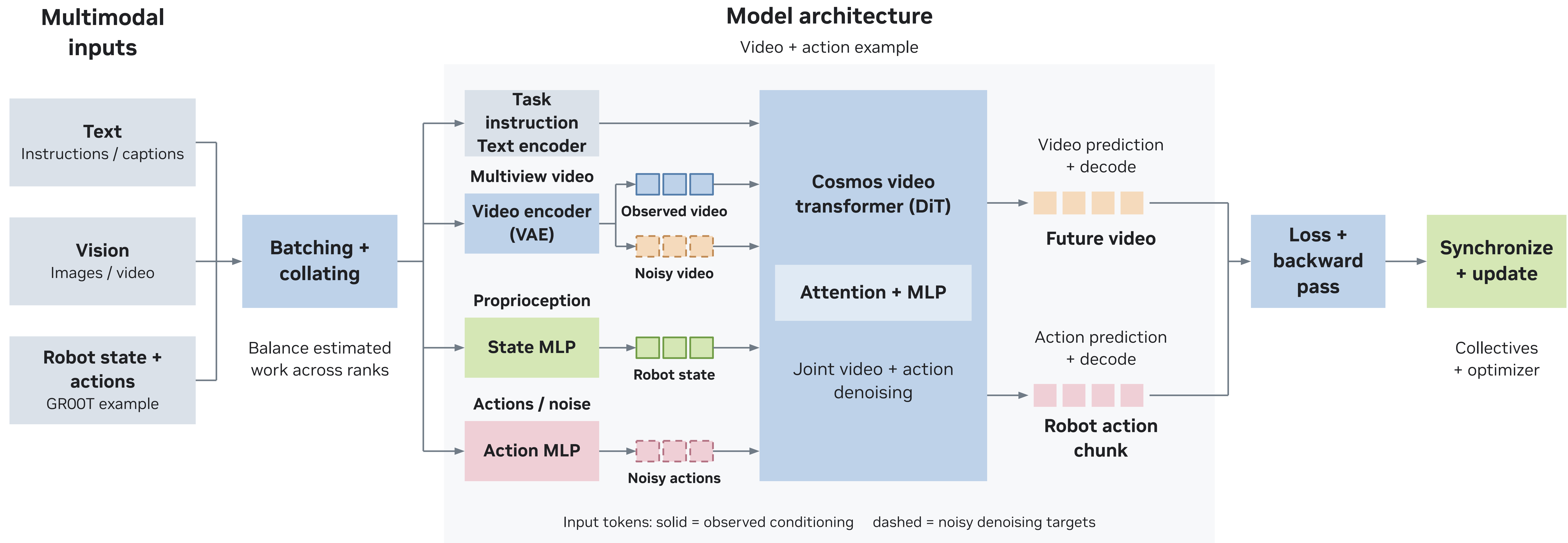
Optimize latency within
edge compute and memory limits.

Scaling Multimodal Training with PyTorch



Multimodal Training with PyTorch

Training system for heterogeneous multimodal inputs: from batching to synchronized updates



Three Ways to Balance Multimodal Batches

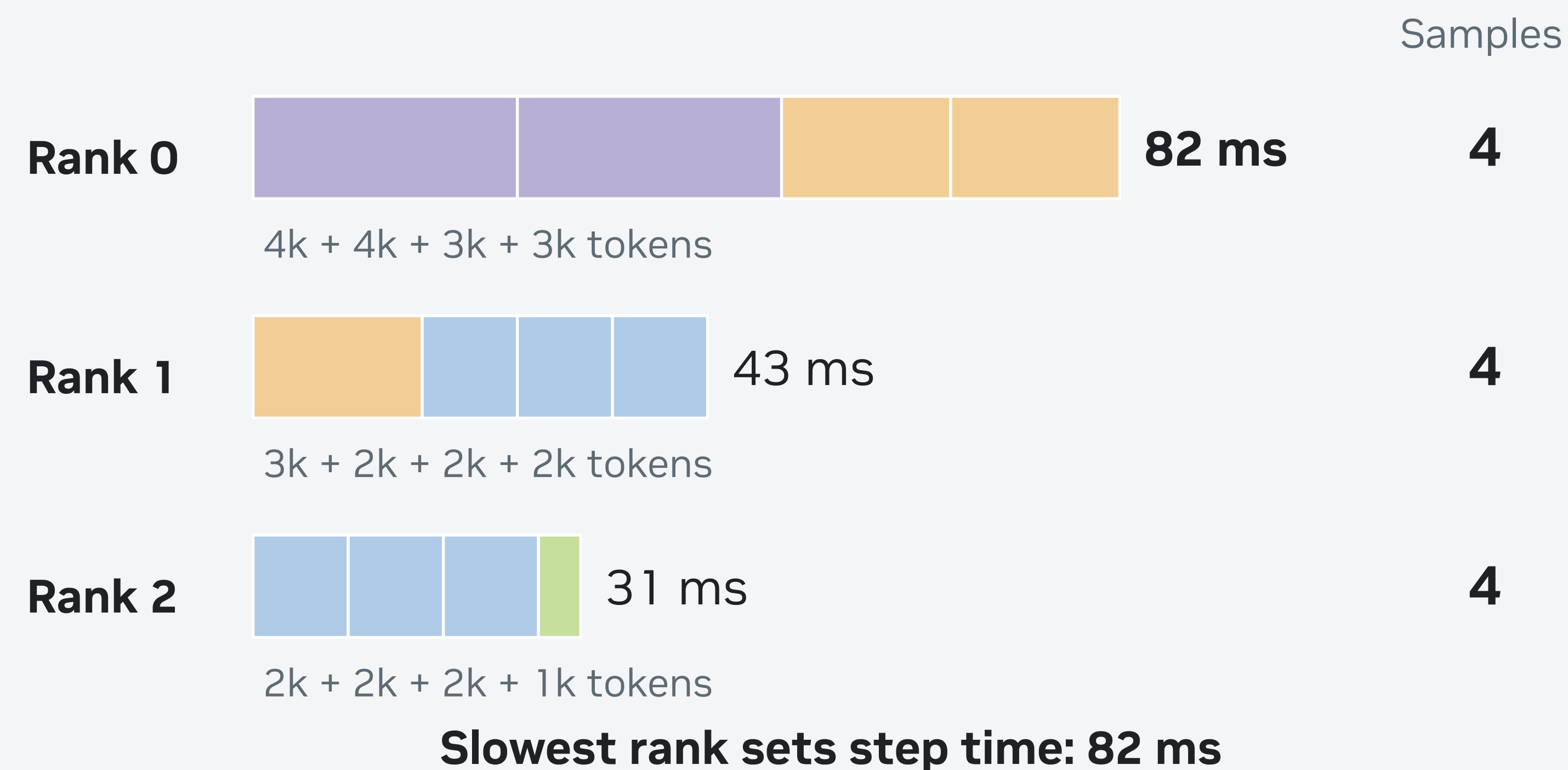
Equal sample counts and equal token counts can still mean unequal compute

- Equal sample counts ignore differences in sequence length.
- Equal token counts can still leave uneven compute because attention cost grows quadratically.

Bar width = predicted time · segments = examples · sequence lengths shown below

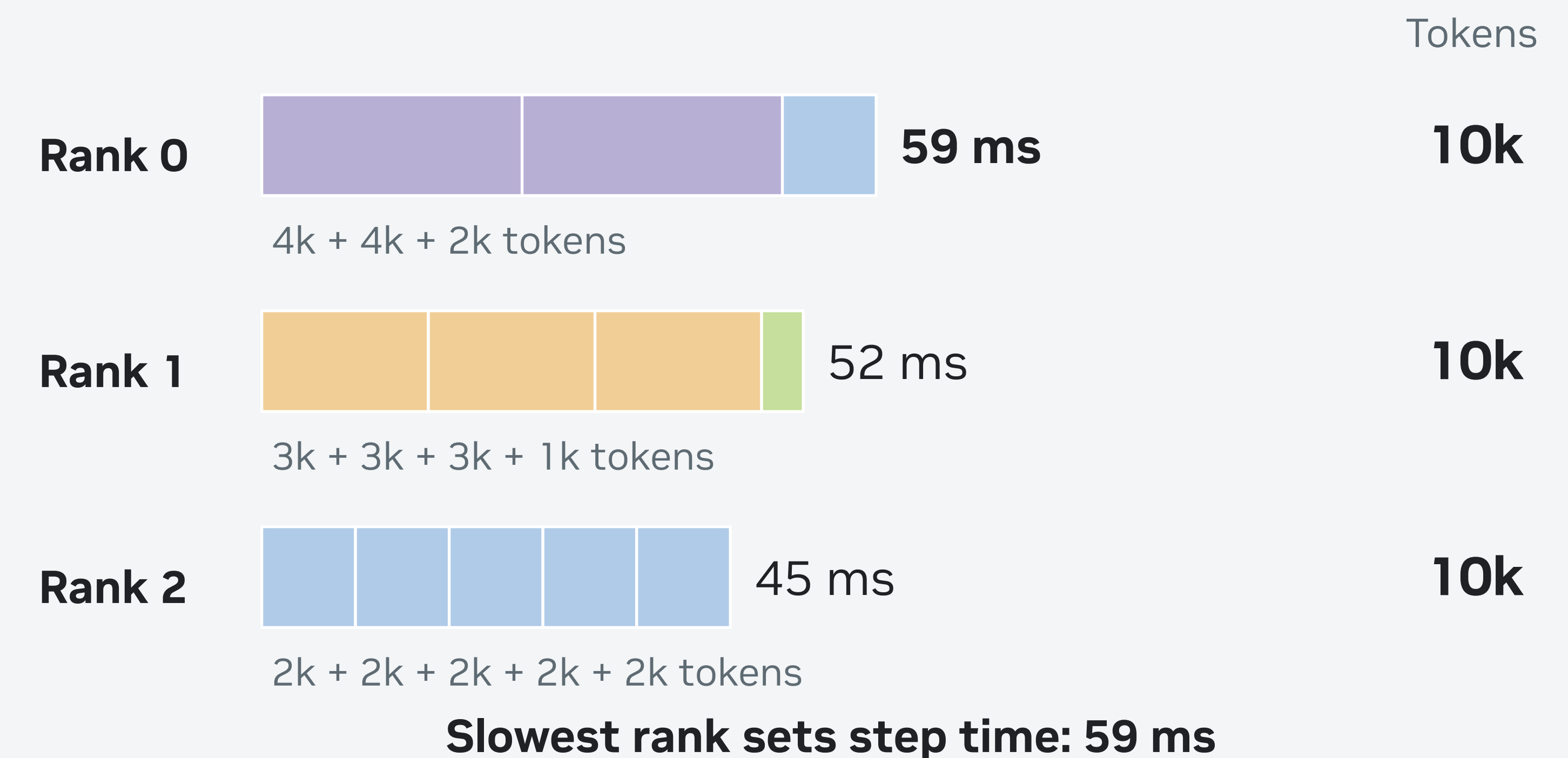
1 Balance samples

4 examples per rank



2 Balance tokens

10k tokens per rank (sequence packing)



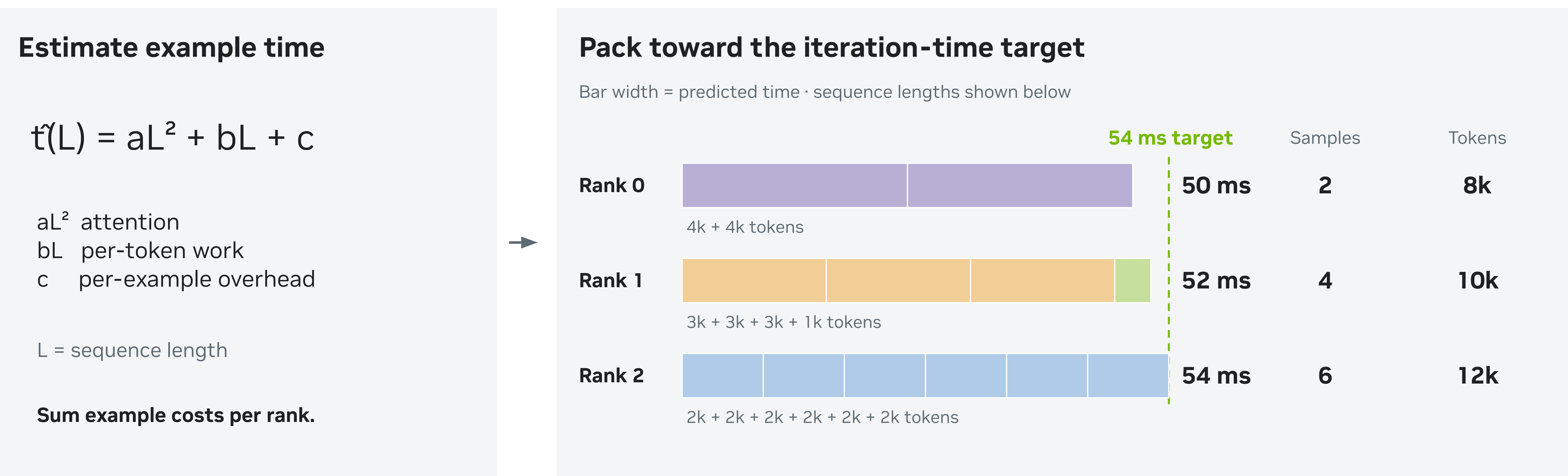
Both methods do NOT balance compute, and lead to GPU underutilization.

Illustrative timings · same twelve examples in both panels · model details on the next slide

We Use a Cost Model to Balance Compute

We balance predicted compute using a linear + quadratic cost model.

- Predict each example's time from its sequence length using cost model.
- Pack toward a shared iteration-time target; sample and token counts can vary.



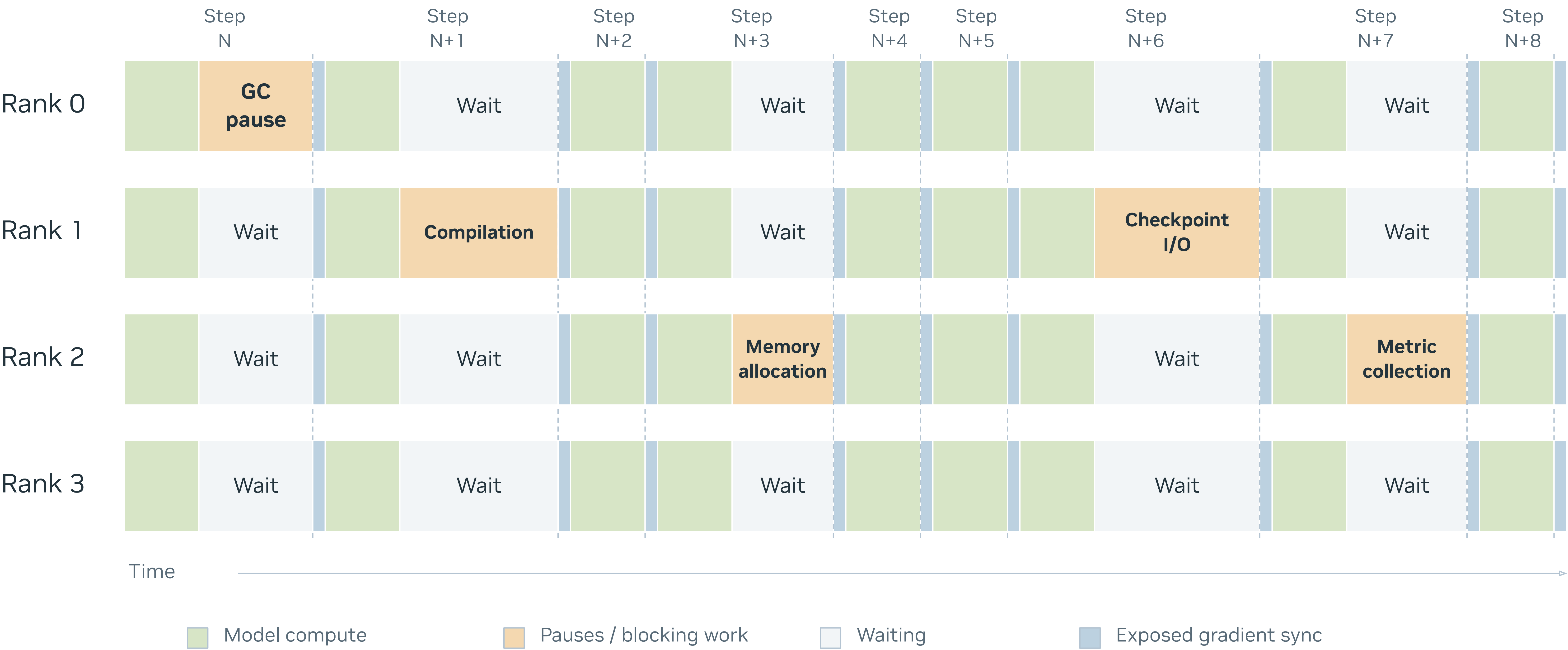
Cost model balances compute across ranks improving GPU utilization.

Illustrative fit: $\tilde{t}(L) = L^2 + 2L + 1$ ms; L in k tokens. Same twelve examples; fixed step overhead excluded.

One Rank's Stall Can Delay Everyone

Balanced batches do not eliminate host-side stragglers

Nine training steps: each local pause makes the other ranks wait at synchronization.



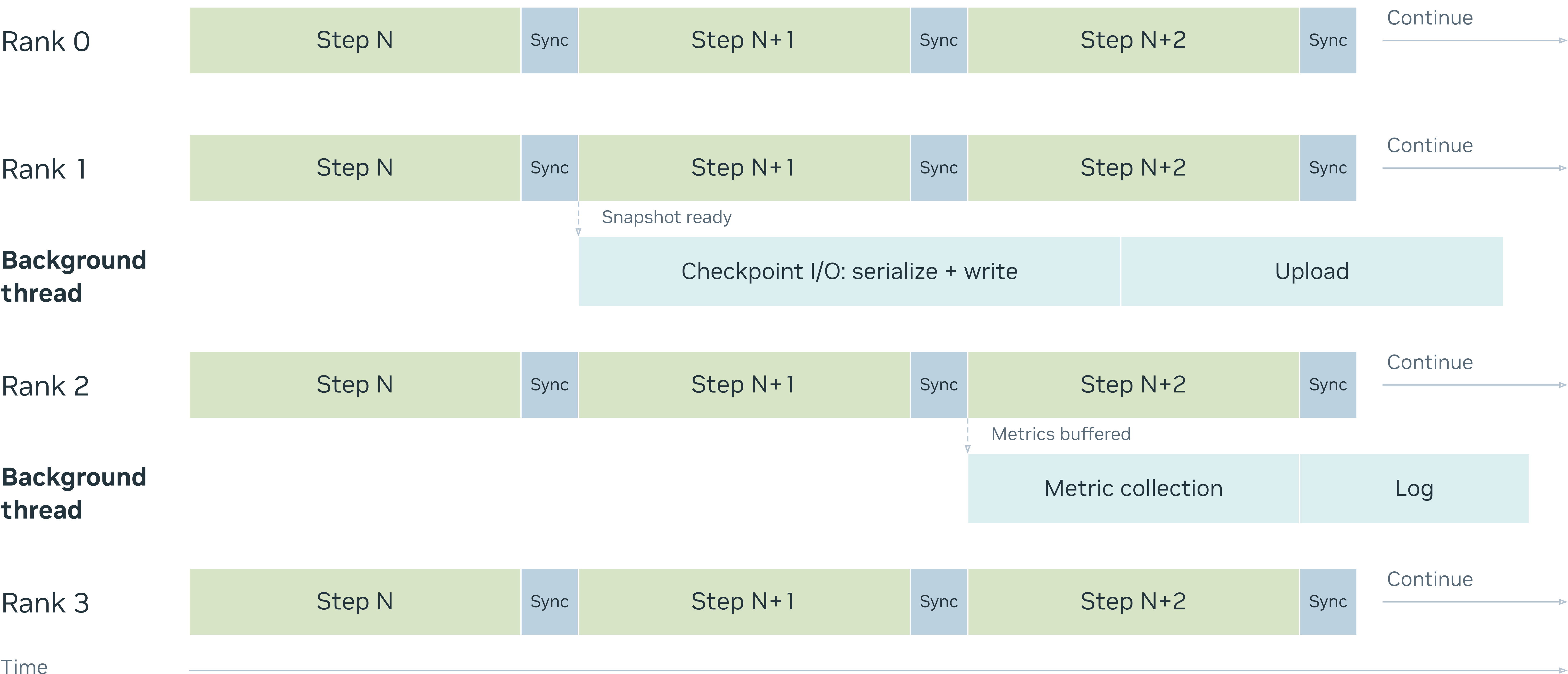
Synchronized execution - any delay in one rank amplifies total slow down.

Illustrative schedule, not a measured timing trace.

Keep Checkpointing and Metrics Off the Critical Path

Background threads handle checkpoint I/O and metric collection

Save/load optimizations target slower object stores such as S3 and GCS, where tail latency can reach 10–100× the median.

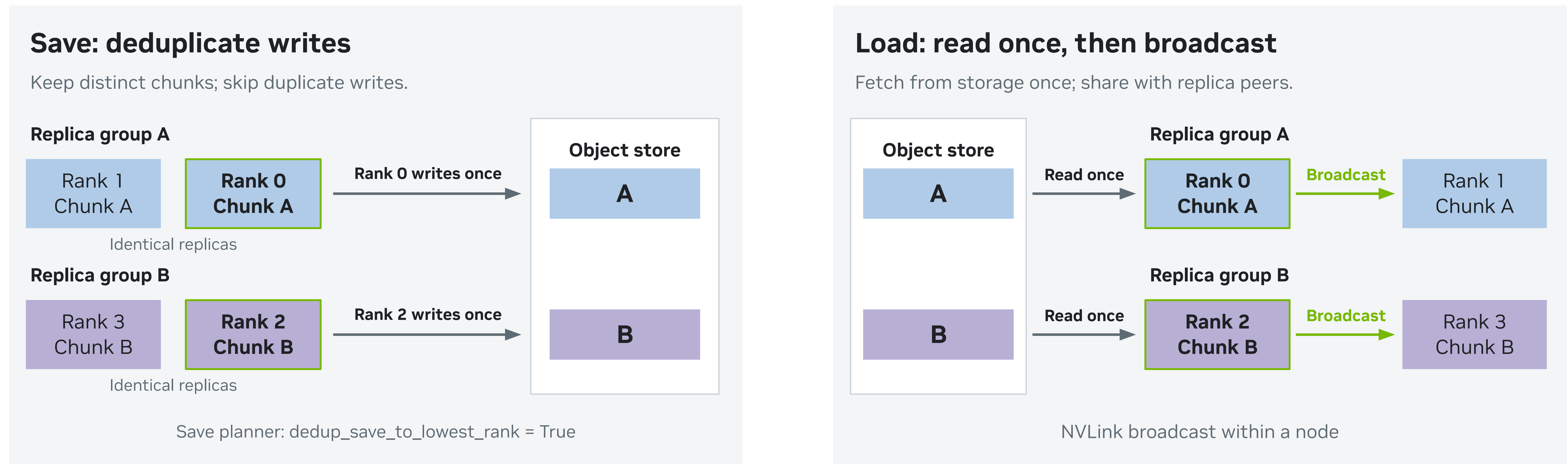


Illustrative schedule; snapshot staging and buffer handoff are not shown.

Efficient Checkpoint Saving and Loading

Save one copy of replicated chunks; read once and broadcast on load

- Save: write each replicated tensor chunk once through its lowest-rank owner, cache save plans.
- Load: read each replicated chunk once per replica group, then broadcast it to peer ranks.

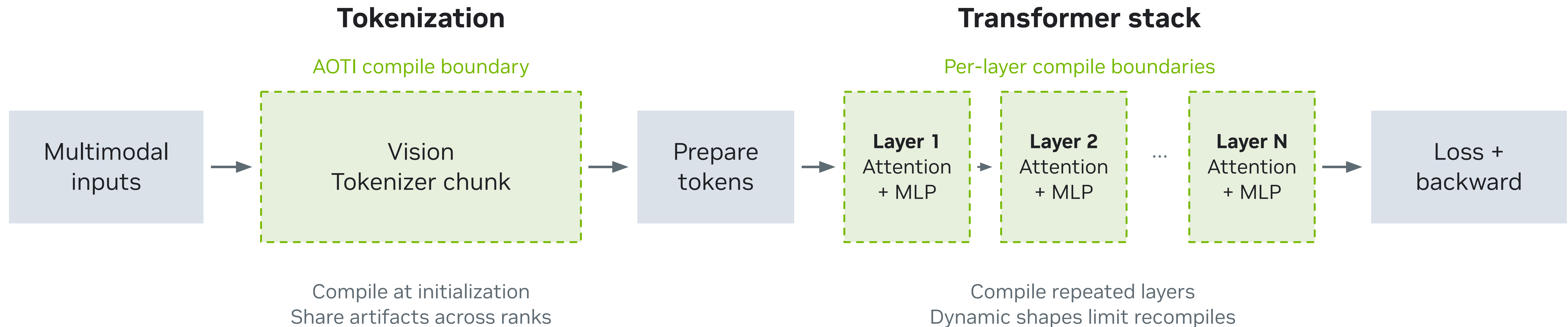


Reduce redundant object-store I/O during checkpoint saves and loads.

Illustration: A and B are distinct tensor chunks, each replicated across two ranks. Only redundant copies are deduplicated.

Compile the right regions

Separate compilation boundaries for tokenizers and transformer layers



Normalized execution time

Each model's baseline = 100 · lower is better

Baseline



100

Compiled



75-85

Lighter segment: model-dependent range

15-25%

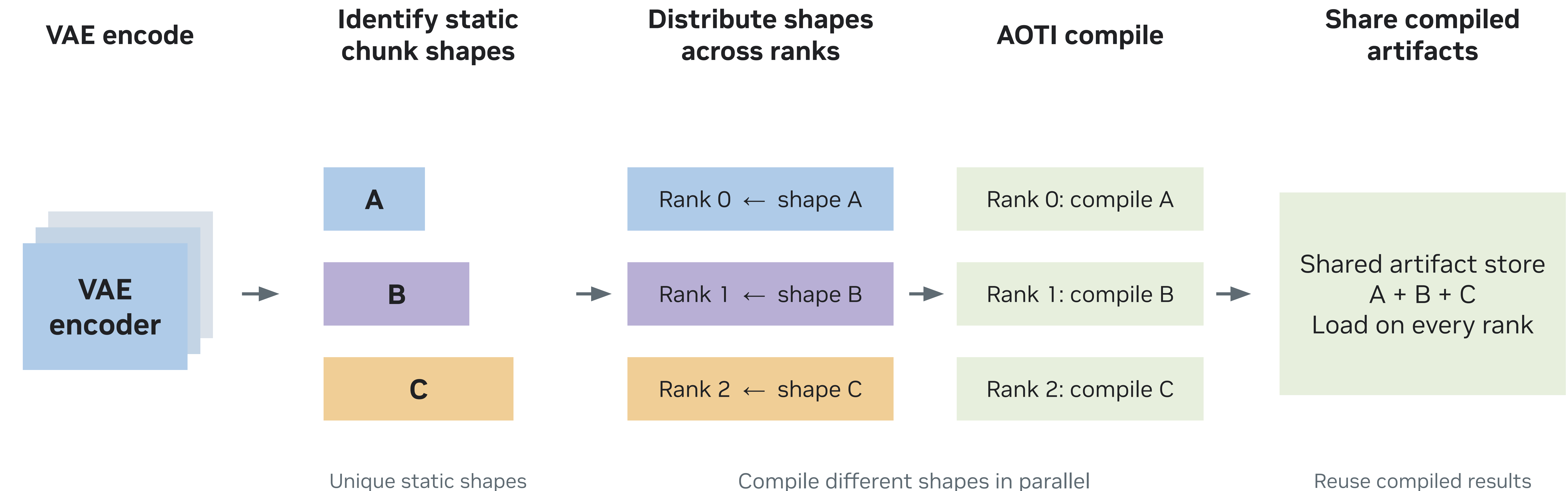
lower execution time

Profile expensive regions. Compile within deliberate boundaries.

Precompile VAE chunk shapes

Distribute AOTI compilation across ranks; share the resulting artifacts

- Identify static VAE chunk shapes and distribute AOTI compilation jobs across ranks.
- Compile during initialization, then share and load the compiled artifacts on every rank.

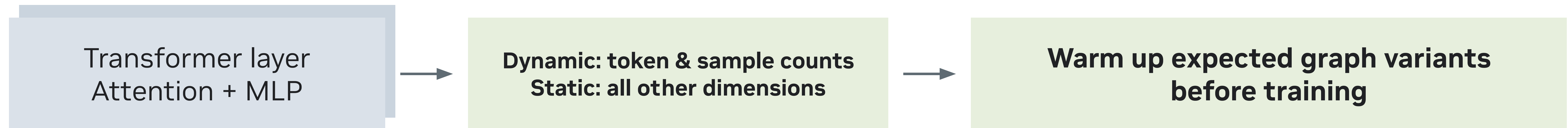


Compile each unique VAE chunk shape at initialization; share and load artifacts across ranks.

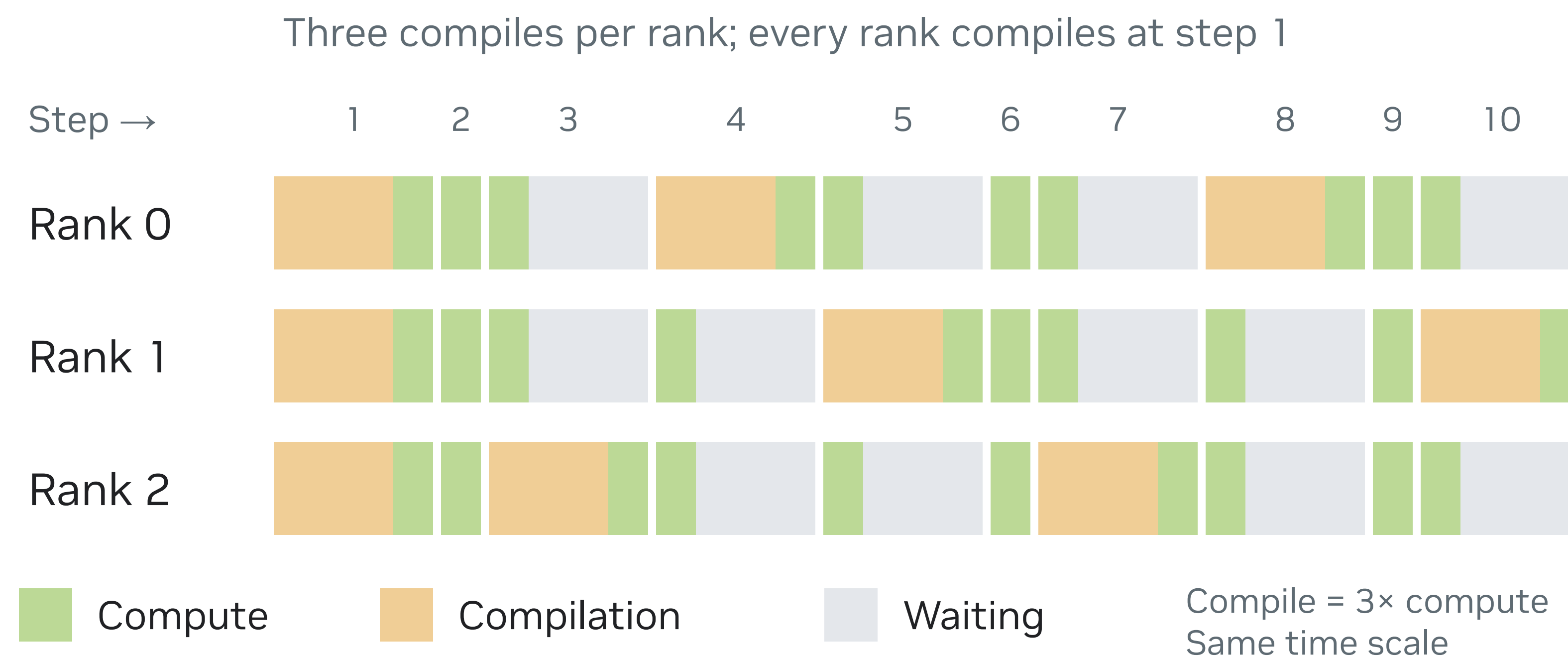
Warm up dynamic transformer shapes

Only token and sample counts are dynamic; all other dimensions stay static

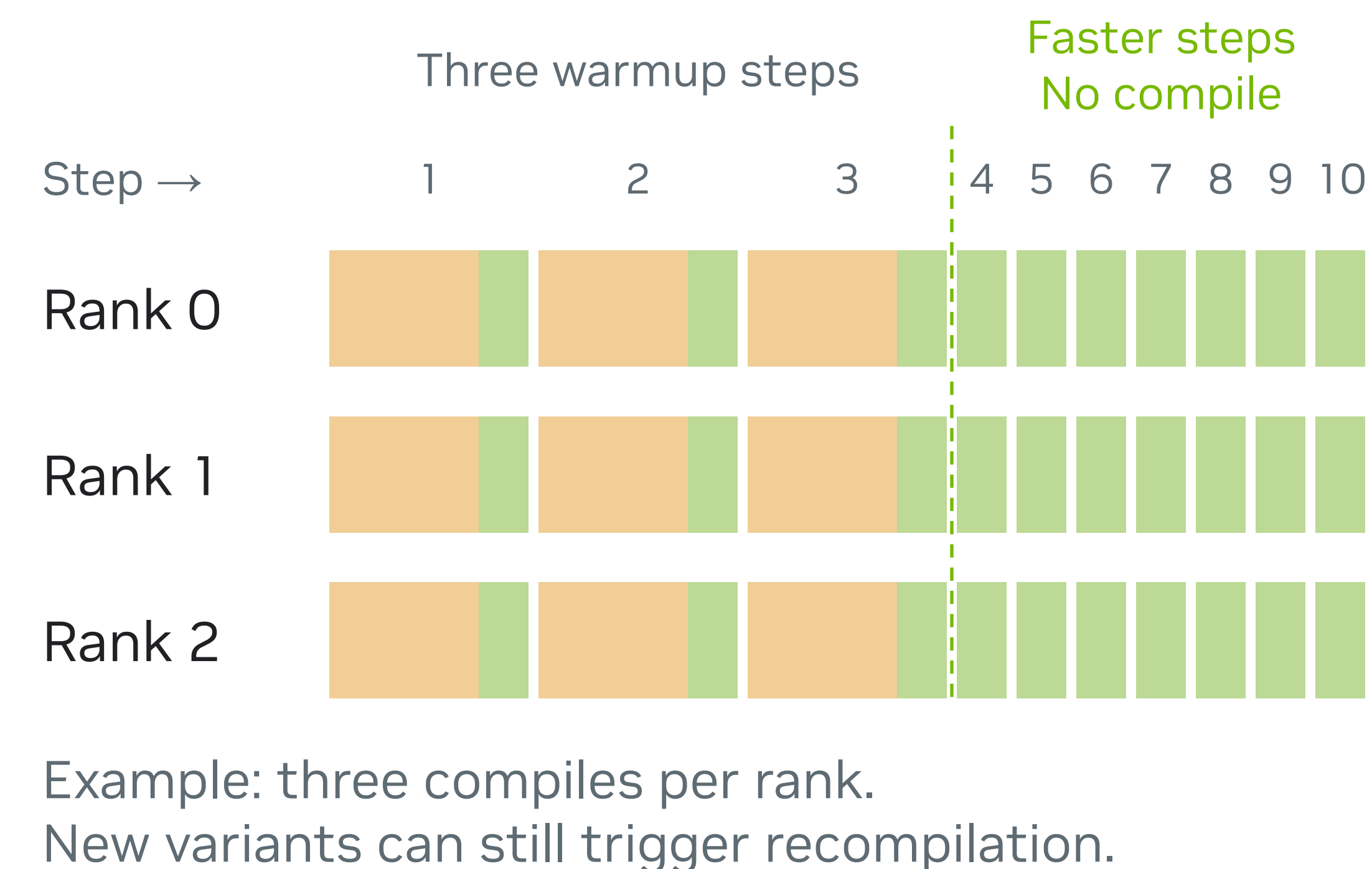
- Compile each transformer layer with dynamic token and sample counts; keep all other dimensions static.
- Warm up expected variants on every rank before training to reduce compilation stragglers.



Scattered compiles across 10 steps



Warmup: compile in steps 1–3



Unbacked shapes
Too fragile for this training path

Warm up expected variants on every rank before training.

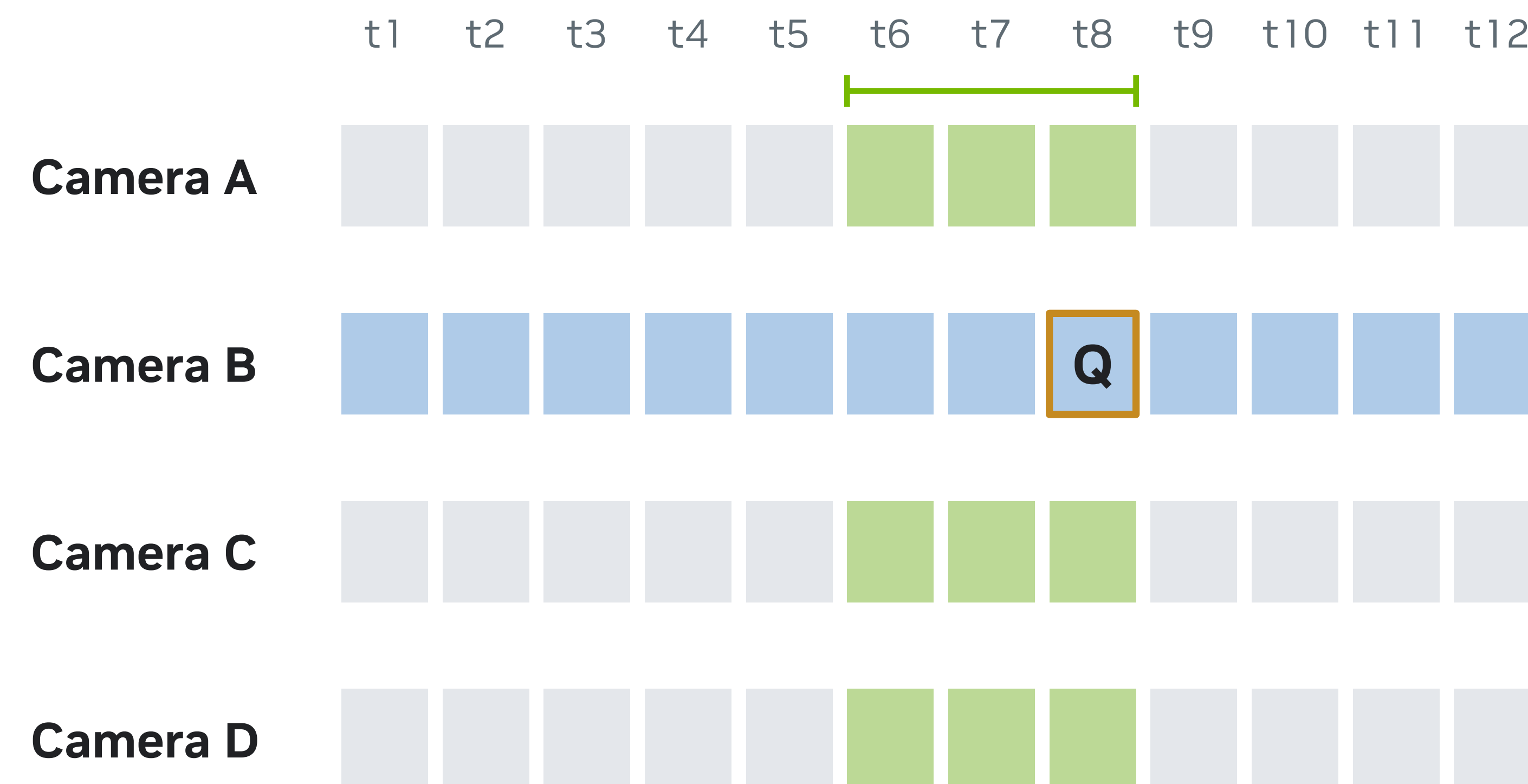
Multiview Modeling Needs Sparse Attention

Sparse attention with a sliding window across cameras

- Multiple cameras views → jointly denoise all views with a shared transformer.
- Attend to all tokens on same view, use sliding window for cross view → introduces sparsity in attention patterns.

A narrow cross-view temporal window

12 timestamps · N tokens/frame for every camera



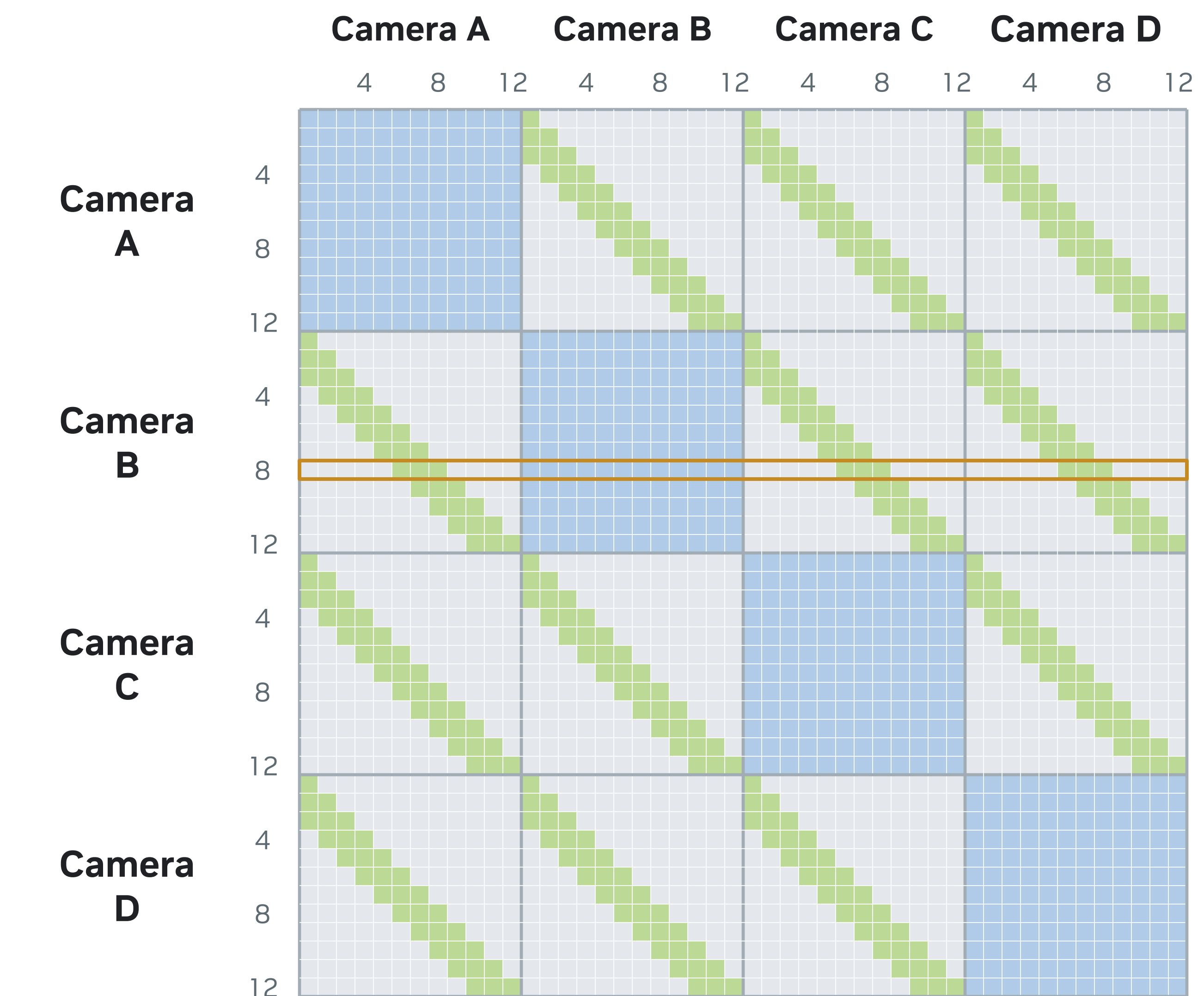
Query B,t8 reads t6–t8 in every other stream.

Causal across views only.
Each stream keeps full temporal context.

Camera tokens

Cross-view window

Attention mask: sized by token count



Query × key tokens; each camera = N tokens / frame.

Blocked

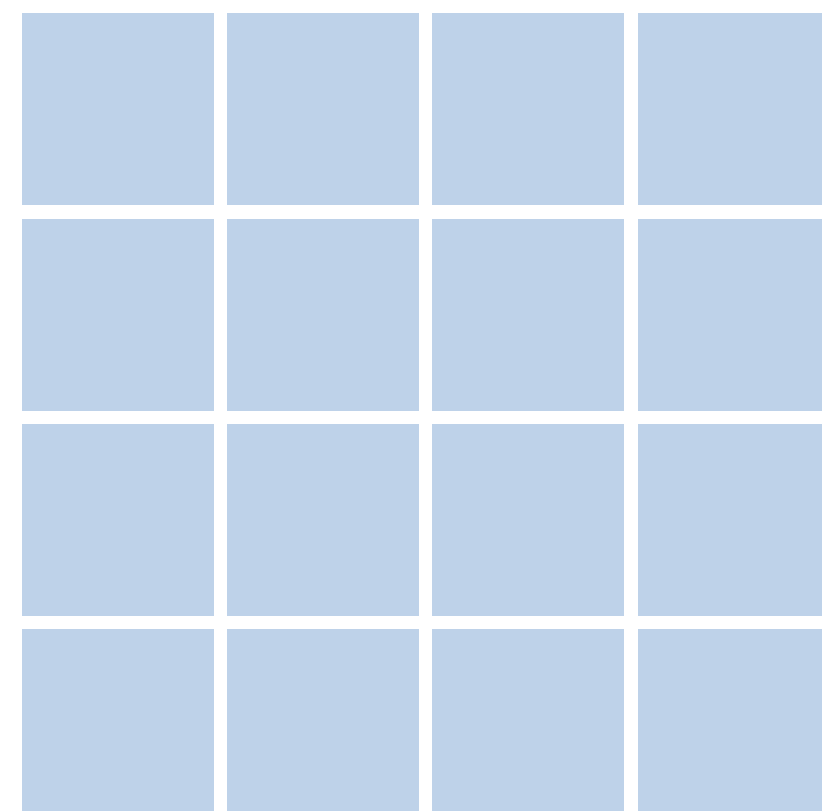
Selected query

Sparse Attention: Evaluating FlexAttention

Our evaluation exposed a performance issue with partial blocks

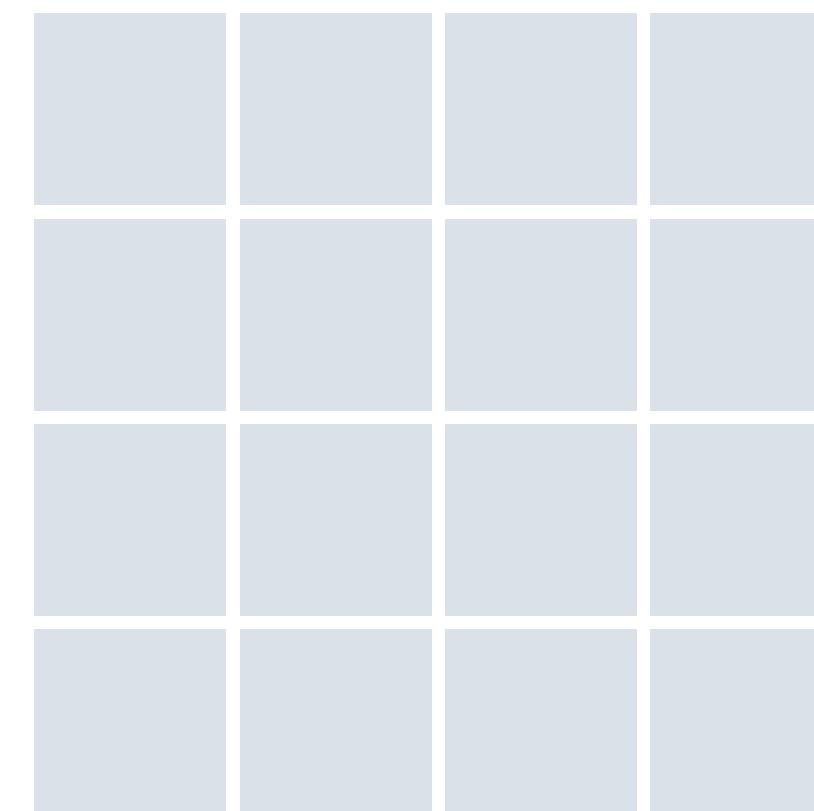
- We evaluated FlexAttention (with FlashAttention-4 backend) for our sparse attention pattern.
- **mask_mod** and **score_mod** made the pattern straightforward to express.
- Our evaluation found that partial blocks added masking overhead and reduced GPU efficiency.

Full block



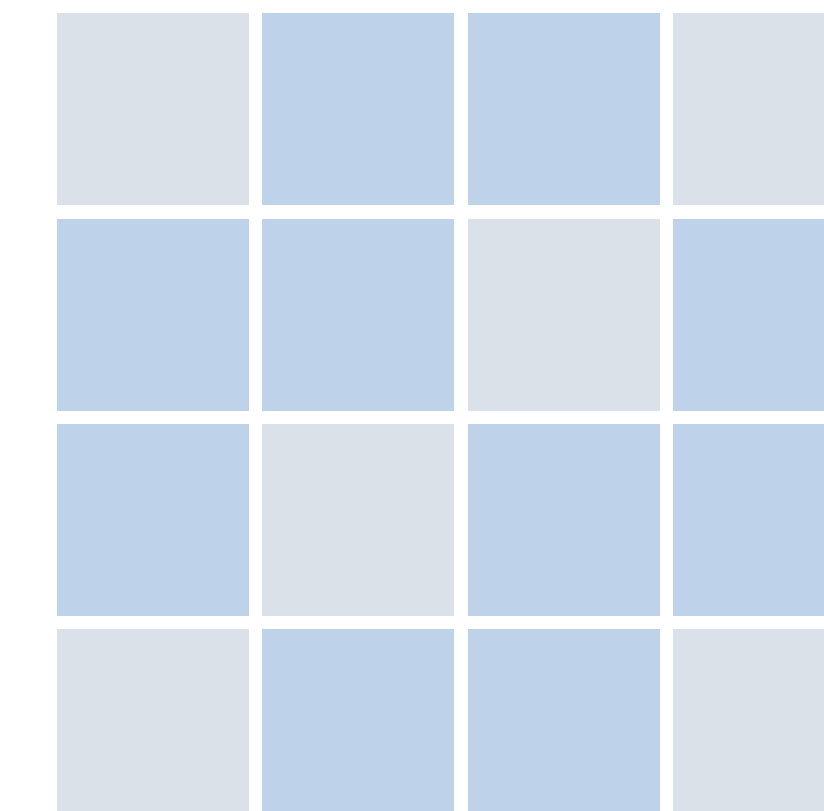
Compute without elementwise masking

Empty block



Skip the block

Partial block



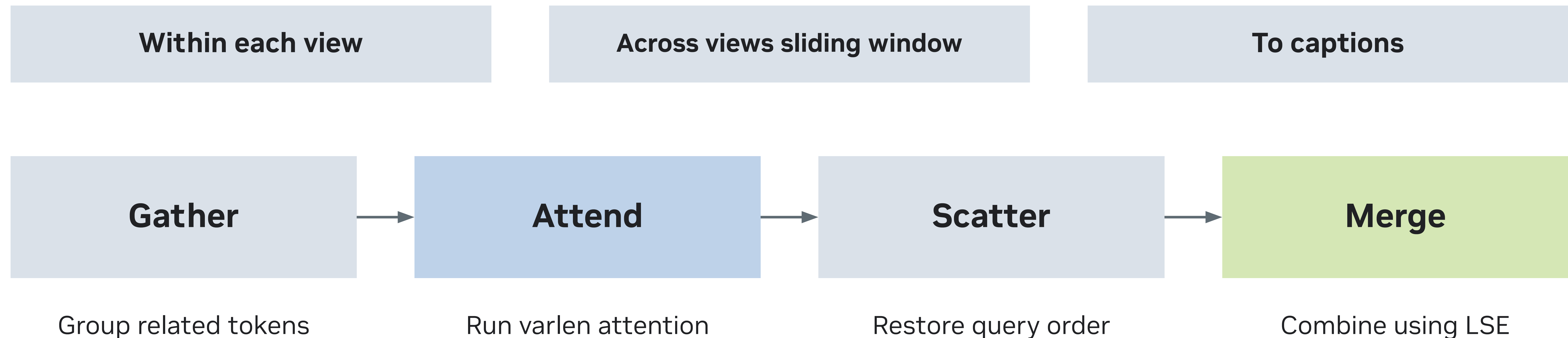
Elementwise masking
Reduced GPU efficiency

For sliding window cross-view attention in multiview training, MFU dropped from 55% to 15%.

Multiview Attention: Gather, Attend, Merge

Regroup interactions into efficient variable-length attention calls

Cosmos example: multiple camera views, and captions



Reported 2.5-3x speedup for the tested attention operations

Validate output and gradient agreement; measure full-step benefit.

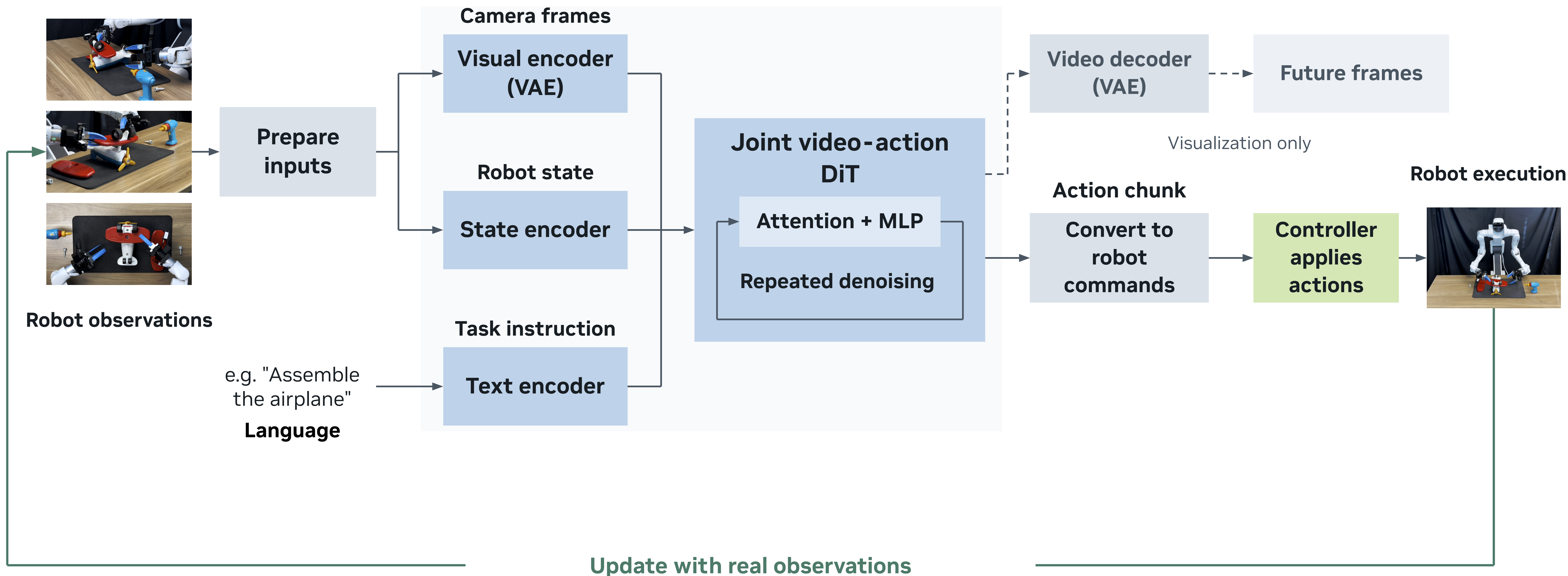
Merge normalization correctly and account for overlapping interactions.

From Fast Prediction to Continuous Motion



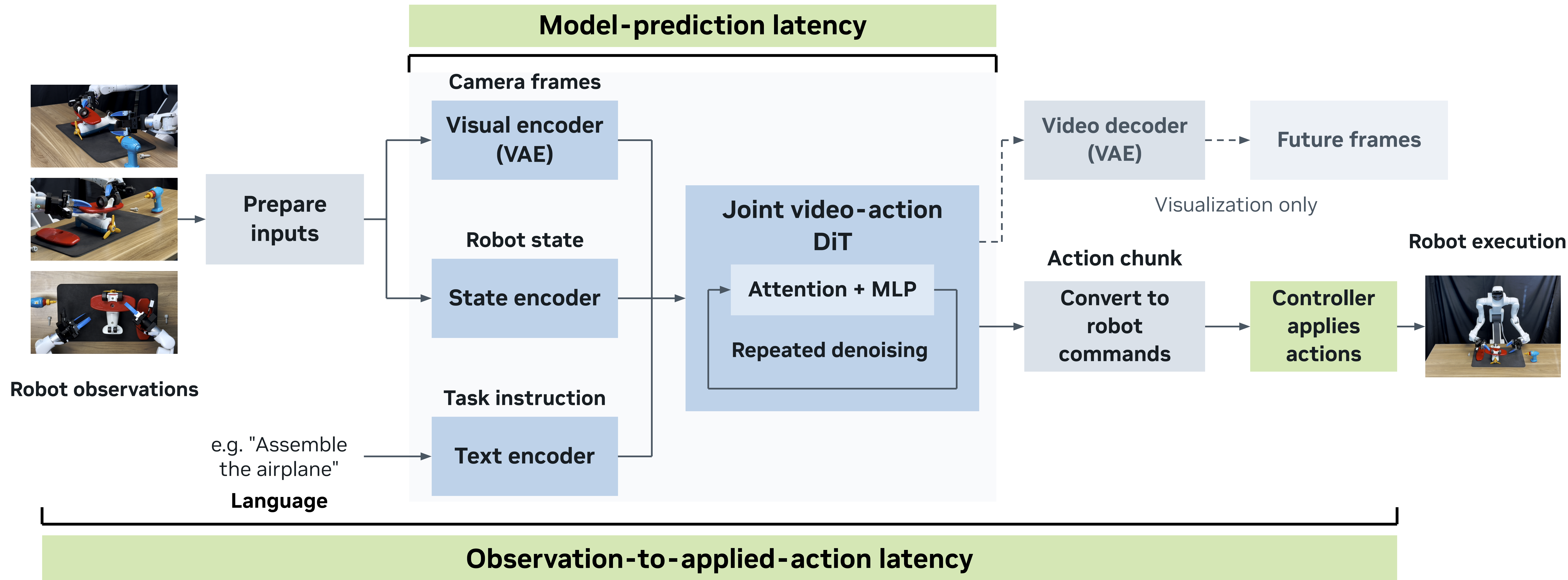
From Observation to Action

Closed-Loop Real-World Execution



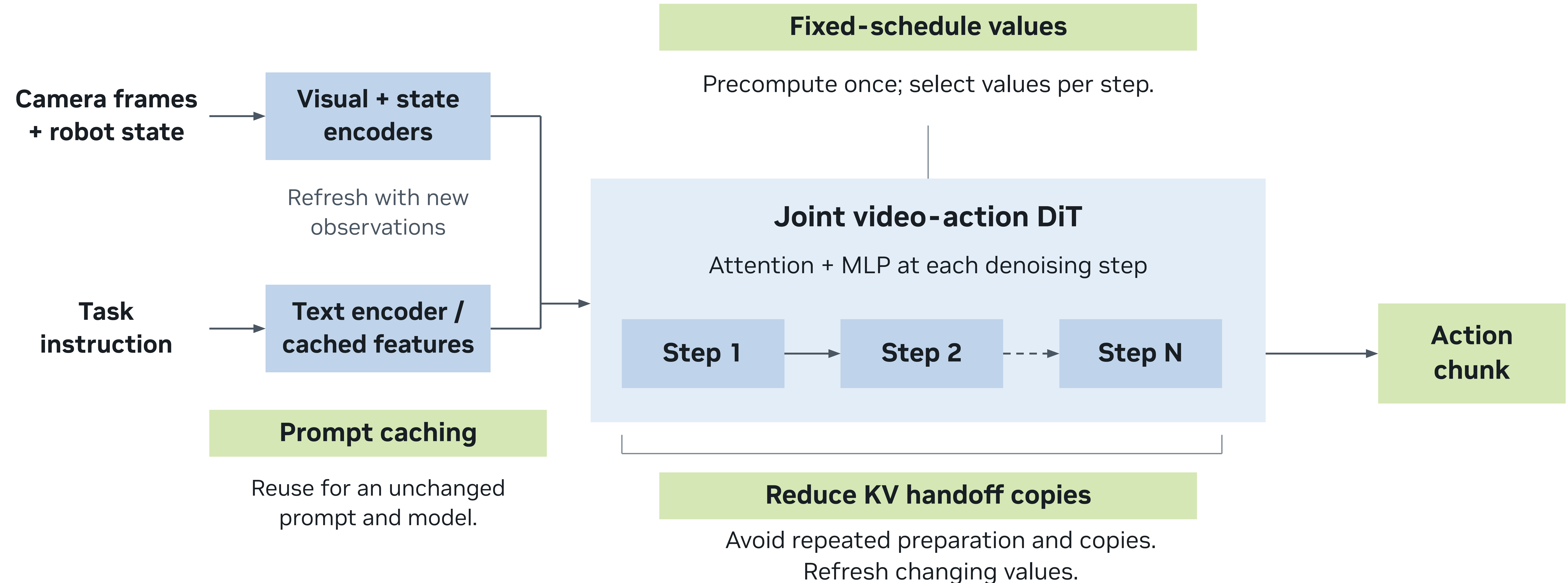
From Observation to Action: The Latency Budget

Optimize the full path, not just model prediction



Less Work in Every Prediction

Inside model prediction: reuse stable inputs and storage



Reuse what stays unchanged; recompute what changes.

Faster Execution of the Prediction Path

Faster steps, less launch overhead, and fewer iterations

Optimized execution

GR00T: TensorRT engines for VAE + DiT.
Specialize execution for fixed input shapes.

CUDA Graph replay

Replay encoding + denoising as one graph.
Remove repeated CPU dispatch between stages.

Selective quantization

FP8 joint DiT + VAE; BF16 action engine.
Keep sensitive layers at higher precision.

Denoising schedule tuning

Reduce video/action denoising iterations.
Fewer model evaluations; validate task quality.

GROOT WAM: Prediction Latency Optimization Journey

Illustrative cumulative latency estimates; lower is better

Eager BF16 = 100

Normalized latency

Eager BF16 baseline



Optimized execution (TensorRT)



+ CUDA Graph replay



+ Runtime optimizations



+ Selective FP8 quantization



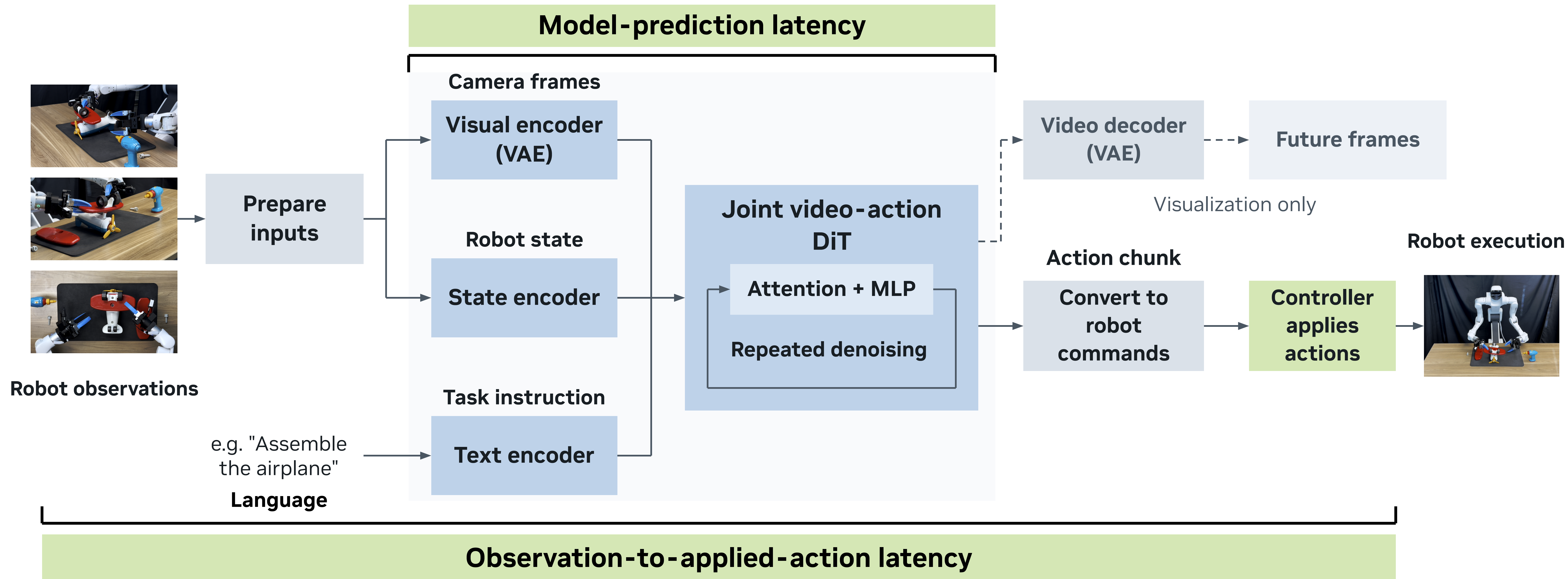
+ Denoising-step tuning*



*Task/checkpoint-dependent

Faster Prediction Is Only Part of the Response

Now optimize the work around prediction



Beyond Model Prediction

Optimize the CPU path and server responsiveness

Batch camera histories

Bounded CPU workers across cameras and history; preserve frame order.

Use faster codec paths

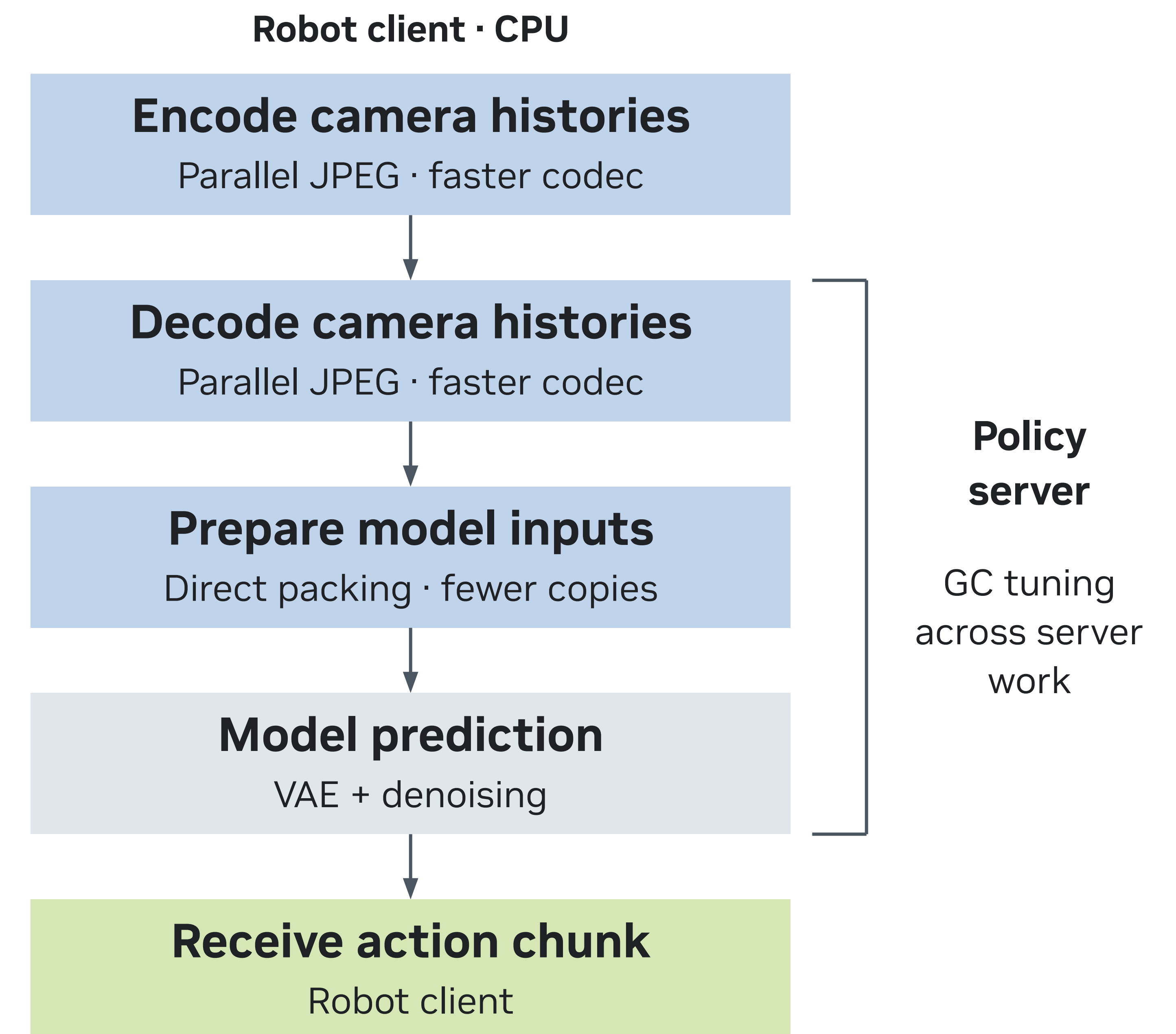
Accelerate JPEG encode/decode; keep quality and fallbacks.

Remove redundant copies

Build the final image layout directly; avoid redundant full-clip copies.

Reduce server-side pauses

Collect + freeze at startup/reset; reduce long pauses.



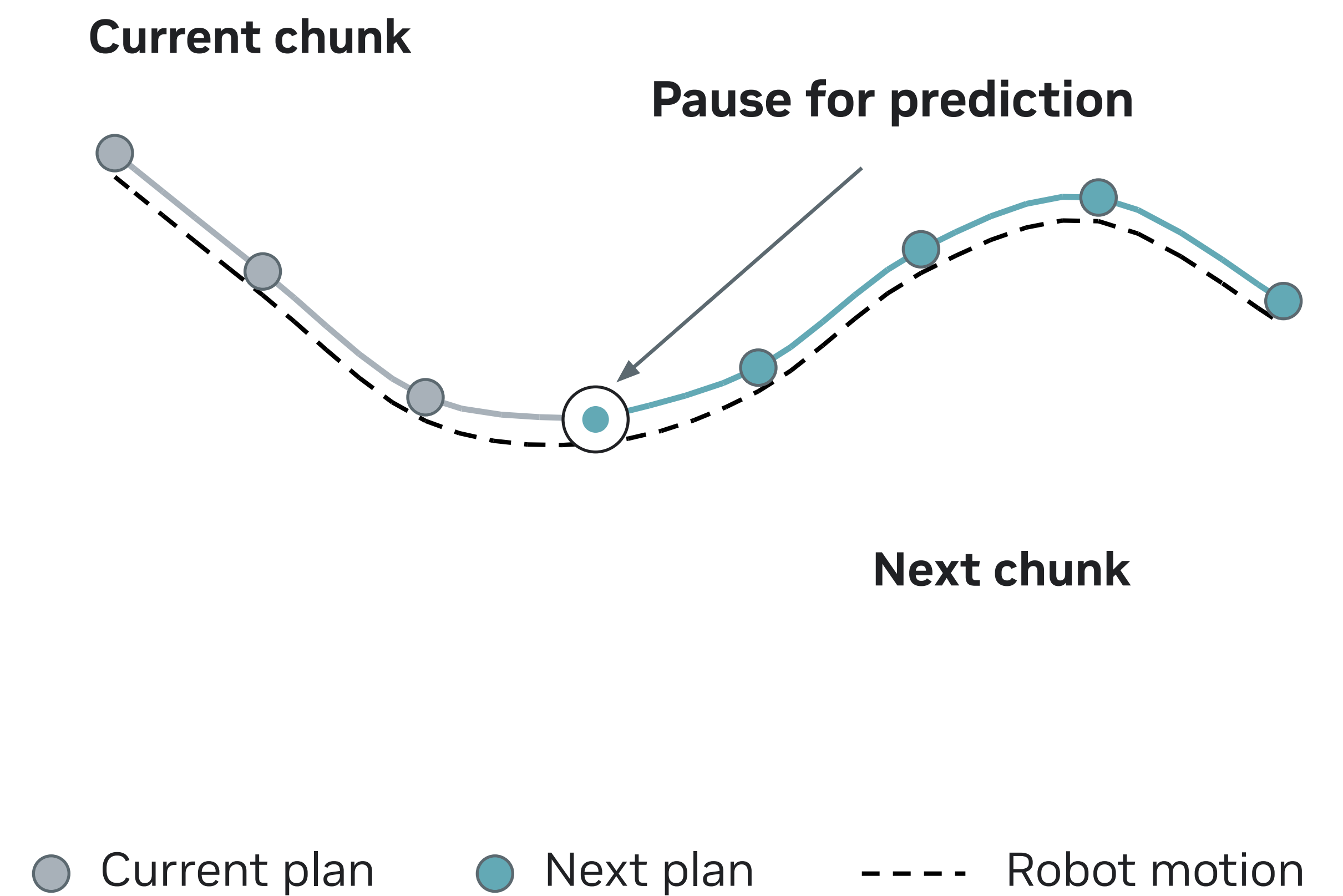
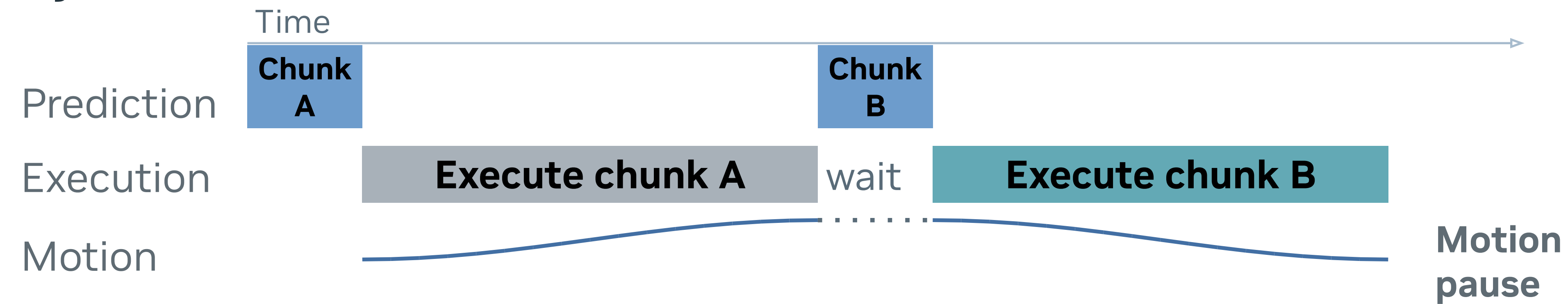
Measure the full round trip

Across embodiments and target hardware; track typical latency and rare stalls.

Prediction Continues While the Robot Moves

Synchronous: predict, execute, then wait

Synchronous

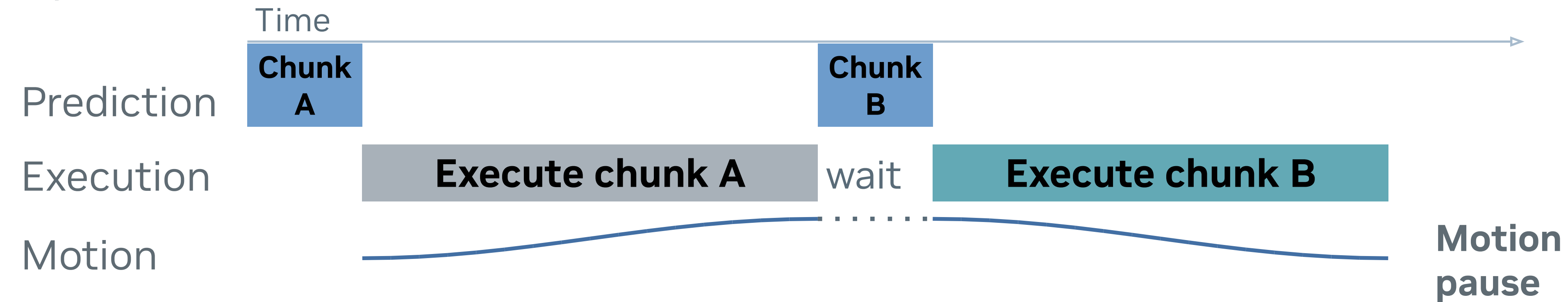


Action chunk = a sequence of robot commands. Motion and timing are illustrative.

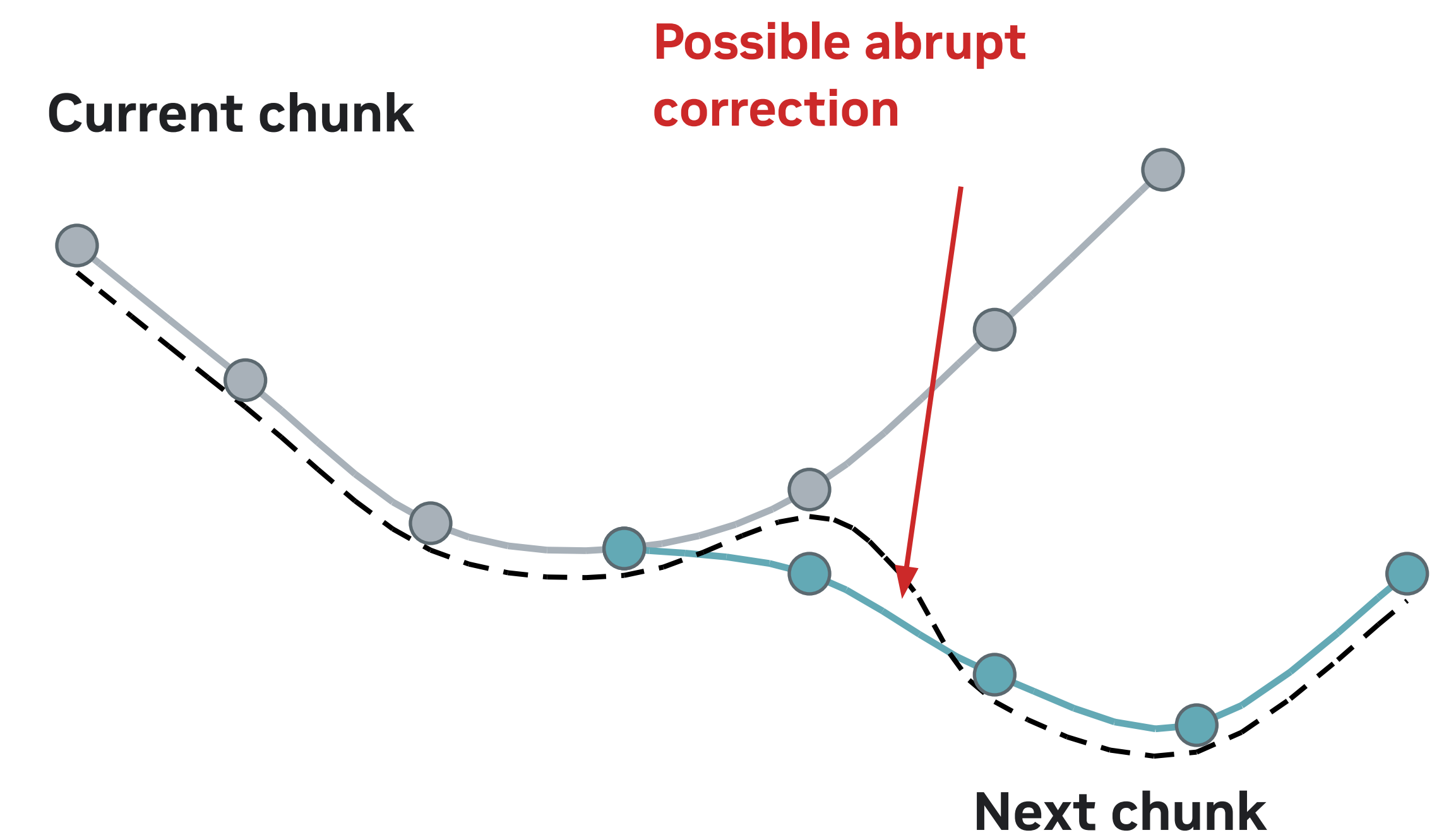
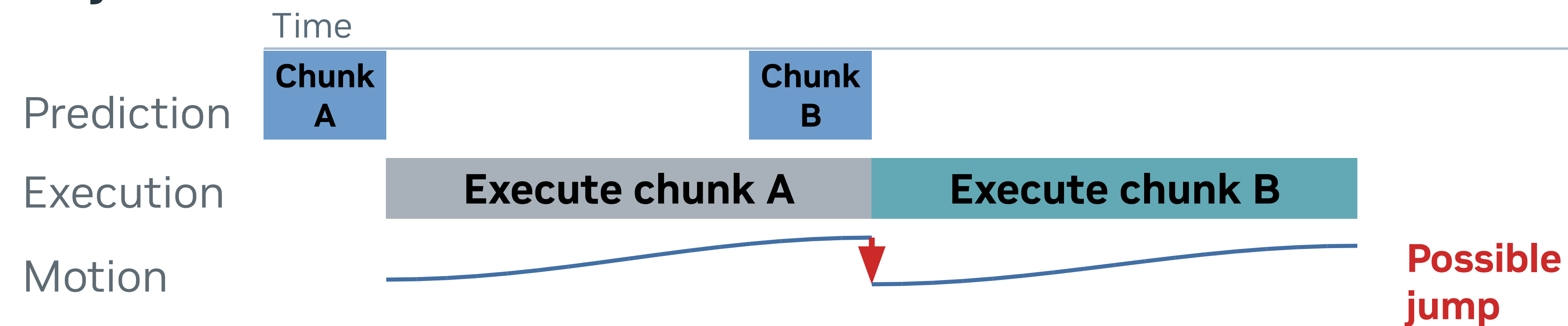
Prediction Continues While the Robot Moves

Asynchronous: overlap prediction and execution

Synchronous



Asynchronous



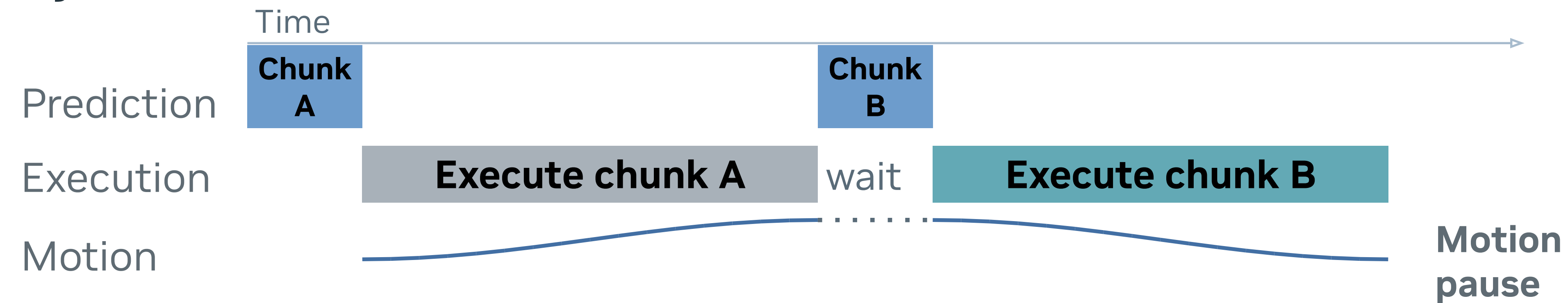
A new plan can disagree with the ongoing move.

Action chunk = a sequence of robot commands. Motion and timing are illustrative.

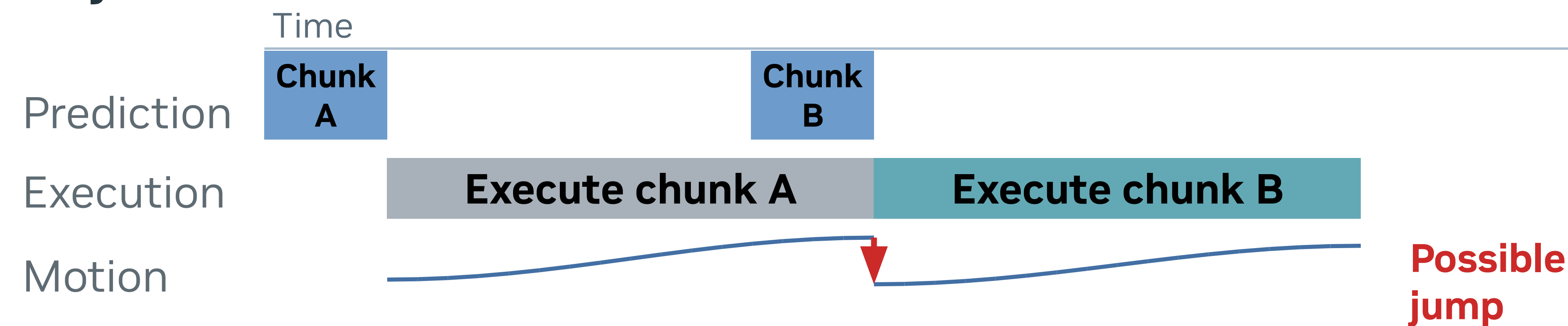
Prediction Continues While the Robot Moves

Training-Time RTC: learn to continue committed actions

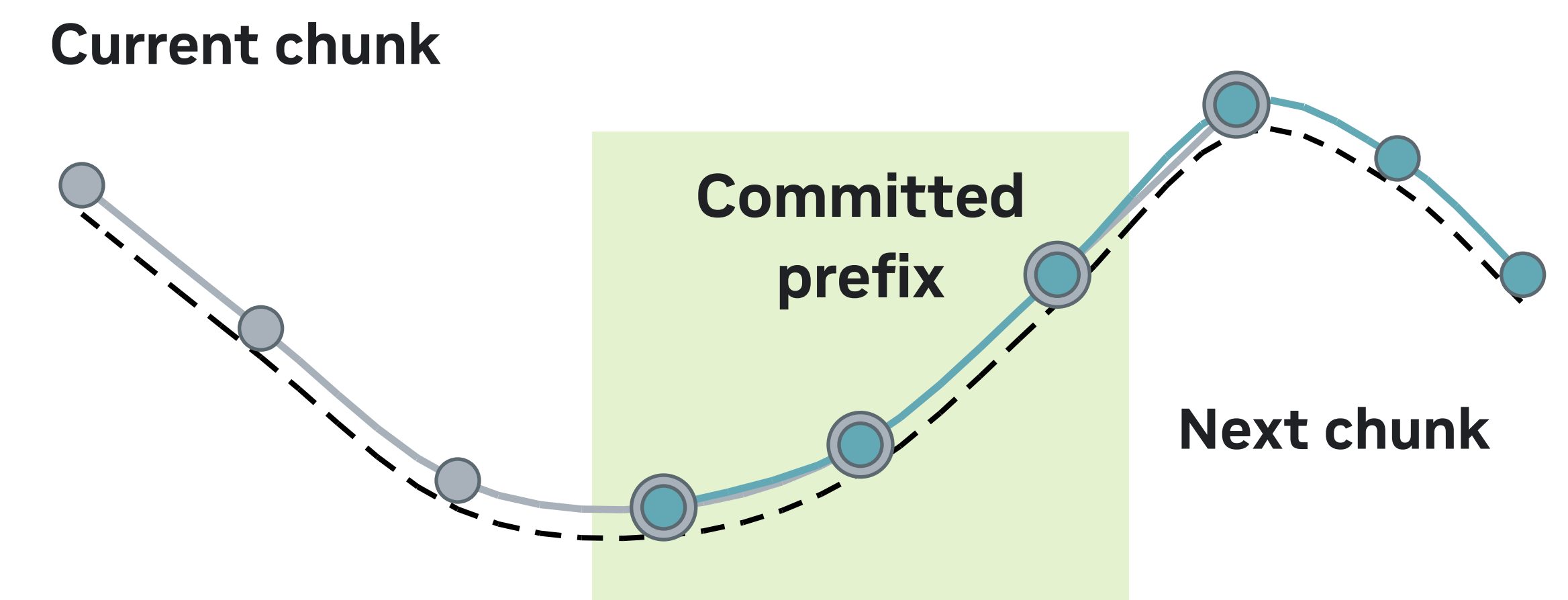
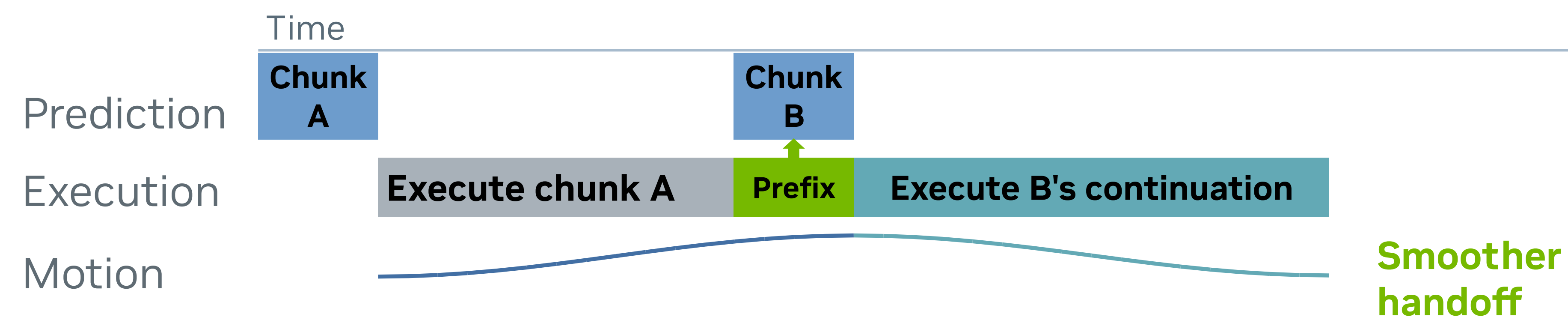
Synchronous



Asynchronous



RTC: condition on committed actions



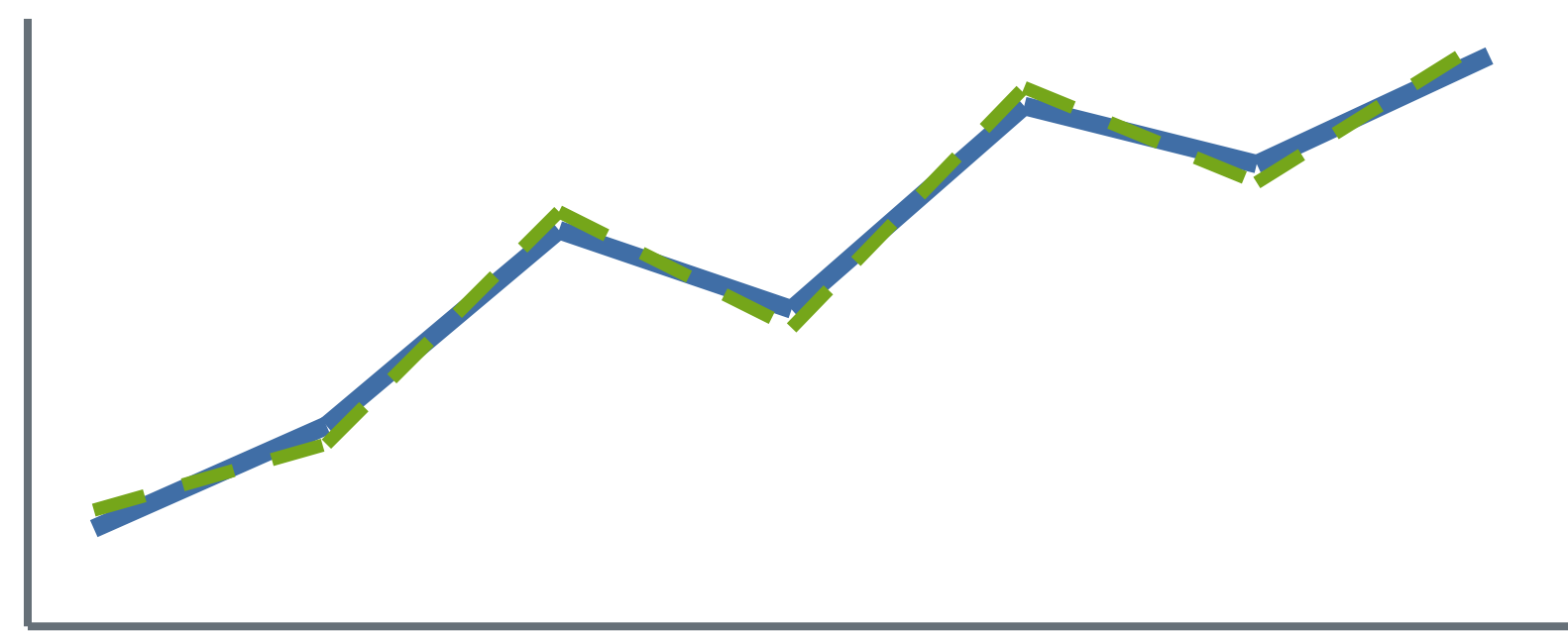
● Current plan ● Next plan - - - - Robot motion

Queued actions stay fixed while the model predicts a compatible continuation.

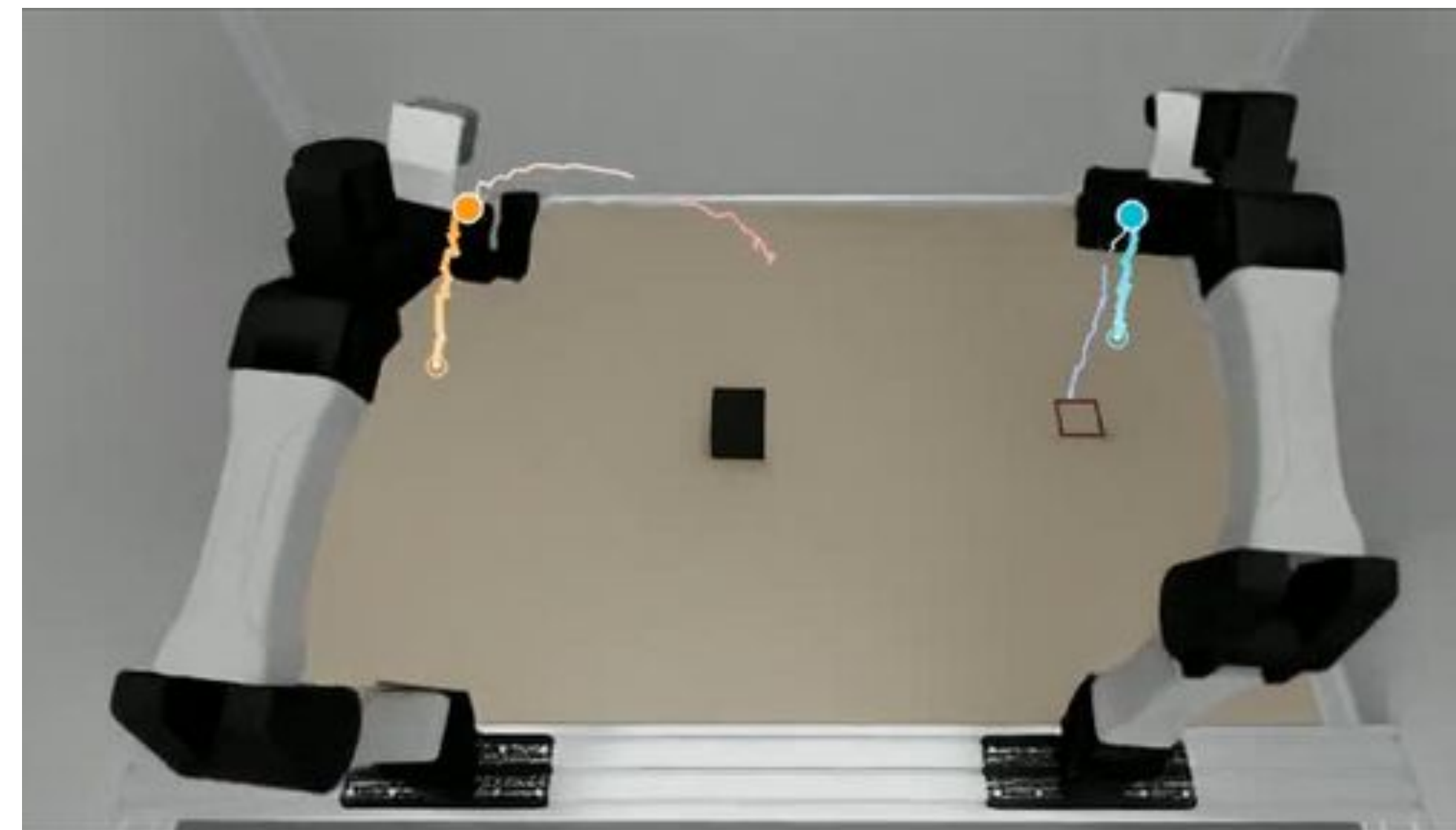
Action chunk = a sequence of robot commands. Motion and timing are illustrative.

Validate Speed and Robot Behavior

Measure latency, task success, completion time, and motion smoothness



Illustrative: **recorded** vs. **predicted** actions



Numerical checks

Simulation

Real-robot evaluation

Open-loop prediction accuracy

Compare with baseline

Task success

Behavior across scenarios

Task success

Completion time

Motion smoothness

Takeaways

- **Balance compute, not sample counts.**
- **Keep deferrable work off the training critical path.**
- **Choose compilation boundaries and shapes deliberately.**
- **Design inference for timely, continuous robot motion.**
- **Validate task success, not just speed.**

Thank you

Questions? Visit us at
NVIDIA exhibit booth D1.

