SCALING ROBOT POLICIES OVER WIDE-AREA NETWORKS

TECHNICAL REPORT

General Sequences*

November 2026

ABSTRACT

A robot that runs its policy on a remote GPU must receive each chunk of actions before the current one runs out; in simulation, 400 ms of delay cut $\pi_{0.5}$'s success on LIBERO-10 from 96% to 24%. GS-WAN treats this deadline as one budget for the network and the GPU, so that time saved on the network lets a GPU serve more robots. GS Inference answers robots together, holding each request only as long as its robot can spare, accepts only the robots it can keep on time, and runs a slower planner beside the policy without delaying its actions; GS Network streams cameras as low-latency video and removes the delay that robot software adds after each pause; and a routing layer gives each robot the closest GPU it can obtain. Answering robots together cut the GPU cost per robot by up to 2×, and with more robots asking than one GPU could serve, GS Inference kept every robot it accepted on time where Ray Serve and Triton, which accept every robot, kept none. GS Network cut a request's time outside the server by 2.7× and 4.5× against openpi's and GR00T's own servers, and every robot GS-WAN accepted over real networks was on time. Both layers are open source at `https://github.com/General-Sequences/sequence-ai`.

1 Introduction

Large robotics policies, such as vision-language-action (VLA), diffusion and video-action models, need a GPU that most robots do not carry, so a robot sends its observations over a wide-area network (WAN) to a remote GPU and receives its actions back. At scale, many robots in different places share the GPUs of a few cloud regions.

A robot keeps executing its current actions while it waits for the next ones, so every request has a deadline: the response must arrive before the robot runs out of actions (Figures 1 and 4). **We call *scaling* the problem of serving many robots, spread across different regions, from a limited number of GPUs, while every robot still receives its next actions in time.**

*Please cite this work as "General Sequences (2026)". Author: Yongzhe Wang (see Authorship at the end of the document). Correspondence regarding this technical report can be sent to founders@generalsequences.com.

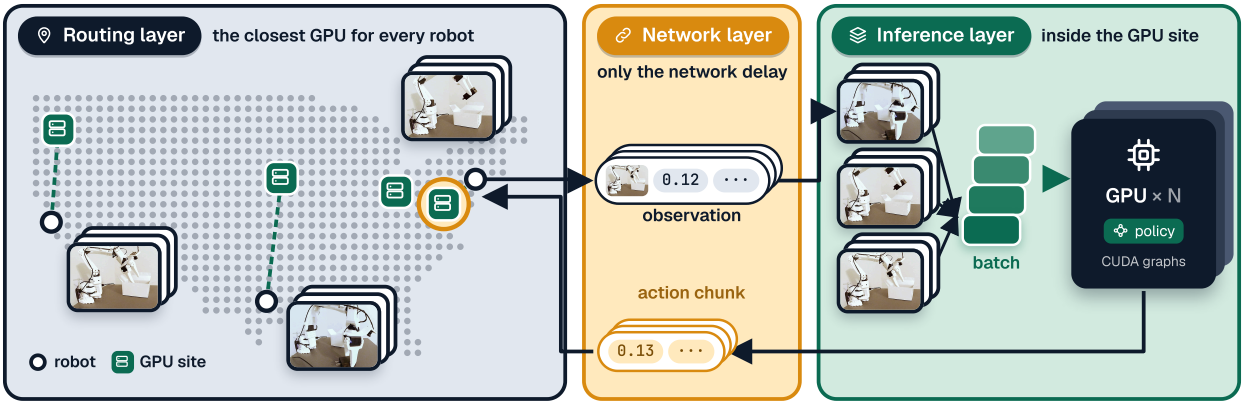


Figure 1 | Overview of GS-WAN. Left, the routing layer: robots in different regions are each linked to the closest GPU site, and the request traced through the figure goes from the north-eastern robot to the circled site. Middle, the network layer: the robot's observation travels to that site and the action chunk comes back, over a connection kept fast across the robot's pauses. Right, the inference layer inside the site: requests from several robots wait only as long as each robot's remaining actions allow and are answered together in one pass on one GPU, and the part of the policy that runs many times per request is replayed from a recording of its GPU work (CUDA graphs).

Scaling is hard at each layer of this path; the section on each layer begins by measuring why.

1) **Inference: sharing a GPU without missing deadlines.** Every robot must receive its next actions before its current ones run out. Answering several robots together in one pass on the GPU, which we call a *batch*, raises a GPU's throughput but makes each request wait longer. We want to put as many robots on a GPU as their deadlines allow, and to make each inference faster without changing its outputs.
2) **Network: extra delays after each pause.** With today's robot software, which sends one request per chunk, a robot uploads camera frames, waits for its actions and pauses, again and again. Networks and servers are tuned for other traffic, and after each pause the next upload can take several times the network delay. We want to remove them without changing the client a robot already uses.
3) **Routing: distance consumes the time budget.** Every request pays the *network delay* to its GPU, the time a message takes to get there and back, and that delay comes out of the same time budget that batching needs. Where a robot is served therefore decides how many robots its GPU can hold. We want to place each robot so that enough of its budget is left to share the GPU.

We introduce GS-WAN (General Sequences Wide-Area Network), an inference framework for serving robot policies over WANs that combines a few methods at each layer to address these difficulties (Sections 3 to 5). *GS Inference*, the inference layer, groups requests from several robots into one forward pass, as serving systems batch requests within a latency target [10, 12]. What it adds is that each request's deadline comes from its own robot, from the actions that robot has left and its own network delay, so the group runs before any of those robots runs out of actions. Before robots connect, it also computes how many robots a GPU can serve on time and sends the others to another GPU. The layer speeds up a policy with CUDA graphs [26], as `torch.compile` can [3], and keeps a graph only when its outputs match the uncaptured run exactly. *GS Network*, the network layer, removes the extra network delays that TCP and servers add to a robot's requests: it keeps the connection warm while the robot pauses, sends frames at the size the policy uses, compresses them only when that saves time, and returns each action chunk in one write. It also includes adapters for the existing clients of openpi and LeRobot. In the routing layer, a hosted service gives each robot the *closest* GPU it can obtain, so that its network delay leaves time for sharing the GPU, and keeps a *standby replica*, a second copy of the model, ready to continue the robot's episode. GS-WAN does not modify the policy, requires neither retraining it nor replacing the client a robot already uses, such as openpi's [30], and each of its components can be used on its own.

This report makes four contributions.

- **One deadline that ties latency to cost.** A request must arrive before its robot runs out of actions, so the time the network does not take is time the request can wait to share a GPU pass with other robots, and the GPU then keeps more robots on time. Replaying the batch times measured for $\pi_{0.5}$ on an A100, which itself kept twelve robots on time, a server that answered robots together kept eleven on time when all were 20 ms away

and eight when a third were 100 ms and a third 200 ms away; answering them one by one kept six either way (Sections 2 and 3.5).

- **GS Inference**, which answers several robots together on one GPU, holding each request only as long as its robot can wait, computes in advance how many robots a GPU can serve on time, and runs a slower planner beside the policy without delaying its actions. Answering robots together cut the GPU cost per robot by up to 2×, for $\pi_{0.5}$ on an A100 and for GR00T on an L4, and with more robots asking than fit, GS Inference kept every robot it accepted on time where Ray Serve and Triton, which accept every robot, kept none (Sections 3 and 3.7).
- **GS Network**, which removes the extra delays that today's robot software pays on a WAN, with server pings during pauses, single-write responses and compression only where it saves time, also for robots that keep openpi's or LeRobot's [6] own clients. With one robot 20 ms from its GPU, it cut a request's network latency, its time outside the server, by 2.7× against openpi's own server and by 4.5× against GR00T's; over real paths, Zenoh, the transport of LeRobot's remote inference, took up to 2.5× as long (Sections 4, 4.7 and 6.2).
- A **routing layer** that gives each robot the closest GPU it can obtain and keeps a standby replica ready (Section 5).

GS Inference and GS Network are released in the open-source Python package `sequence-ai` (`https://github.com/General-Sequences/sequence-ai`), with a tool that measures a robot's link (Sections 3.11 and 4.9).

The rest of this report follows a request along its path. Section 2 defines action chunks, a request's time budget and the scaling objective. Section 3 describes GS Inference, which shares and speeds up the GPU, Section 4 GS Network, which keeps the connection between a robot and its GPU fast, and Section 5 the routing layer, which decides where a robot's GPU is. Each begins with what we measured at its layer, and Sections 3 and 4 end with the modules of `sequence-ai` that implement them. Section 6 evaluates GS-WAN with real policies, network links and GPUs, and Section 7 lists the limits of our measurements, which cover simulated robots and GPUs in the United States. Prior work is acknowledged in the section where it applies.

2 Background: Remote Inference for Robot Policies

This section defines the terms used in the rest of the report. GS-WAN was built for our own serving platform, which we call *the platform*.

2.1 The policy as a black box

In general, a robot sends its GPU a continuous stream, the video of its cameras and the readings of its joints, and the GPU sends back *action chunks*. We treat the policy as a black box over this stream (Figure 2): whenever the robot needs new actions, at time t, the policy reads the stream up to t and returns a chunk,

$$A_t = \pi(I_{\le t},\, s_{\le t},\, \ell), \qquad A_t = (a_t,\, a_{t+1},\, \ldots,\, a_{t+H-1}), \tag{1}$$

where $I_{\le t}$ are the camera frames up to t, $s_{\le t}$ the robot's states, ℓ an instruction, and the chunk holds H actions, one for each of the next control steps. A policy may read only the newest frames and state, as many vision-language-action policies do, or a history of them, as video-action policies do; what happens inside π is the policy's own, and GS-WAN does not change it. How the stream travels between the robot and its GPU is the network layer's concern (Section 4). GS-WAN needs four things from a policy, all of which its author declares or already has: the call itself; the size of its input images, so that frames can be resized before they are sent; the timing of its chunks (Section 2.2); and, optionally, a batched form of the call and access to its PyTorch modules, for batching and CUDA-graph replay (Section 3). It reads nothing else from the policy and does not change its weights. With `serve()` in `sequence-ai` 0.63.0, a user's own server needs only the call, and the batched form stays optional.

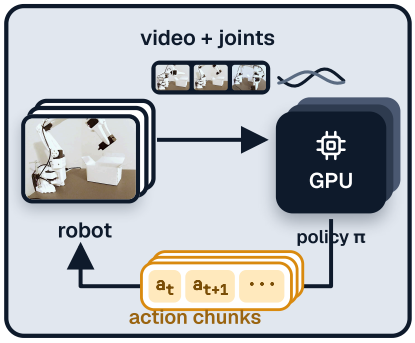


Figure 2 | A robot streams video and joint readings to its GPU, which sends back action chunks.

2.2 Action chunks and asynchronous execution

An action chunk is a matrix with one row of joint or end-effector targets for each of the next control steps; we call each row an *action*. A chunk typically holds tens of actions, executed at tens of hertz. The robot executes one chunk while the next is computed, as in SmolVLA's asynchronous inference [32] and real-time chunking from π_0 and $\pi_{0.5}$ [5]; how

the robot moves from one chunk to the next is left to such methods [5, 34, 29], which compose with GS-WAN, since it changes only how long a request takes.

Two parameters, declared by the policy's author, describe when the robot asks for the next chunk (Figure 3): the number of valid actions in a chunk, N, and the step k at which the robot requests the next one. With $k = N$ the program is *synchronous*: the robot finishes the chunk, sends its observation and stands still until the response arrives. With $k < N$ it is *asynchronous*: the robot keeps moving through the remaining $N - k$ actions, which at a control rate of h Hz last $(N - k)/h$ seconds; we call this time the *buffer*: how long the robot can keep moving on the actions it already has while the next chunk is on its way.

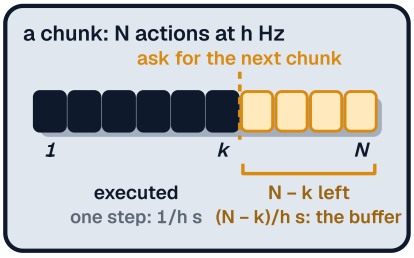


Figure 3 | A chunk of N actions at h Hz: at step k the robot asks for the next one, and the $N - k$ actions left keep it moving for $(N - k)/h$ seconds, its buffer.

2.3 The path of a request and its time budget

An inference *request* covers everything from the robot's decision to request a chunk until it holds the response. Its end-to-end latency is

$$t_{\text{request}} = t_{\text{encode}} + t_{\text{upload}} + t_{\text{queue}} + t_{\text{batch}} + t_{\text{process}} + t_{\text{inference}} + t_{\text{download}} + t_{\text{decode}}. \quad (2)$$

$t_{\text{inference}}$ is the model's forward pass, and t_{process} the server's own work around it: decoding the request, the policy's input and output transforms, and encoding the response. t_{queue} is the time a request waits for a model *replica* (one loaded instance of the model) that is busy with other robots, and t_{batch} is the time it is held so that it can share a forward pass with them. t_{upload} and t_{download} are the network transfers; together they take at least one *network delay*, the time a message takes to reach the GPU and come back, and more when the data do not fit in the connection's *congestion window*, the number of packets TCP may send before it must wait for the other end's acknowledgment. When the robot streams its frames continuously, the upload on a request's path shrinks to that of its newest frame. We report latencies by their median (p50) and by the time that only the slowest 5% of requests exceeded (the 95th percentile, p95), and we call everything except $t_{\text{inference}}$ the time *outside the model*; tables label it "Rest".

The observation behind GS-WAN's design is that *when a robot sends a request while part of its current chunk remains, the request does not have to be served as fast as possible; it has to be served before the robot runs out of actions.* Fleet schedulers on local networks already use this deadline [4, 1, 2], as serving systems in general batch requests within a latency objective [10, 12]; what a WAN adds is that each robot's own time outside the server, which differs between robots, comes out of the same deadline. The request's *budget* is what is left of the chunk being executed when the request is sent, at most the buffer $(N - k)/h$. Its *slack* is its spare time: how much longer the request could take and still arrive before the robot runs out of actions. Formally, the slack is the budget minus everything the request will spend before its response arrives (Figure 4):

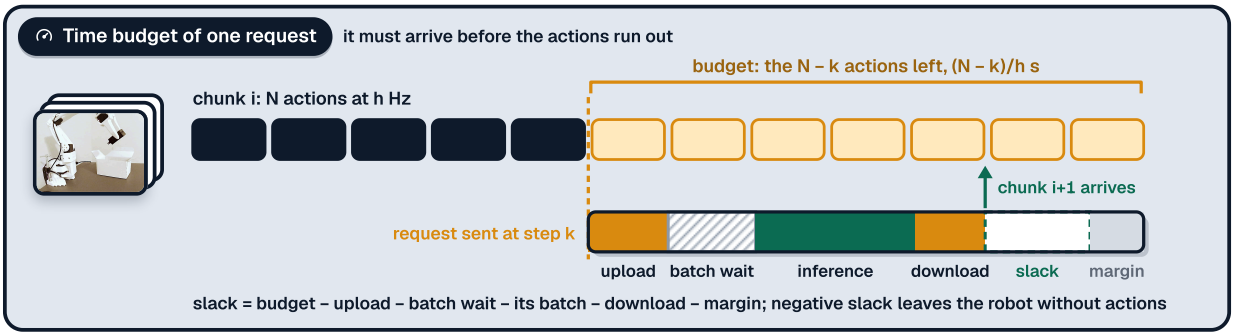


Figure 4 | The time budget of one request. Top, the robot executes chunk i, N actions at h Hz, and asks for the next chunk at step k; the $N - k$ actions left, $(N - k)/h$ seconds, are the request's budget. Below, the request spends that budget on the upload, the wait for a batch (zero for a robot that does not share its GPU), its batch's inference and the download; chunk $i+1$ arrives, and what is left before chunk i runs out, less a margin, is the request's slack (Equation 3). A negative slack leaves the robot without actions.

$$\text{slack} = \text{budget} - \text{time outside the server} - \text{wait for a batch} - \text{time of its batch} - \text{margin}. \quad (3)$$

The time outside the server is the robot's own network and client time, which varies across robots; the time of its batch is the measured server time of a batch of the size the request would join. A response arrives in time when the slack is

not negative, so a request may wait for a batch at most as long as the slack would stay non-negative. Batching across robots consumes slack (Section 3.5).

2.4 The scaling objective

For one GPU that runs one replica of a policy, scaling means serving as many robots as possible while every robot's slack stays non-negative. A replica that serves robots $r \in R$, each in batches of size b_r, must satisfy, for every robot,

$$o_r + w_r + S(b_r) + m \leq B_r, \tag{4}$$

where B_r is the robot's budget, o_r its time outside the server, w_r its wait for a batch, $S(b)$ the server time of a batch of size b, and m a margin; the number of robots a replica can accept under this constraint, and under a limit on how busy its GPU may be, is its number of *seats*: how many robots it can take on while every one of them still receives its actions in time (Section 3.6). Three quantities set the answer. A larger batch costs every robot in it more server time; a robot farther from the GPU has a larger o_r and so leaves less of its budget for batching; and the GPU must exist in a region near the robots, which on-demand capacity does not guarantee. GS-WAN treats the three together: the inference layer packs robots into batches within each robot's slack (Section 3), the network layer removes avoidable network delays from o_r (Section 4), and the routing layer places robots near GPUs that can actually be obtained (Section 5).

Each of the next three sections begins with what we measured at its layer. Each experiment states what it measures and what we expected, and holds everything fixed except the factors it varies; each policy keeps its own timing: its control rate, the actions it uses from each chunk and the step at which it asks for the next, as given with each result.

3 GS Inference: Sharing and Speeding Up the GPU

The inference layer runs on the GPU server, where most of a request's time is spent. This section first measures that time and how much of a GPU one robot uses, then describes how GS Inference shares the GPU and speeds up each pass.

3.1 Where a request's time goes

The baseline is one robot on a GPU it does not share, with the network taken out; what real places add to it is in Section 4.1.

The baseline. Table 1 gives each policy's cost on an unshared L4: the model's inference is almost all of a request's time in the server, and the server's own work around it and the handling of the request outside the server each add a few milliseconds. Inference does not depend on the network, so this cost is the reference for everything below: a real path or a shared GPU adds to it. On an A100, $\pi_{0.5}$'s inference took well under half as long as on the L4, while GR00T and SmolVLA took longer, because that machine's CPU was slower to launch their work on the GPU (Section 3.10).

	One robot on an L4, ms			A100, ms	
Policy	Server's work around the model	Model inference	Overhead outside the server	Model inference	Request size
$\pi_{0.5}$ DROID (JAX)	6.0	185.3	2.1	70.3	301 KB
GR00T N1.6 DROID	12.1	209.9	2.1	294.3	346 KB
SmolVLA base	2.6	252.3	4.6	363.3	1,843 KB

Table 1 | The baseline: where one request's time goes with one robot on an NVIDIA L4 that it does not share, every policy at $\pi_{0.5}$'s timing of one request every 533 ms; medians in ms over two runs of 60 s at each of five network delays from 0 to 200 ms, added by a relay and pooled, since none of these times changed with the network delay. *Server's work around the model*: decoding the request, the policy's own input and output transforms, and encoding the response. *Model inference*: the model's forward pass alone. The two together are about the time in the server. *Overhead outside the server*: what the time outside the server adds to the network delay itself, for packing, unpacking and passing on the request and its response. *Model inference on an A100*: the same forward pass on an NVIDIA A100-40GB, timed alone 40 times without a robot; GR00T and SmolVLA, which spend much of each call waiting for the CPU to launch their work on the GPU, were slower there because that machine's CPU was slower (Section 3.10). *Request size*: the uncompressed camera frames and joint readings the policy sends with each request.

3.2 GPU use with one robot

With one robot, most of a GPU sits idle. Between one robot's requests the GPU was busy only part of the time, and less still at each policy's own timing, because GR00T asks once a second and SmolVLA only after its whole

chunk (Table 2); within a call, too, the GPU was idle part of the time with GR00T and most of it with SmolVLA (Section 3.10). How many robots can share a GPU therefore depends on each policy's timing and on how idle the GPU is within a call (Sections 3.5 and 3.6).

	Measured, one request every 533 ms		Computed for the policy's own timing		
Policy	One call, ms	Kernel running	Time between requests, ms	Calls' share of the time	Kernel running
$\pi_{0.5}$ DROID	191.4	34.8%	533	36%	34.8%
GR00T N1.6 DROID	223.3	26.7%	1,000	22%	15%
SmolVLA base	255.3	11.2%	1,667 + latency	13%	3%

Table 2 | What one robot uses of an NVIDIA L4. The first two columns were measured in the runs of Table 1, where every policy sent one request every 533 ms; the last three are computed for each policy's own timing, not measured, except that $\pi_{0.5}$'s own timing is the one measured, so its kernel share is the measured value. *One call*: the server's median time for one request, almost all of it the model's call with its input and output transforms. *Kernel running*: the share of time in which any kernel was running, as NVIDIA's monitoring tool reports it; where computed, the calls' share times the share of a call in which the GPU was busy when calls ran back to back. *Time between requests*: one cycle of the policy's own timing; SmolVLA asks only after finishing its chunk, so its cycle also includes the wait for the response. *Calls' share of the time*: one call over the time between requests. A kernel runs for less than the calls' share because within a call the GPU also waits for the CPU, for SmolVLA most of all (Section 3.10).

3.3 Design

With one robot, the model takes most of a request's time and most of the GPU sits idle. Four rules apply to any policy of the form of Equation 1, whatever its model and GPU:

1) Share a GPU by answering several robots together in one pass of the model, which we call *batching*, in the batch sizes the policy supports, with one replica of the model per GPU (Sections 3.4 and 3.8).
2) Hold each request for a batch only as long as its robot can wait: what is left of the robot's current chunk of actions, minus its own time outside the server, the time of the batch and a margin. We call this time the request's *slack* (Section 3.5).
3) Before the robots connect, compute how many robots each GPU can accept and still answer on time, from the policy's declared timing, the measured time of each batch size and each robot's own network delay. A request can arrive just after another batch has started and must then wait for it, so the count is best found by simulating the robots' requests on a clock. A number of robots counts as on time when the share of late responses stays within what a robot tolerates; the largest such number that leaves the GPU some headroom is the GPU's *seats*, and a robot that finds no free seat goes to another GPU (Section 3.6).
4) Speed up inference only in ways that keep the outputs exactly the same (Section 3.10); faster paths that change the outputs are future work (Section 3.12).

GS Inference implements the first three rules with its holding rule and its count of how many robots fit, and the fourth with CUDA graphs, kept only where the outputs stay exactly the same; mechanism details are in Appendix B.1. How much each gains depends on the policy and the GPU: batching on how a policy's pass grows with the batch size, and replay on how much of the time the GPU waits for its host processor (Table 3). On the platform, the server side of GS-WAN runs in the *worker*, the server process on the GPU machine that holds the model replicas; the robot reaches it directly, not through the platform's gateway, and with every request reports how long its requests spend outside the worker and how much of its current chunk remained.

Policy	Batch size 1		Median time of one pass (ms), batch size 1 / 2 / 4 / 8 / 16	Requests/s, batch 1 / 16	Gain
	Kernel running	Power			
NVIDIA L4, power limit 72 W					
$\pi_{0.5}$ (DROID, JAX)	99.5%	97%	192.6 / 313.0 / 555.0 / 1,048.4 / 2,042.4	5.19 / 7.83	1.5×
GR00T N1.6 (DROID)	69.0%	95%	211.4 / 248.4 / 366.0 / 660.7 / 1,318.5	4.73 / 12.14	2.6×
SmolVLA (base)	20.0%	53%	261.6 / 274.3 / 297.7 / 356.7 / 501.1	3.82 / 31.93	8.4×
GR00T N1.6, CUDA graphs	93.5%	100%	162.5 / 226.2 / 368.9 / 673.1 / 1,340.1	6.15 / 11.94	1.9×
NVIDIA A100-40GB, power limit 400 W					
$\pi_{0.5}$ (DROID, JAX)	98.0%	67%	70.3 / 106.7 / 163.0 / 268.2 / 478.2	14.22 / 33.46	2.4×
GR00T N1.6 (DROID)	32.0%	17%	294.3 / 295.7 / 308.7 / 351.4 / 444.2	3.40 / 36.02	10.6×
SmolVLA (base)	26.0%	14%	363.3 / 367.8 / 371.7 / 377.1 / 403.4	2.75 / 39.67	14.4×
GR00T N1.6, CUDA graphs	65.0%	44%	97.0 at batch size 1, 142.1 at 4	10.31 at batch size 1	

Table 3 | Answering several robots together, on an L4 and on an A100. A pass took time linear in the batch size once one request kept the GPU fully in use: on the L4, $\pi_{0.5}$'s kernels ran all the time at nearly the card's full power, and each request added to its batch cost about two thirds of a lone pass, so answering robots together gained little. On the A100, $\pi_{0.5}$'s kernels also ran almost all the time, but at two thirds of the card's power, and from two requests on each added request cost a little over a third of a lone pass; a request added to GR00T's or SmolVLA's batch first used the time in which no kernel ran. The time of one pass of each policy's model, without its input and output transforms, at each batch size: 40 runs per batch size in eager execution, with the same script on both GPUs; no network or server is involved. *Kernel running*: the share of time in which any kernel was running, as nvidia-smi reports it; it does not show how much of the GPU each kernel uses. *Power*: the card's power draw as a share of its limit; near the limit, the card lowered its clock as the batch grew. *Median time of one pass*: in ms, for the whole batch. *Requests/s*: requests answered per second at batch sizes 1 and 16, the batch size divided by the median. *Gain*: the second over the first. The CUDA-graph rows replay GR00T's repeatedly called modules from CUDA graphs (Section 3.10); on the A100 they were run on a second machine of the same type, at batch sizes 1 and 4 only. SmolVLA's vision model could not be captured, and $\pi_{0.5}$ runs in JAX. The A100 machines' host processor was about half as fast per thread as the L4 machine's, which slows the eager PyTorch policies, GR00T and SmolVLA (Section 3.10). A repeat on a second L4 gave gains of 1.49×, 2.40× and 7.77×, within 10% of these. With the input and output transforms that serving also runs, batch size 1 took 197.9, 223.3 and 259.3 ms on the L4 and 79.8, 310.5 and 365.0 ms on the A100, for $\pi_{0.5}$, GR00T and SmolVLA.

3.4 Batching in declared sizes

The rest of a GPU's time could answer other robots, but robots answered one by one each wait for the others' passes. For a policy that declares that it can take several requests in one call, pending requests are combined into one pass, using only the batch sizes the policy declares. A batch goes to the model when it is full, or a short time after its first request if the model is free; a batch that raises an exception is retried with its requests answered one by one. We expected answering robots together to pay off as far as a single request leaves the GPU idle, and it did (Table 3): on both GPUs, SmolVLA, which left the GPU idle most of the time, gained the most throughput, and $\pi_{0.5}$, whose kernels ran almost all the time, gained the least. Every policy gained more on the A100 than on the L4: one request left more of the larger GPU's compute unused, and on that machine's slower host an eager PyTorch pass spent much of its time launching kernels, a cost that a whole batch shares. For $\pi_{0.5}$, the vision and prefix computation sets the ceiling: per request it costs almost as much in a batch as alone, while the action expert becomes nearly free. So answering robots together helps small or launch-bound policies most, and helps little a policy that already keeps the GPU fully in use. Throughput is not capacity: a larger batch takes longer, and every request in it must still arrive on time (Sections 3.5 and 3.6). Unlike in language-model serving, where cache memory limits a batch [20], these batches are limited by the GPU's compute and by how long each robot can wait.

3.5 How long each request waits

Holding every request for the same fixed time before answering can fail in both directions: too short, and robots that could share a pass miss each other; too long, and the response arrives after the robot has run out of actions. GS Inference's rule for how long to hold a request gives each request a countdown, its slack (Equation 3; Figure 5): the rest of its robot's current chunk, minus the time the robot's requests spend outside the server, minus the time the batch will take, minus a margin. Fleet servers on a local network schedule by each robot's remaining actions alone [4, 1]; subtracting each robot's own time outside the server is what lets robots at different distances share a batch over a WAN. The open batch waits for the other robots of its replica until the first countdown runs out, and a robot too far away to wait for a shared batch is answered first, in a separate batch that fits its own timing. We asked how many robots one card keeps on time when it answers them one by one and when it answers them together with GS Inference, on an L4 and on an A100, each policy at its own timing, and what each robot then costs (Figure 6 and Table 4). We expected

answering the robots together to add robots where the GPU is idle within a pass and the policy leaves time to wait, and it did. On an L4 it doubled the GR00T robots, whose GPU is idle a third of each pass and whose robots ask once a second, and added none for $\pi_{0.5}$, which keeps the L4 fully in use. On an A100 it kept twelve $\pi_{0.5}$ robots on time where answering them one by one kept six, which made a $\pi_{0.5}$ robot cheaper there than on an L4. For GR00T, the A100 kept more robots on time than the L4 only with CUDA graphs, because its host launched kernels more slowly (Section 3.10), and the L4 stayed the cheapest per robot. GS-WAN's clock, counting lateness alone, never counted a robot too many, and with GS Inference it found every count exactly.

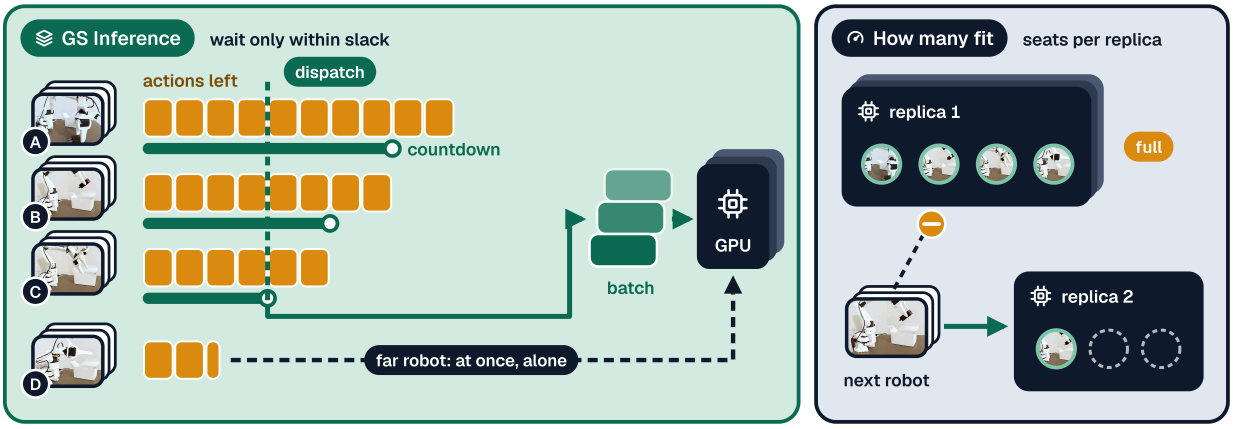


Figure 5 | GS Inference. Left, three robots have different amounts of their current chunk left (amber); the green bar under each is its request's countdown, its slack, the time the request may still be held. The batch is sent to the GPU when the first countdown runs out, here robot C's, so no robot runs out of actions, and a robot too far away to wait (D) is answered at once, alone. Right, before robots connect, it computes how many robots each replica can answer before they run out of actions, the replica's seats; when its seats are full, the next robot goes to another replica.

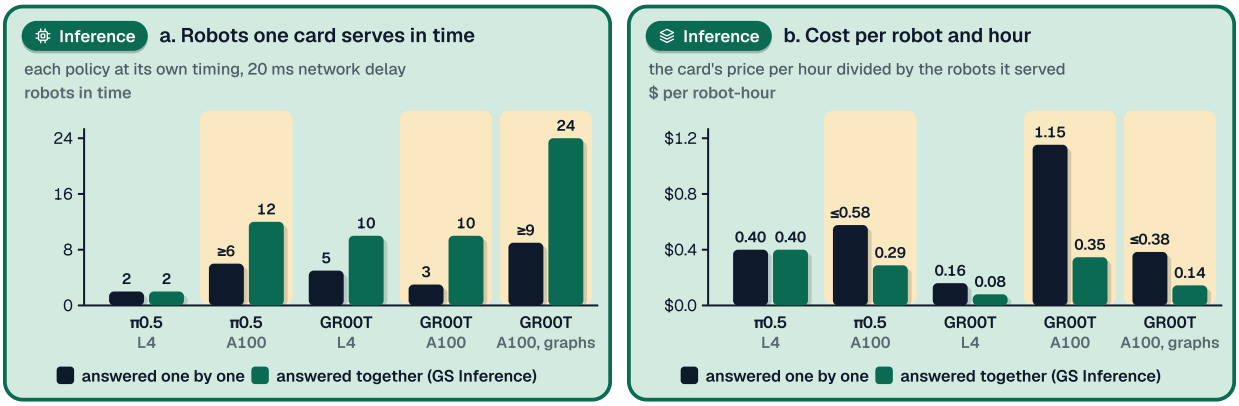


Figure 6 | Robots per card, on an NVIDIA L4 and an NVIDIA A100-40GB. (a) Robots one card kept on time, with fewer than 5% of responses late, answering them one by one or together with GS Inference, each policy at its own timing and 20 ms away; "≥" marks a count measured only from below. (b) The card's price per hour ($0.80 for the L4, $3.46 for the A100) divided by the robots it served, in dollars per robot and hour. On the A100, answering the robots together kept twelve $\pi_{0.5}$ robots on time where answering them one by one kept six, and with CUDA graphs it kept 24 GR00T robots; for GR00T, the L4 was the cheapest per robot.

		Robots kept on time		$ per robot and hour	
Policy	Card	One by one	Together	One by one	Together
$\pi_{0.5}$ DROID	L4	2	2	0.400	0.400
$\pi_{0.5}$ DROID	A100-40GB	$\geq$6	12	$\leq$0.577	0.288
GR00T N1.6, eager	L4	5	10	0.160	0.080
GR00T N1.6, eager	A100-40GB	3	10	1.153	0.346
GR00T N1.6, CUDA graphs	A100-40GB	$\geq$9	24	$\leq$0.384	0.144

Table 4 | Robots per card and what each costs. How many robots one card kept on time, answering them one by one or together with GS Inference, each policy at its own timing ($\pi_{0.5}$: a 467 ms budget, a request every 533 ms; GR00T N1.6: 1,133 ms, every 1,000 ms). The robots ran on the GPU's machine behind a relay that adds 20 ms of network delay; a number of robots counts as on time when fewer than 5% of responses arrived after the robot had run out of actions, in each of two 60 s runs. *Eager*: PyTorch's default execution; *CUDA graphs*: GR00T's repeated modules replayed (Section 3.10). *$ per robot and hour*: the card's price per hour ($0.80 for the L4, $3.46 for the A100-40GB) divided by the robots. GS-WAN's clock, counting lateness alone, proposed each count from the card's measured batch times, and the GPU then ran that count and one robot more; "$\geq$": one more robot was also on time and more were not run (seven $\pi_{0.5}$ robots answered one by one were late on the stand-in of Table 5). With the machine's CPU and memory, the GR00T robot-hour on the L4 is $0.122.

A robot farther from its GPU can wait less, so it leaves less room to answer robots together. We measured this on a stand-in server that sleeps for the A100's measured $\pi_{0.5}$ batch times instead of running the model (Table 5). With every robot 20 ms away, GS Inference kept eleven robots on time against six answered one by one; with a third of the robots each at 20, 100 and 200 ms, it kept eight, while answering them one by one still kept six. GS Inference needs no hold time to be chosen: each request waits as long as its own robot can spare.

	Robots kept on time	
How the server answers	All 20 ms away	A third each 20, 100, 200 ms away
Answered one by one	6	6
GS Inference	11	8

Table 5 | Robots one replica kept on time, with the robots near or far. Robots kept on time, with fewer than 5% of responses arriving after the robot had run out of actions in each of two 60 s runs, answered one by one or together with GS Inference. The robots were all 20 ms of network delay away, or a third of them each at 20, 100 and 200 ms. A stand-in server on a CPU sleeps for the A100's measured $\pi_{0.5}$ batch times instead of running the model, at $\pi_{0.5}$'s own timing (a 467 ms budget, a request every 533 ms). On the A100 itself, GS Inference kept twelve robots on time (Table 4).

3.6 How many robots fit

The same quantities answer how many robots one replica can accept before its responses arrive late; accepting robots until requests queue up finds the answer only by failing. Each robot fits the largest declared batch size whose time, plus that robot's own time outside the server and a margin, is within its buffer, the time its remaining actions last after it sends the request. Because robots send on their own clocks, a request can also arrive while another batch runs and wait for it, which no formula captures. GS Inference therefore runs two, three and more robots on a simulated clock, as its server would answer them, from many starting points spread over every way their requests can fall in a cycle. A number of robots is within capacity when at least 95% of their responses arrive before their buffers run out and, from every starting point, their batches fill at most 0.9 of a cycle; the first number that fails ends the count. The largest number within capacity is the replica's *seats*, the robots it accepts; earlier schemes that decide how many robots a server takes have no network term [2, 16]. Robots are accepted nearest first, so the farthest robot is the first one a replica cannot accept, and the platform moves it to another replica. We checked the seats on an L4 at each policy's own timing and three network delays (Table 6). Every accepted robot was on time at every network delay, and a farther robot left $\pi_{0.5}$ fewer seats, as the shared deadline predicts.

Policy	Network delay, ms	Seats	Late with the seats filled
$\pi_{0.5}$ DROID	20	2	0% / 0%
$\pi_{0.5}$ DROID	50	2	0% / 0%
$\pi_{0.5}$ DROID	100	1	0% / 0%
GR00T N1.6 DROID	20	7	0% / 0%
GR00T N1.6 DROID	50	7	0% / 0%
GR00T N1.6 DROID	100	7	0% / 0%

Table 6 | Checking the seats on one L4. The seats GS-WAN computed for each policy at its own timing ($\pi_{0.5}$: a 467 ms budget, a request every 533 ms; GR00T N1.6: 1,133 ms, every 1,000 ms), and the share of responses that arrived after the robot had run out of actions with the seats filled, in two runs of 60 s with different random starts. *Network delay*: added by a relay between the robots and the server.

When the seats were computed with every robot at the farthest robot's network delay, one distant robot lowered the batch size, and thus the seats, for every nearby one; fitting each robot by its own network delay keeps a distant robot from reducing the capacity of the others.

3.7 Against open-source batching servers

General-purpose serving systems also answer requests together: Ray Serve holds each request up to a set time, and NVIDIA Triton's dynamic batcher up to its queue delay. Neither knows how long a robot can wait, and by default both accept every robot that connects. We put each, at its default wait and at 2 and 50 ms, in place of GS Inference in front of a stand-in that replays the A100's measured $\pi_{0.5}$ batch times (Figure 7). With sixteen robots asking, more than one A100 can answer on time, every configuration of both accepted all sixteen and kept none on time, with more than 80% of their responses late, while GS Inference accepted eight, kept all eight on time and sent the others elsewhere; on the A100 itself with the real model, Ray Serve at its default was late on 98% of responses and GS Inference again kept its eight on time. With the robots at 20, 100 and 200 ms and none turned away, GS Inference kept ten robots on time where Ray Serve kept at most eight at any of its waits.

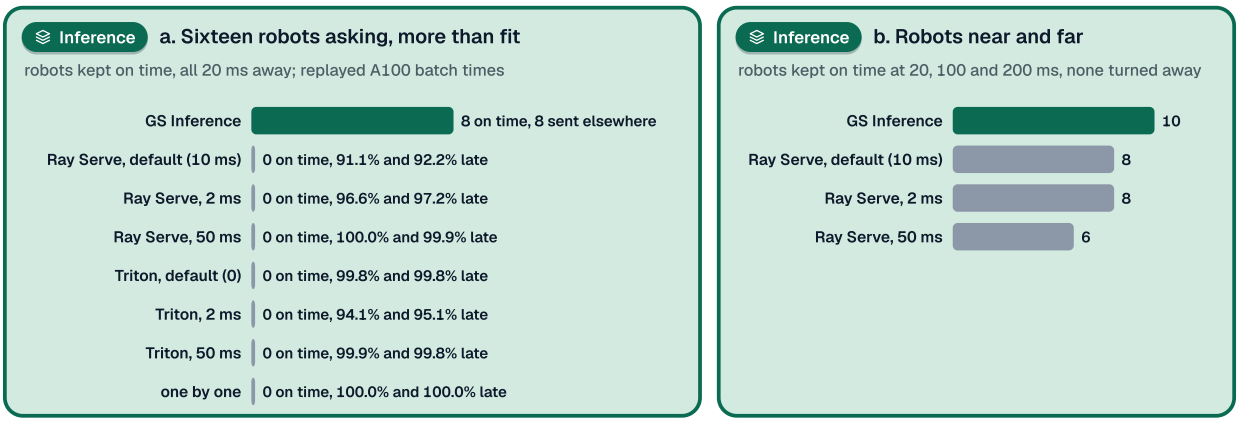


Figure 7 | GS Inference against open-source batching servers. Each server answers $\pi_{0.5}$ robots at its own timing in front of a stand-in that replays the batch times measured on an A100, two runs of 60 s per setting. (a) Sixteen robots asking, all 20 ms away: the robots kept on time, and for the others the share of late responses in each run; GS Inference sends the robots beyond its seats elsewhere, Ray Serve and Triton accept every robot. (b) Robots a third each at 20, 100 and 200 ms, every robot accepted: the most robots each server kept on time, with fewer than 5% of responses late in both runs.

3.8 One replica per GPU

A forward pass in the server's own process blocks its event loop, and a model that crashes brings the server down. The `replicas` component runs each replica in its own process bound to one GPU, dispatches each call to an idle replica, and replaces a process that died. We compared one replica that answers two robots together with two replicas that answer one robot each, on one L4 at each policy's own timing. Two replicas held two copies of the model and competed for the GPU whenever their robots' requests fell close together; one replica held one copy and put both

robots' requests in one pass. Answering robots together in one replica spends the slack that robots leave, which raises how many robots a GPU keeps on time (Table 4). GS-WAN runs one replica per GPU and answers its robots together, because what it guarantees is each robot's deadline.

3.9 A planner beside the policy

Some robot programs pair a fast action policy, often called System 1, with a slower model that decides what the robot should do next, System 2: for example a vision-language model that turns the camera view and the task into the next short instruction for the policy. The planner can take several times as long as an action request, so it must never sit on the action's path. In `sequence-ai`, a policy declares its planner as a method of the same class as its action call, `@seq.plan(every_s=..., wake_on=...)` beside `@seq.infer`; the hosted platform runs each robot's plans on a clock, or sooner when an observation the planner names arrives, in the background of its actions, one at a time per robot, and each action call reads the newest plan instead of waiting for one. In one program run on the platform, a vision-language model planned every 2 s for a $\pi_{0.5}$ policy on one L4. Its plans took 716 ms at the median, longer than the 467 ms the robot's actions last, yet the policy's 362 action requests took 230 ms at the median and none failed, because none of them waited for a plan (Figure 8).

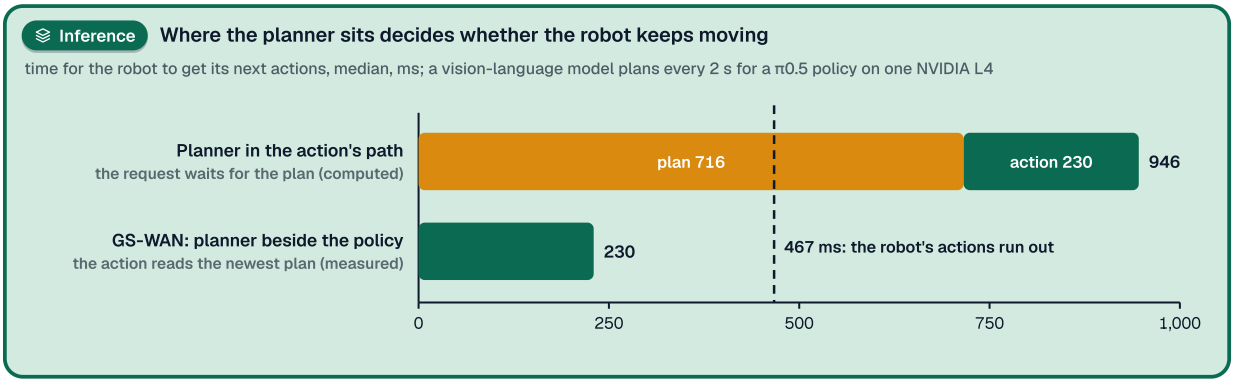


Figure 8 | Where the planner sits. One robot on one NVIDIA L4, with a vision-language model planning the next instruction every 2 s (System 2) for a $\pi_{0.5}$ policy (System 1); median times in ms. With GS-WAN the planner runs beside the policy and each action request reads the newest plan, measured over 362 requests. Placed in the action's path, a request would also wait for the plan, computed here as the two medians added, and its actions would arrive after the robot's own had run out at 467 ms, the buffer its template declares.

3.10 CUDA graphs with exact outputs

The denoiser of a diffusion or flow policy is called several times per inference with the same shapes, and in eager PyTorch each call launches its kernels one at a time, thousands per inference of GR00T, so the GPU spends much of each call waiting for the next launch. CUDA graphs, introduced in CUDA 10 [26], remove this overhead by replaying recorded kernels with one launch; VLA inference engines use them [24, 33, 21], as does `torch.compile` [3] in its reduce-overhead mode, and GS-WAN does not introduce them. A compiler that also fuses operations reorders the arithmetic, which changes the outputs, and a policy is validated on its outputs. The component adds two things to standard capture: it chooses the module to capture automatically, from a census of the submodules called repeatedly in each inference, and it keeps a graph for a shape only if the graph's output is exactly the same as the eager output, with the capture run against a clone of the random number generator's state; the module stays in place. On the hosted platform, graphs are tried automatically on each GPU type and kept only when a policy's answers stay the same bit for bit and its requests get at least 10% faster. We expected exact replay to remove this overhead without changing the outputs. On GR00T N1.7 on an L4 it made a call about one and a half times faster (Figure 9), with exactly the same outputs on new inputs, a new prompt and a larger batch, and capture took a fraction of a second per shape and no extra GPU memory. `torch.compile` reached about the same speed only after a long compilation, and both of its modes changed the outputs. Across three policies on the L4 (Table 3), replay helped only where the GPU waited for the host: GR00T gained at small batch sizes and not once a batch kept the GPU busy, SmolVLA's vision model could not be captured, and $\pi_{0.5}$ runs in JAX, which already compiles the whole computation. NVIDIA's own deployment of GR00T is faster still, by accepting outputs that are close to the eager ones rather than the same [27]. A profile of one call on the L4 showed where the saving comes from: replay cut GR00T's call while its kernels' own time stayed about the same, so what it removes is the host's time to launch them, and a policy whose kernels already fill the call, such as

$\pi_{0.5}$ in JAX, has nothing to gain. The host therefore matters as much as the GPU for such a policy. On an A100 whose host processor was about half as fast per thread as the L4 machine's, eager GR00T ran slower than on the L4; replayed from CUDA graphs, the same call ran three times faster than eager on that A100, with exactly the same outputs, and faster than on the L4 (Table 3). A larger GPU helps a launch-bound policy only once its launches are removed.

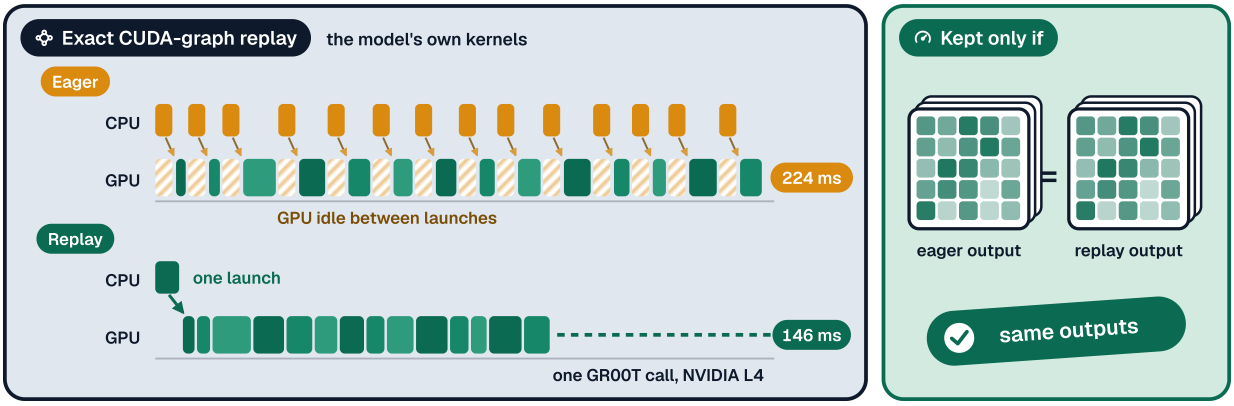


Figure 9 | Replaying a module's own kernels from a CUDA graph. Top, in eager execution the CPU launches the denoiser's kernels one at a time, and the GPU waits between them. Bottom, the same kernels, recorded once, are replayed from a CUDA graph with one launch. Right, a graph is kept only if its output is exactly the same as the eager output. On one GR00T N1.7 inference call on an L4, the replay took the call from 224 to 146 ms; on an A100 whose host launched kernels more slowly, it took GR00T N1.6's model call from 296 to 97 ms, measured on the same machine. CUDA graphs are a standard CUDA feature; GS-WAN adds the choice of module and the check that the outputs are exactly the same.

3.11 Using the inference layer

GS-WAN's inference and network components are released in the open-source Python package `sequence-ai`, under Apache-2.0, the inference components in `sequence_ai.engine` and the network components in `sequence_ai.network`; the same package is the client a robot uses to reach the platform's hosted service. This section and Section 4.9 show how to use them; Table 17 in Appendix A lists every component with its module and where this report measures it. Its routing components run in the hosted service, GPU Routing, and are described but not released. The package requires Python 3.9 or later and installs httpx and Pillow with it, and on Python 3.10 or later also pydantic and `sequence-base`; one install also brings uvicorn, numpy and zstandard for the GPU server. The inference layer is `sequence_ai.engine`: `slack` holds GS Inference's holding rule and its count of how many robots fit, `batcher` combines requests into one pass in the batch sizes a policy declares, `replicas` runs one replica per GPU, and `cuda_graphs` the exact replay. On the GPU server they wrap the model the user already has, and the policy's declared timing gives each replica's seats. Before relying on a computed number of seats, a user can check it with `examples/fleet_on_a_clock.py`, which runs the robots' requests on a clock with the policy's own batch times and prints, for each number of robots, the share of responses on time, the starting points with a late response and the load (Section 3.6). From version 0.63.0, `serve(policy)` runs a policy as a server in one call, answering robots together when it is given batch sizes, and `engine.openpi.batched` gives an openpi policy in JAX a batched call.

```
from sequence_ai.engine import cuda_graphs, slack
from sequence_ai.engine.batcher import Batcher
cuda_graphs.install(model, ["action_head.denoiser"])  # kept only if exactly the same
batcher = Batcher(infer_a_batch, sizes=[1, 2, 4, 8])  # several robots in one pass
action = await batcher(observation)                   # in each request's handler

# The policy's declared timing gives each replica's seats
play = slack.playback(hz=20, valid_len=15, replan_at=8)
seating = slack.Seating(table, play["buffer_s"] * 1000, sizes=[1, 2, 4],
                        cycle_ms=(play["play_s"] - play["buffer_s"]) * 1000)
seats, over = seating.seat([("near", 30.0), ("far", 200.0)])  # (robot, delay in ms)
```

A policy that runs on the hosted platform declares a planner beside its action call with the `seq` decorators of the same package; the platform runs the planner in the background, and the action call never waits for it (Section 3.9).

```
import seq

@seq.policy(name="my-policy", gpu="L4", max_gpus=1, hz=15,
            action_chunk_valid_len=15, action_chunk_replan_at=8,
            models=["<planner model>"])
class MyPolicy:
    @seq.plan(every_s=2)       # in the background, one plan at a time
    async def plan(self, robot):
        newest = robot.state["seq.observations"][-1]
        return await ask_planner(newest)  # the next instruction, or None

    @seq.concurrent(sizes=[1, 2, 4, 8, 16])  # robots answered together
    @seq.infer                 # never waits for a plan
    def infer(self, obs, *, seed, episode):
        steps = [newest_plan(e) for e in episode]  # each robot's own
        return run_policy(obs, steps)
```

3.12 Future work

Techniques that trade a policy's exact outputs for speed, such as quantization, fused kernels and compilers like TensorRT, are future work: each changes the outputs, so a policy served with one would have to be validated again, and GS-WAN would offer them as options that a user chooses.

4 GS Network: Keeping the Connection Fast

The network layer carries the stream of Section 2.1 between a robot and its GPU over a wide-area network (WAN), a path that robot middleware leaves to purpose-built tools [25], rather than over the local networks on which policy servers have been measured [17]. In general the robot streams its frames and states, and the GPU returns a chunk whenever the robot needs one. Today's policy clients, such as openpi's, implement the simplest case, one request per chunk: the robot captures its newest *observation*, its current frames and state, uploads it, and executes its current chunk while it waits for the next. Networks and servers are not tuned for this periodic, upload-heavy traffic. This section first measures what the network adds to a request, then describes how GS Network removes the extra delays.

4.1 Where a request's time goes on the network

At real places. The baseline of Section 3.1 added the network delay with a relay, which only adds delay; a real path also changes what TCP does while the robot pauses between requests (Section 4.3). We put one robot at a time in six places, all using the same L4 in Virginia without routing (Table 7). The time in the server stayed at its baseline, so only the time outside it changed. We expected that time to be about one network delay, and often it was not: the robot's connection sends nothing while it pauses, Linux shrinks the window of an idle connection, and an upload that no longer fit the shrunken window took several network delays (Table 8). With the server's pings during the pause, the upload took about one network delay from every place but Virginia, where a whole network delay is only a few milliseconds.

Robot in	Network only	$\pi_{0.5}$, budget 467 ms: Total time, median	$\pi_{0.5}$, budget 467 ms: Total time, slowest 5%	$\pi_{0.5}$, budget 467 ms: Late	GR00T N1.6, budget 1,133 ms: Total time, median	GR00T N1.6, budget 1,133 ms: Total time, slowest 5%
Virginia (near the GPU)	3	204 / 199	207 / 201	0 / 0	240 / 230	242 / 233
Iowa	30	291 / 230	292 / 234	0 / 0	340 / 255	342 / 258
Oregon	75	285 / 284	292 / 295	0 / 0	466 / 313	524 / 325
London	77	360 / 281	368 / 287	0 / 0	471 / 319	531 / 350
Tokyo	162	378 / 378	390 / 388	0 / 0	706 / 414	863 / 428
Singapore	222	460 / 461	662 / 644	**44.5% / 36.6%**	666 / 489	694 / 502

Table 7 | Wired robots. One robot at a time at each place, all using the same L4 in Virginia with no routing, so the robots in Tokyo and Singapore crossed the Pacific; each robot was a cloud machine on a wired network with the GPU to itself, two runs of 60 s per setting. Each cell gives plain TCP / GS-WAN's server pings, small messages the server sends while the robot pauses so that the connection keeps its sending window (Section 4.4). *Network only*: the time a message takes to reach the GPU's machine and come back, nothing else (the median TCP handshake), in ms. *Total time*: from sending a request to receiving its actions, inference included (the time in the server, almost all of it inference, was about 195 ms for $\pi_{0.5}$ and 221 ms for GR00T in these runs), in ms: the median, and the time that only the slowest 5% of requests exceeded. *Late*: the share of $\pi_{0.5}$'s requests whose total time exceeded its budget; the median can be under the budget while the slowest requests are over it, as from Singapore. No GR00T request was late.

Robot in	Time, ms: Robot's pause	Time, ms: Connection idle	Time, ms: Resend timeout	Window, packets: Before the pause	Window, packets: After the pause	Upload fits	Outside the server, in network delays: Plain TCP	Outside the server, in network delays: With pings
$\pi_{0.5}$: an upload of 301 KB, 209 packets								
Virginia	330	527	203	286	72	0%	3.44	1.71
Iowa	243	469	232	286	72	0%	3.18	1.15
Oregon	248	518	276	430	215	100%	1.20	1.18
London	173	444	277	279	140	26%	2.15	1.12
Tokyo	155	511	362	430	215	97%	1.13	1.13
Singapore	74	487	423	410	215	59%	1.19	1.20
GR00T N1.6: an upload of 346 KB, 239 packets								
Virginia	760	986	203	262	16	0%	5.44	1.67
Iowa	660	911	230	281	35	0%	3.97	1.14
Oregon	533	830	276	285	40	0%	3.26	1.23
London	530	827	277	285	41	0%	3.23	1.25
Tokyo	295	675	364	214	73	0%	2.99	1.18
Singapore	334	777	426	491	246	68%	2.00	1.20

Table 8 | Why the upload after a pause took several network delays from some places: the wired robots of Table 7. Every column but the last is with plain TCP, and each is its own median over a place's requests, so the columns need not agree exactly with one another. TCP lets a connection have only a limited number of packets in flight, its *window*, before it must wait for an acknowledgment, so an upload larger than the window takes more than one network delay. *Robot's pause*: from a response to the robot's next request. *Connection idle*: how long the robot's connection had sent no data when the request began, which adds the wait for the previous response to the pause. *Resend timeout*: how long Linux waits for an acknowledgment before it sends again (the retransmission timeout, RTO), at least the network delay plus 200 ms. Linux halves an idle connection's window once for each whole timeout it stays idle, and applies the halving only when the connection next sends. *Window before the pause*: the last value sampled, every 0.2 s, before the request. *Window after the pause*: computed from the window before, the idle time and the timeout by Linux's rule, since the samples cannot show a halving made at the moment of sending. *Upload fits*: the share of requests whose window after the pause held the whole upload. *Outside the server*: the request's time outside the server divided by its network-only time (Table 7), with plain TCP and with GS-WAN's server pings; 1.0 means one network delay. With pings, the robot's replies to them count as sending, so the connection was idle for less than one timeout on most requests and almost every upload fit. One case is not explained: from Singapore, GR00T's window held most of its uploads, yet its time outside the server was 2.0 network delays.

$\pi_{0.5}$ met its budget everywhere except Singapore, where the network delay and inference alone nearly fill it, with or without pings; GR00T, whose budget is more than a second, was in time everywhere.

One upload, without a model. We expected an upload after a pause to pay one extra network delay, because Linux shrinks the congestion window of a connection that has been idle (Section 4.3). From the same cities, with a server that runs no model, it depended on the window the connection had built before the pause: an upload that still fit the

shrunken window took about one network delay, a larger one two or more, and for uploads up to 300 KB the server's pings brought the cloud machines' paths back to about one (Table 9).

		Until the upload was acknowledged, ms (in network delays)				
		30 KB	300 KB		2 MB	
Robot in	Network only	Plain TCP	Plain TCP	With pings	Plain TCP	With pings
Virginia (near the server)	1.0	1.2 (1.17)	2.0 (2.04)	2.0 (1.98)	16.6 (16.84)	16.6 (16.82)
Iowa	24.5	30.6 (1.25)	27.4 (1.12)	26.7 (1.09)	46.7 (1.90)	43.5 (1.77)
Oregon	60.6	68.5 (1.13)	99.8 (1.65)	74.5 (1.23)	104.6 (1.73)	80.9 (1.34)
London	98.0	102.9 (1.05)	107.1 (1.09)	106.3 (1.08)	218.3 (2.23)	125.5 (1.28)
Tokyo	164.4	172.3 (1.05)	200.6 (1.22)	195.9 (1.19)	180.7 (1.10)	182.7 (1.11)
Singapore	220.3	229.0 (1.04)	268.2 (1.22)	246.6 (1.12)	249.6 (1.13)	247.3 (1.12)

Table 9 | One upload after a pause, over real paths, with no inference. A small cloud machine at each place uploaded to one server in Virginia that runs no model and only acknowledges each upload; each upload began 333 ms after the previous one was acknowledged, about the pause of a $\pi_{0.5}$ robot near its GPU. 40 uploads per cell, in the daytime, with the link-measuring tool released with `sequence-ai`. Each cell gives the median time from the start of an upload until the server acknowledged it, in ms, and in brackets the same in network delays: each upload divided by the network delay measured during its own 20 uploads. *Network only*: the time a message takes to reach the server and come back, the median TCP handshake, in ms. *With pings*: with GS-WAN's server pings during the pause. Linux's default settings. A 30 KB upload took about one network delay with plain TCP from every place, so pings are shown only for the larger ones. Near the server a network delay is about 1 ms, so the time there is that of sending the bytes, and its bracketed values are large. Whether a larger upload took about one network delay or two depended on the window that the uploads before it had built (Table 8). The server here was a small VM in Virginia rather than the GPU machine of Table 7, so the network-only times differ from that table's; London's route changed during the run.

In a closed loop, the idle time, and so the window after the pause, follows from the policy's timing (Table 8). A stream does not pay this network delay, since its connection does not go idle, but it needs an uplink fast enough for its frames (Appendices B.2.2 and B.2.8). The size of an upload follows from its frames. Encoded, a camera sends many times fewer bytes, as video far fewer than as JPEG, and on a slow uplink that decides how much of a robot's time budget one frame uses (Figure 10).

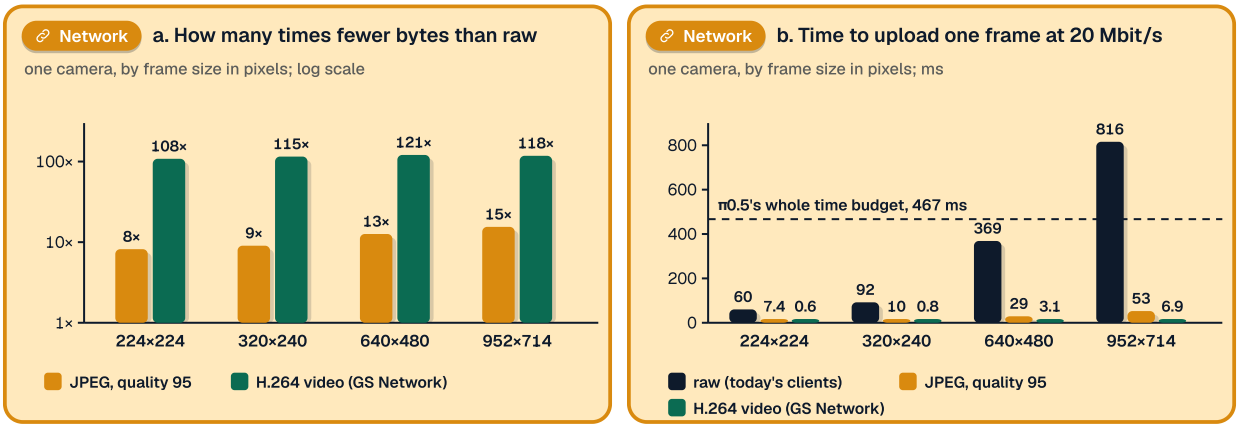


Figure 10 | What encoding a camera's frames saves. GS Network's client streams camera frames as low-latency H.264 video, which needs about a hundredth of the raw bytes, and with `sequence-ai` 0.63.0 a robot connected to a user's own `serve()` sends JPEG at quality 95 by default. (a) How many times fewer bytes one camera sends than raw RGB frames: JPEG at quality 95, frame by frame, and video at 15 frames per second against raw frames at the same rate; log scale. (b) The time one frame takes to upload on a 20 Mbit/s uplink, the slow link of Table 13, in ms, for video its average frame; dashed, $\pi_{0.5}$'s whole time budget.

4.2 Design

Three rules apply to the connection between a robot and its GPU:

1) Keep each request to one network delay: a connection that pauses between requests must not lose its congestion window, the amount of data TCP may send before it waits for an acknowledgment, and a response must not be split into pieces that wait for each other (Sections 4.3 to 4.6).
2) Send frames at the size the policy uses, and compress them only when that saves time on the robot's own uplink (Section 4.6).
3) Where the client can be changed, stream the frames as they are produced, so that only the newest frame is on a request's path, and send camera frames as video, which shrinks the stream at the cost of changing the pixels the policy sees (Section 4.6).

GS-WAN's network layer, *GS Network*, implements the first two rules for one request per chunk, on both ends of the connection and for existing clients: it keeps the request and response those clients use and removes the extra network delays that TCP and servers add to them. The client in `sequence-ai` that reaches the hosted service also follows the third: it streams each observation as it is produced, with camera frames as low-latency H.264 video (Section 4.6). The components for a user's own server do not stream, and we have evaluated neither what the codec changes in a policy's actions nor QUIC [15], a transport over UDP. Mechanism details and the tools that measure a link are in Appendix B.2.

4.3 The congestion window after a pause

On Linux, a connection that has sent nothing for longer than its retransmission timeout (RTO), the time TCP waits for an acknowledgment before it resends data, which is at least the network delay plus 200 ms, has its congestion window reduced before it sends again: the window is halved once for every RTO of idle time, down to an initial window of 10 packets, as RFC 2861 specifies [13]. If the reduced window cannot hold the next observation, the observation is sent in two parts, the second only after the first is acknowledged, and the request pays one extra network delay (Figure 11). The reduction is specified behavior (RFC 7661 [11] later obsoleted RFC 2861), and an engineering report traced slow messages to it and removed it with a heartbeat [9]. What this report adds is when it affects a robot's requests, and two mitigations that need no administrator rights on the robot: a heartbeat sent by the server, or BBR. Four mitigations remove it, and they differ in who can apply them: setting `net.ipv4.tcp_slow_start_after_idle` to 0, which needs administrator rights on the robot; BBR [7], a congestion control that paces by the measured bottleneck bandwidth and network delay, on the connection; a small request from the robot about every 100 ms while it is otherwise idle, which keeps the time since the last send below the RTO; and a ping from the server at the same interval (Section 4.4). `window` returns the socket options that select BBR where the machine allows it, and its `Warm` helper sends the robot's pings. We expected an upload after a pause to take two network delays by default and one with any of the four mitigations. Between two cloud machines without loss that is what happened, at every network delay but the longest (Table 10). A second pair of machines across the country, on two kernel versions, repeated it in every setting, and WebSocket and Zenoh requests, which took several network delays unmodified, took about one with any of the mitigations. So a robot on Linux that pauses between requests pays one extra network delay on each, and any of the four mitigations removes it. At the longest network delay the pause spans only one timeout, the window is halved once, and the halved window still holds the small upload, so there is nothing to remove.

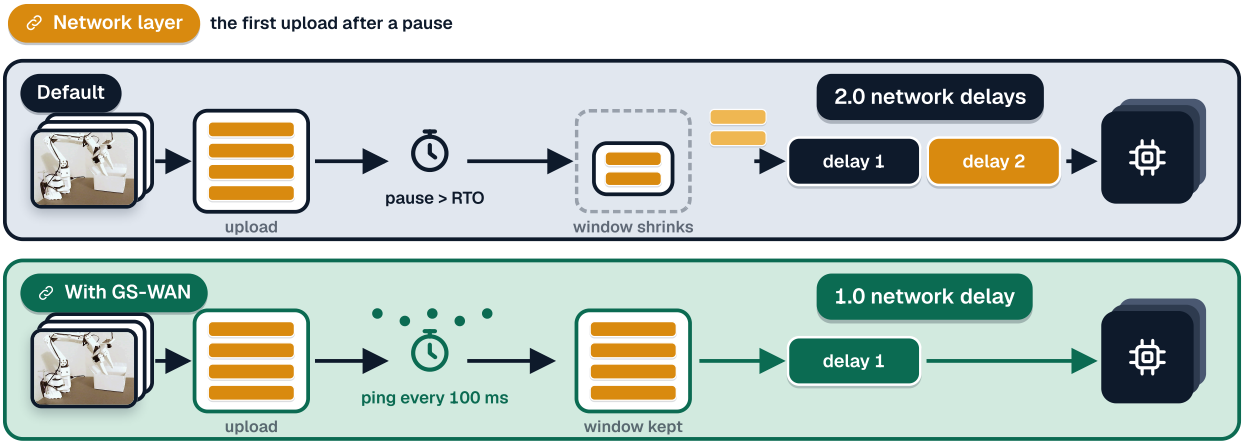


Figure 11 | The first upload after a pause. Top, Linux by default: the robot uploads, then pauses for longer than the connection's retransmission timeout (RTO); the kernel shrinks the congestion window, the next upload no longer fits, and it takes two network delays. Bottom, with GS Network: pings during the pause, or BBR, keep the window open, and the next upload takes one. Between two cloud VMs without loss, a 24 KB upload 0.53 s after each response took 2.0 network delays by default at every network delay up to 101 ms, and 1.0 with any of the four mitigations (Table 10).

Network delay	Default	Server pings	Client sends while idle	BBR on the client socket	Kernel option at 0
12.4 ms	**1.99 (40)**	1.01 (0)	1.01 (0)	1.01 (0)	1.01 (0)
21.4 ms	**2.00 (40)**	1.01 (0)	1.01 (0)	1.00 (0)	1.01 (0)
40.5 ms	**1.99 (40)**	1.00 (0)	1.00 (0)	1.00 (0)	1.00 (0)
65.5 ms	**2.00 (40)**	1.00 (0)	1.00 (0)	1.01 (0)	1.00 (0)
81.3 ms	**2.00 (40)**	1.00 (0)	1.00 (0)	1.00 (0)	1.00 (0)
100.7 ms	**2.00 (40)**	1.00 (0)	1.00 (0)	1.00 (0)	1.00 (0)
242.2 ms	1.00 (0)	1.00 (0)	1.00 (0)	1.01 (0)	1.00 (0)

Table 10 | An upload after a pause, by default and with each of the four mitigations. Between two cloud VMs with no loss, the client uploads 24 KB 0.53 s after each reply, 40 times per setting, with the measurement tool released in `sequence-ai`; no model runs, and the server replies with a few bytes. *Network delay*: the median time of an empty request and its reply on the same connection without a pause, in ms. Each cell: the median time from sending the upload to receiving the reply, in network delays, and in brackets how many of the 40 uploads took more than 1.5 network delays. *Default*: Linux's defaults. *Server pings*: the server pings the client every 100 ms. *Client sends while idle*: the client sends a few bytes every 100 ms while it has nothing else to send. *BBR*: the client's socket uses BBR congestion control. *Kernel option at 0*: `tcp_slow_start_after_idle` set to 0 on the client. At 242 ms the pause spans a single timeout, and the halved window still holds the upload.

4.4 Server-side pings

The mitigations above are applied on the robot, and a robot's software is often the hardest part to change: it may be someone else's client library, or firmware. A WebSocket library answers a ping with a pong automatically, so if the server pings about every 100 ms, the robot sends a few bytes in every interval and its kernel never considers the connection idle; `ping` returns the options that existing servers already accept. Server pings brought every upload after a pause back to about one network delay between two cloud machines (Table 10). Without them, a larger upload paid more network delays the longer the pause, since each timeout halves the window again, and this held in two separate measurement periods. In closed loops from six places, the pings brought every robot's time outside the server to about one network delay from every place but Virginia, where one is only a few milliseconds, and they mattered most where the halved window no longer held the upload (Tables 7 and 9). A server can remove the robot's extra network delay without any change on the robot, with an interval of 100 ms that leaves room for one lost ping within the RTO.

4.5 Existing clients

An HTTP/1.1 client that starts a request while a ping is still waiting for its answer opens a second connection for it, with a new handshake and a congestion window that starts small again, which is what the ping is meant to prevent.

`connect(url)` opens one connection, sets the socket options of Section 4.3 on it, and never runs a ping and a request at the same time. `openpi.keep(policy)`, GS Network's adapter for openpi's client, adds BBR, TCP keepalive and pings while the robot is idle to the connection that openpi's client opens, and `lerobot.keep_alive_file(path)` writes a setting of Zenoh, the publish-subscribe transport that LeRobot's remote inference [6] uses, that makes it send a keep-alive frame every 100 ms. With a stand-in for openpi's client class, an object with the connection and the method that openpi's client uses, our openpi adapter brought an upload after a pause from two or more network delays to about one at every network delay tested, and the Zenoh setting did the same between two Zenoh peers; without either, every upload took more than 1.5 network delays (Tables 11 and 12). Robots that run openpi's own client can get the same reduction without replacing their protocol.

Network delay	Unmodified	Server pings	Our openpi adapter on the client	BBR alone on the client socket
11.3 ms	**37.6 (3.3)**	13.4 (1.2)	12.7 (1.1)	12.7 (1.1)
21.4 ms	**44.4 (2.1)**	23.5 (1.1)	22.7 (1.1)	23.7 (1.1)
40.5 ms	**82.4 (2.0)**	41.8 (1.0)	41.8 (1.0)	43.7 (1.1)
80.5 ms	**163.8 (2.0)**	82.8 (1.0)	82.6 (1.0)	87.1 (1.1)

Table 11 | A WebSocket upload after a pause. Between the two cloud VMs of Table 10, a 100 KB upload 0.53 s after each response, 40 times per setting, with the `websockets` library on both ends, as in openpi's server and client; no model runs. *Network delay*: measured with `ping`. Each cell: the median time from sending the upload to receiving the response, in ms, and in brackets in network delays. Unmodified, all 40 uploads took more than 1.5 network delays in every row; with any of the other three, none did. *Server pings*: the server sends WebSocket pings every 100 ms. *Our openpi adapter*: on the client; this machine had no BBR, so its pings alone kept the window.

Network delay	Zenoh, unmodified	With the keep-alive setting on both ends
11.3 ms	**38.1 (3.4)**	13.9 (1.2)
21.4 ms	**44.8 (2.1)**	23.2 (1.1)
40.5 ms	**82.8 (2.0)**	42.3 (1.0)
80.5 ms	**164.3 (2.0)**	82.2 (1.0)

Table 12 | A Zenoh request after a pause. Between the two cloud VMs of Table 10, two Zenoh peers over TCP, one answering the other: a 100 KB request 0.53 s after each response, 40 times per setting; no model runs. Each cell: the median time from sending the request to receiving the response, in ms, and in brackets in network delays, measured with `ping`. Unmodified, all 40 requests took more than 1.5 network delays in every row; with the setting that `lerobot.keep_alive_file` writes, a keep-alive frame every 100 ms, none did. LeRobot's own policy server and robot were not used.

4.6 Fewer bytes and fewer writes

Four components shorten the upload and the response by sending fewer bytes or fewer writes (Table 13); a fifth, sending observations as they are produced, is part of the client in `sequence-ai` that reaches the hosted service.

Component	Without it, the request is	Frames	Bytes saved	Time saved per request, ms: 20 Mbit/s: predicted / measured	Time saved per request, ms: Unlimited link
`frames`	JSON, raw frames	two 224×224	359,392	143.8 / 150.9	0.8
`frames`	msgpack, raw frames	two 224×224	269,570	107.8 / 111.4	−1.1
`msgpack_io`	JSON, raw frames	two 224×224	100,373	40.1 / 44.0	2.2
`compress`	msgpack, raw frames	two 224×224	145,278	58.1 / 60.3	−0.6
`frames`	msgpack, raw frames	three 640×480	2,544,958	1,018.0 / 1,057.4	24.6
`msgpack_io`	JSON, raw frames	three 640×480	921,617	368.6 / 411.3	47.7
`compress`	msgpack, raw frames	three 640×480	1,497,590	599.0 / 621.7	25.6
`one_write`	JSON, behind a proxy	two 224×224	0	0 / 49.9	−0.1
`one_write`	JSON, direct	two 224×224	0	0 / −0.1	0.0

Table 13 | Time saved by each byte or write component. Each row compares the same requests with and without one component, 200 requests per setting sent one after another with no pause, between two cloud VMs with a 50 ms network delay and a 20 Mbit/s uplink or an unlimited one (about 290 Mbit/s). The frames come from real robot video, resized on the robot before any component; the server decodes each request as a policy server would and returns a fixed chunk, and no model runs. `frames`: JPEG at quality 95; `msgpack_io`: msgpack with arrays as raw bytes, instead of JSON, which carries them as base64; `compress`: lossless compression of an array when it saves time; `one_write`: each response written in one piece. *Bytes saved*: per request. *Time saved*: the median request without the component minus the median with it, from sending the request to reading the reply; the robot's own time to encode the request is not included, and a negative value is a cost. *Predicted*: the time the saved bytes take at 20 Mbit/s. The proxy, a TCP proxy on the server's machine, can hold the server's second write until the first is acknowledged: on the slow link this delayed the median request, and on the unlimited link only the slowest 5%, which `one_write` brought from 106.2 to 55.4 ms.

Resizing and encoding frames. Cameras produce more pixels than policies use: one robot in our data sent two 1,280×720 frames to a policy that takes 224×224 images. `frames` resizes a frame on the robot to the policy's input; the caller specifies the mode and the resampling filter, since the filter is part of what a policy was trained on, and the padded mode reproduces openpi's own client byte for byte. It can also encode a frame as JPEG, which sends far fewer bytes (Table 13).

Arrays as raw bytes. JSON encodes bytes as base64, a third larger on the wire, and arrays as lists of numbers. `msgpack_io` encodes observations and responses as msgpack with numeric arrays as their raw data, reads the array layouts of two ecosystems, and rejects anything whose decoding could execute code. Binary arrays saved as much time as their fewer bytes predict on a slow uplink; on a fast link they saved little for two small frames and still tens of milliseconds for three large ones (Table 13). Binary frames cost nothing and save time on any link where the upload is large against its rate.

Compression only when it reduces latency. Compressing every array is counterproductive on a fast link, where compression takes longer than the upload time it saves, and worthwhile on a slow uplink. `compress` compresses an array, per path in the observation, as long as the upload time saved at the uplink's measured rate exceeds the compression and decompression time; `compressed` restores it byte for byte on the server. On camera frames, a fast lossless compressor nearly halved the bytes (Table 13), taking well under a millisecond on small frames. On a slow uplink carrying raw arrays, compression nearly halves the transmission time for little computation; on a link fast enough that compressing costs more than it saves, it is skipped. Learned compression [19] and sending only the visual tokens that matter [23] weigh the same coding time against upload time.

Single-write responses. uvicorn writes a response in pieces, and behind one type of tunnel we measured, each later piece was held until the client had acknowledged the first, so every request paid one extra network delay. `one_write` gives uvicorn's HTTP protocol a transport that sends everything written in one event-loop iteration as a single write, and `WholeResponse` holds a response's start until its first body part arrives. Behind that tunnel, writing each response in one piece removed the extra network delay, and it did so again when later middleware split responses. Behind a proxy that holds later pieces the same way, it saved one network delay on the median request over a slow uplink, and on a fast one in the slowest requests (Table 13). A server behind a tunnel or proxy that forwards later pieces only after earlier ones are acknowledged should write each response in one piece; on a direct connection it neither saved nor cost time (Table 13).

Sending observations as they are produced. A policy that uses several frames needs many of them before it can start, and if they are all uploaded with the request, the request waits for the whole upload. The platform's own client uploads each observation as soon as it is produced, on the same connection, with up to eight uploads on their way at

once, so that only the newest frame is on the request's critical path; this is the client in `sequence-ai` that reaches the hosted service, and the components for a user's own server do not stream. For camera frames sent as video, timed from capture until the server had decoded them, this nearly halved the median at a long network delay without loss, and it was faster in almost every lossy setting we tried. A camera frame marked as video goes further: `sequence-ai` codes it against the frames before it as low-latency H.264, with a keyframe every second, which sends far fewer bytes (Figure 10) but changes the pixels the policy sees.

4.7 Against other open-source transports

We compared GS Network with two transports that other open-source robot stacks use: Zenoh's query and reply, which LeRobot's remote inference and ROS 2's `rmw_zenoh` use, and a WebRTC data channel. Robots in Virginia, Iowa, Oregon, London and Tokyo sent $\pi_{0.5}$'s observation at its own timing to a stand-in server in Virginia that answers after 194 ms, as raw arrays of 301 KB and as JPEG of 23.6 KB (Figure 12). Zenoh's connection went idle before every request, so its uploads paid extra network delays after each pause: from Iowa its raw uploads took 2.5 times as long as GS Network's, and from Oregon and London its JPEG uploads took almost twice as long. With raw arrays, the WebRTC data channel took more than three times as long as GS Network from every city but Virginia, and it was late on nearly every request from Oregon, London and Tokyo, where GS Network was on time.

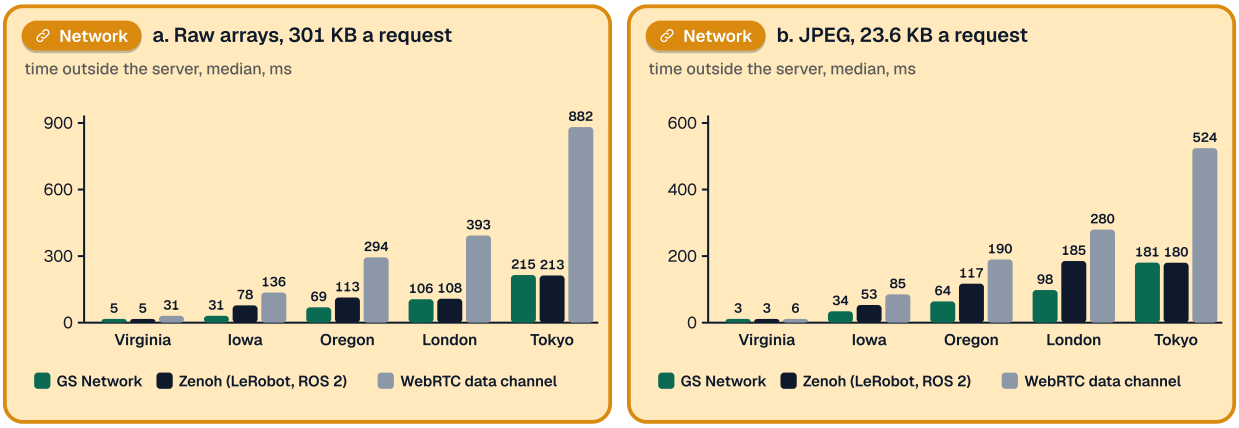


Figure 12 | GS Network against the transports of other open-source robot stacks. The median time a request spent outside a stand-in server in Virginia, which answers after 194 ms, from robots in five cities at $\pi_{0.5}$'s own timing, two runs of 60 s per setting: GS Network (WebSocket with server pings, binary arrays and single-write responses, from `sequence-ai`), Zenoh's query and reply as LeRobot's remote inference and ROS 2's `rmw_zenoh` use it, and a WebRTC data channel, the last two with their default settings. (a) Raw arrays, 301 KB a request. (b) JPEG, 23.6 KB a request.

4.8 A planner adds no traffic

A planner placed beside the policy on the server reads the observations the robot already sends, and its result stays on the server, where the next action call reads it: the robot sends nothing more and waits for nothing more (Section 3.9). Over a WAN this matters twice. A plan that takes longer than the robot's time budget never delays an action, and a planner that runs remotely costs the robot no extra network delay.

4.9 Using the network layer

The network layer is `sequence_ai.network`, in the package of Section 3.11: `window` and `connect` keep a robot's connection fast across its pauses, `ping` and `one_write` do their part on the server, `frames`, `msgpack_io` and `compress` send frames at the size the policy uses, compressed only when that saves time, and `openpi` and `lerobot` apply GS Network to those projects' own clients. On the robot, one call applies it to one connection, and the same call given a policy's name instead of an address reaches the hosted service.

```
import sequence_ai
# BBR, TCP keepalive, a ping while idle, msgpack, on one connection to your own server
link = sequence_ai.connect("http://your-gpu:8000")
actions = link.act({"image": frame, "state": state})["actions"]
```

The package also holds the tools that measure a robot's link: `bench/`, which times uploads after a pause, with pings and as a stream; `tcp_link`, which reads each client's network delay and retransmissions from the server's kernel; `latency_stats`, which keeps a server's recent latencies and reports their percentiles; and `wifi_power`, which warns when a robot's Wi-Fi power saving is on (Appendix B.2.9). The calls a robot program makes are in Appendix A.

5 Routing Layer

The routing layer decides where a robot's GPU is. Every request pays the network delay to its GPU out of its robot's budget (Equation 3), so how far a robot can be from its GPU depends on its policy. $\pi_{0.5}$, with a short budget and, on an L4, a long inference, has little time left for the path, and from Singapore it reached that limit and was late on more than a third of its requests; GR00T, with more than a second, has several times as much (Table 7). On a shared replica, a far robot also leaves less time to wait for a batch. If a replica planned every robot's batches by its farthest robot's delay, that one robot would lower the batch size, and with it the capacity, for all of them; GS-WAN therefore plans each robot by its own delay (Section 3.6). Where GPUs are, relative to robots, decides how many robots each GPU can serve (Section 5.1).

Two rules apply to any deployment:

1) Give each robot the closest GPU, because its network delay comes out of every request's budget and so decides how many robots the GPU can hold (Section 5.1).
2) For a policy that keeps state across an episode, keep the episode on one replica, log it outside the machine, and hold a standby replica that can continue it (Section 5.2).

These components run in the hosted service and are not released; details are in Appendix B.3.

5.1 Placement

The network delay is paid on every request, and the user should not need to know where GPUs are. As in FogROS2 [14], the platform gives each robot the closest GPU it can obtain. A robot across the country pays a network delay several times that of a nearby GPU on every request, and more after each pause unless the transport keeps the window (Table 9).

5.2 Episode pinning and a standby connection

A policy that keeps state across an episode cannot be served by whichever replica is free, and when its replica or its connection fails, an episode that starts over has lost the task. The router keeps a stateful episode on its own replica, as SGLang's router keeps a session [36]; each frame and each response are appended to an episode log stored outside the machine; and the robot holds a second, idle connection to a standby replica, to which its next request goes directly when the main connection fails, and which reads the episode back and continues it (Figure 13). FogROS2-LS also switches to a standby connection when a server fails [8]; the standby here also restores the episode's state. We timed how long a robot waits for its next answer after a failure, with the released client, when it switches to the standby connection and when it reconnects through the platform's gateway, which tells it which worker to use (Table 14). We expected every trial to recover, the standby to be at least twice as fast, and its wait to stay under half a second. Every trial recovered. Where the worker went silent, the standby answered in under a second, many times sooner than reconnecting, though not within half a second; where only the connection broke, both reconnected to the same worker, equally fast. With a standby, a robot whose replica went silent had its next answer in under a second: the client waits a few network delays to judge that a request is stuck, asks whether the worker is still there, and switches to the standby.

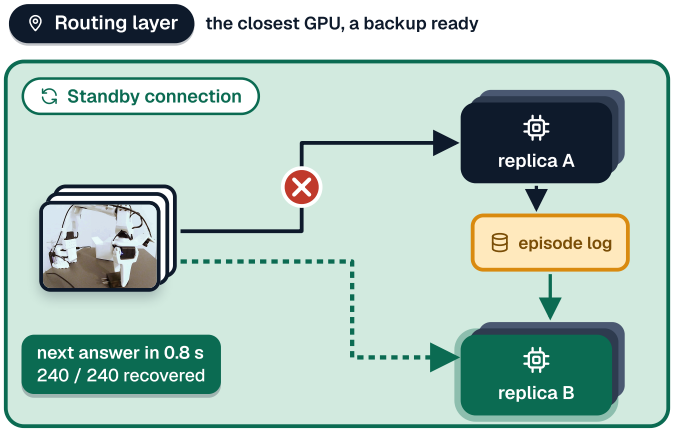


Figure 13 | The standby connection. A robot keeps a main connection to replica A and an idle standby connection to replica B. When the main connection breaks, the next request goes to B, which reads the episode log and continues the episode. With the released client, the robot had its next answer about 0.8 s after its replica went silent, where reconnecting through the gateway waited out the request's deadline, and every trial recovered (Table 14).

Failure	Recovery	Recovered	Median	Slowest 5%
The client's default deadline, 30 s				
Worker silent	standby connection	40 of 40	794.4	834.2
	reconnect through the gateway	40 of 40	35,539.1	35,542.9
Connection broken	standby connection	40 of 40	347.5	376.5
	reconnect through the gateway	40 of 40	344.0	347.4
The deployment's timeout set to 2 s (deadline 12 s)				
Worker silent	reconnect through the gateway	10 of 10	17,532.8	17,541.5
The platform's default timeout, 300 s (deadline 310 s)				
Worker silent	standby connection	3 of 3	789.2	830.4
	reconnect through the gateway	3 of 3	315,624.6	315,631.9
The robot's own 1 s timeout, `connect(timeout_s=1)`				
Worker silent	reconnect through the gateway	10 of 10	2,533.6	2,537.2
On a laptop, with the worker's resets reaching the robot (`sequence-ai` 0.62.3)				
Worker killed	standby connection	3 of 3	51.9	72.1
	reconnect through the gateway	3 of 3	151.3	164.4
Worker silent	standby connection	3 of 3	377.5	438.5
	reconnect through the gateway	3 of 3	35,154.9	35,205.2
	the same, also checking for a stuck request	3 of 3	514.2	530.3

Table 14 | Time from a failure to the robot's next answer, in ms, with a standby connection or by reconnecting through the platform's gateway. The first four parts: the released client (`sequence-ai` 0.61.5) at 15 Hz against a stand-in gateway and worker on one machine, with an 80 ms network delay and no model; in each round the order of the failures was shuffled and the order of the two recoveries alternated. *Worker silent*: every packet to and from the worker dropped. *Connection broken*: the robot's connections closed while the worker stayed. *Recovered*: trials in which the next ten requests were also answered. *Median* and *slowest 5%*: the median and the 95th percentile over the trials, interpolated, so with three trials the latter lies between the two slowest. Without a standby, a silent worker's wait is set by the request's deadline; the robot's own 1 s timeout bounds each attempt instead, and the robot reconnected after 2.5 s. The last part repeats the failures on a laptop over loopback, three trials each, with a proxy that delivers the worker's resets. *Also checking for a stuck request*: the client checks for a stuck request even without a standby.

On a laptop, where a killed worker's connection resets reach the robot, the client reconnected through the gateway within a fraction of a second, and checking for a stuck request also without a standby brought a silent worker's wait down to about half a second (Table 14).

With a standby, a robot recovers in under a second whether its worker is reset or goes silent; without one, the released client recovers quickly from a reset but waits out the deadline when the worker goes silent.

6 Evaluation

Figure 14 gives two key results of the inference layer and two of the network layer, each in one named setting; the routing layer's results are in Section 5.

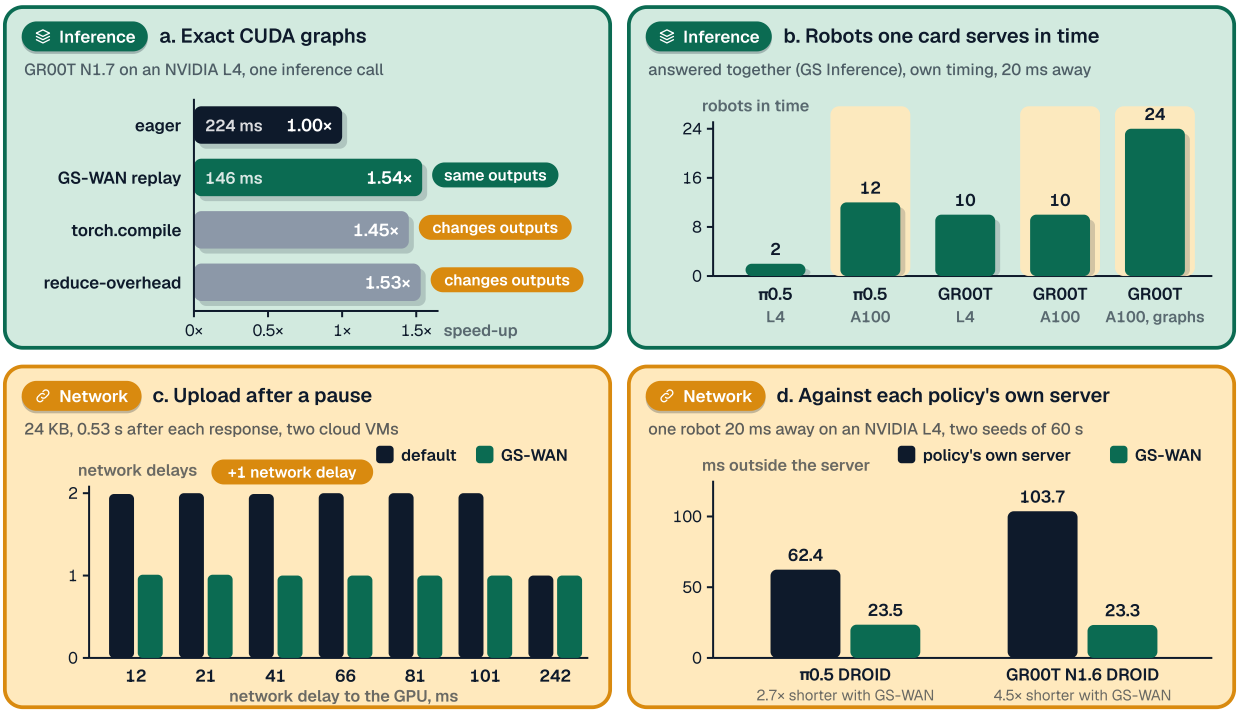


Figure 14 | Key results, each in one named setting. (a) GR00T N1.7 on an L4, one inference call: replaying the diffusion transformer from CUDA graphs made the call 1.54× faster with exactly the same outputs, while `torch.compile`, in its default mode and in its reduce-overhead mode, changed the outputs (Section 3.10). (b) Robots one card kept on time when it answered them together with GS Inference, each policy at its own timing and 20 ms away, on an L4 and on an A100-40GB (Figure 6 and Table 4). (c) Two cloud VMs: the median time of a 24 KB upload 0.53 s after each response, in network delays, at each network delay between them; it took two by default and one with GS-WAN's server pings, except at the longest, where the shrunken window still held the upload (Table 10). (d) One robot 20 ms away on an L4: the median time a request spent outside the server with each policy's own server, openpi's for $\pi_{0.5}$ and GR00T's own for GR00T, and with GS-WAN, whose pings keep the connection's window across the robot's pauses; it was 2.7 times shorter for $\pi_{0.5}$ and 4.5 times shorter for GR00T (Table 15).

6.1 Setup

The inference measurements used one L4 with PyTorch 2.9.0, with the policy called directly and no platform code in between; how many robots one card serves in time was measured on an L4 and on an A100-40GB (Section 3.5). Each component was measured on its own. The policies are $\pi_{0.5}$ [31], fine-tuned for the DROID dataset [18] and the LIBERO benchmark [22]; versions N1.6 and N1.7 of GR00T [28]; MolmoAct2; and SmolVLA [32]. SO-101, xArm6, YAM and Franka FR3 are robot arms, and the simulator is MuJoCo [35].

6.2 End to end over real paths

We compared GS-WAN with each policy's own server end to end, with one to eight robots in two cities sharing one L4 (Table 15). The robots were virtual machines sending random pixels as raw arrays, so GS-WAN's network layer sent pings but not JPEG frames. We expected each policy's own server, which answers requests one by one, to keep its robots in time up to the number it can answer that way, and to make nearly every response late past it; and GS-WAN to accept only the robots that fit, send the others elsewhere, and keep every accepted robot in time. Both held. GS-WAN served as many $\pi_{0.5}$ robots in time as $\pi_{0.5}$'s own server, and more GR00T robots: a GR00T robot can wait longer for its actions, so GS Inference answered several of them together in one pass. When more robots asked than its seats, GS-WAN sent the others elsewhere instead of letting all of them fall behind. With one robot, GS-WAN's pings kept the connection's window open across the robot's pauses and made the time outside the server 2.7 times shorter for $\pi_{0.5}$

and 4.5 times shorter for GR00T. On an A100, openpi's server kept four $\pi_{0.5}$ robots on time and was late on nearly every response once eight or more asked, while GS-WAN accepted eight, kept every one of them on time and sent the others elsewhere (Table 15).

	The policy's own server, one by one			GS-WAN			
Robots	Late	Total time, median	Outside the server, median	Accepted	Late	Total time, median	Outside the server, median
$\pi_{0.5}$ DROID on an L4, budget 467 ms							
1	0%	260	62	1	0%	221	24
2	0%	315	115	2	0%	290	33
4	100%, 99.3%	808	606	2	0%	289	28
8	100%	1,603	1,403	2	0%	342	32
GR00T N1.6 DROID on an L4, budget 1,133 ms							
1	0%	329	104	1	0%	248	23
2	0%	391	166	2	0%	411	30
4	0%	397	172	4	0%	691	31
8	100%	1,798	1,574	7, 6	0%	697, 438	25, 29
$\pi_{0.5}$ DROID on an A100-40GB, budget 467 ms							
1	0%	200	106	1	0%	151	60
2	0%	166	77	2	0%	166	45
4	0%	148	66	4	0%	179	36
8	99.7%, 100%	654	572	8	0%	273	36
12	100%	981	900	8	0%	231	25
16	100%	1,311	1,229	8	0%	200	25

Table 15 | Robots in two cities sharing one GPU, each policy's own server against GS-WAN, each policy at its own timing: the robots split between a machine 20.5 ms and one 30 ms away from the L4, and 21 and 52 ms away from the A100, two seeds of 60 s per setting. *Robots*: how many asked to be served. *Accepted*: how many GS-WAN accepted; it sent the others elsewhere, while each policy's own server took every robot. *Late*: the share of the accepted robots' responses that arrived after their budget. *Total time*: from sending a request to receiving its actions, inference included (about 197 ms of it for $\pi_{0.5}$ and 225 ms for GR00T in these runs), in ms. *Outside the server*: the total time less the time in the server, in ms. Medians are averaged over the two seeds; where the seeds differ in robots accepted or in the late share, each seed's value is given. The policy's own server is openpi's `serve_policy` for $\pi_{0.5}$ and GR00T's ZeroMQ server; it answers requests one by one and sends no pings.

6.3 Task success against delay

We tested whether a fixed delay lowers task success when the robot executes the same number of actions from each observation. $\pi_{0.5}$'s LIBERO template asks for a new chunk only after executing the current one ($N = k = 5$ at 20 Hz), and the benchmark executes a chunk $N - k$ steps after its observation, so declaring other values of (N, k) delays every action by a fixed time without changing the benchmark's code: $(10, 10)$ is the control, $(10, 5)$ delays every action by 250 ms and $(10, 2)$ by 400 ms. We expected the delay to lower success, and it did, far more than we had assumed: with every action delayed by 400 ms, success on LIBERO-10 fell from almost every episode to about a quarter of them, on the same tasks and seeds (Table 16). The delay never turned a failure into a success; it failed every episode of most tasks and left one task untouched.

Declared (N, k)	Delay	Open-loop steps	Success, 50 episodes	95% interval (%)
$(10, 10)$, control	0	10	96% (48 of 50)	86.5 to 98.9
$(10, 2)$	400 ms	10	24% (12 of 50)	14.3 to 37.4

Table 16 | Task success against a fixed delay: $\pi_{0.5}$ on LIBERO-10 in simulation, ten tasks with five episodes each, for each declared (N, k) at 20 Hz. *Delay*: how much later than in the control each action is executed. *Open-loop steps*: how many actions the robot executes from one observation, the same in every setting. *95% interval*: Wilson's, in percent. The difference between the two settings that ran, 72 percentage points, has a 95% interval of 55.6 to 82.1 (Newcombe's); across the 50 pairs of episodes on the same tasks and seeds, the delay turned 36 successes into failures and none the other way (McNemar's exact test, $p = 2.9 \times 10^{-11}$).

7 Limitations

- **Simulated robots.** The robots in our measurements were simulated, on cloud machines and on residential links, rather than a production fleet.
- **Two GPU types.** The policies were measured on NVIDIA L4 and A100 GPUs.
- **Network delay from a handshake.** We measure a path's network delay by its TCP handshake, which on a slow uplink leaves out the time a large upload takes to transmit.

8 Conclusion

A robot that runs its policy on a remote GPU gives every request a deadline: the next actions must arrive before the robot runs out of the current ones. That one deadline ties the whole path together. The network delay, the extra delays that TCP and servers add, and the wait for a shared GPU all come out of the same budget, so time removed from the network is time a request can spend waiting to share a GPU pass, and a GPU that answers robots together within their deadlines can serve more of them at a lower cost per robot. Delay also costs the task itself: in simulation, 400 ms of delay lowered a policy's success from 96% to 24% (Section 6.3). GS-WAN applies this rule at every layer. GS Inference answers robots together, holds each request only as long as its robot can wait, and accepts only as many robots as it computes a GPU can serve on time: answering robots together kept up to twice as many robots on time on one GPU for both policies we measured, each on the GPU that suited it, and every robot GS-WAN accepted was on time while the policies' own servers fell behind once overloaded (Sections 3.5 and 6.2). GS Network keeps the connection fast across a robot's pauses and sends frames as low-latency video: a request's time outside the server was 2.7 and 4.5 times shorter than with the policies' own servers, and video carried about a hundredth of the raw bytes (Sections 4 and 6.2). The routing layer gives each robot the closest GPU it can obtain and keeps a standby ready. None of this retrains or modifies a policy, so the framework applies to any policy that declares the timing of its action chunks. GS Inference and GS Network are released as `sequence-ai`.

References

[1] Dionysios Adamopoulos, Nattapol Chanpaisit, Basel Fakhri, and Christina Giannoula. Efficient vision-language-action management and serving for robot factories. arXiv:2609.12075 [cs.DC], 2026.

[2] Ahnaf Adib, Sabbir Ahmed, and Latifur Khan. FirmVLA: Sequence-SLO serving for heterogeneous VLA fleets. In *NeurIPS Workshop on On-Device Intelligence: Foundation Models under Real-World Constraints*, 2026.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, et al. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

[4] Rohan Bansal, David He, Nadun Ranawaka Arachchige, Zhenyang Chen, Soobum Kim, Kexin Rong, and Danfei Xu. Action chunk scheduling for batched robot policy serving. arXiv:2608.00337 [cs.RO], 2026.

[5] Kevin Black, Manuel Y. Galliker, and Sergey Levine. Real-time execution of action chunking flow policies. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[6] Remi Cadene, Simon Alibert, Alexander Soare, Quentin Gallouedec, Adil Zouitine, Steven Palma, Pepijn Kooijmans, Michel Aractingi, Mustafa Shukor, Dana Aubakirova, Martino Russi, Francesco Capuano, Caroline Pascal, Jade Choghari, Khalil Meftah, Maxime Ellerbach, Jess Moss, and Thomas Wolf. LeRobot: State-of-the-art machine learning for real-world robotics in PyTorch. https://github.com/huggingface/lerobot, 2024.

[7] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *Communications of the ACM*, 60(2):58–66, 2017.

[8] Kaiyuan Chen, Michael Wang, Marcus Gualtieri, Nan Tian, Christian Juette, Liu Ren, Jeffrey Ichnowski, John Kubiatowicz, and Ken Goldberg. FogROS2-LS: A location-independent fog robotics framework for latency sensitive ROS2 applications. Author-hosted PDF, https://autolab.berkeley.edu/assets/publications/media/2024-05-FogROS2-LS.pdf, 2024.

[9] Amnon Cohen. Optimizing global message transit latency: a journey through TCP configuration. Ably engineering blog, July 2024. https://ably.com/blog/optimizing-global-message-transit-latency-a-journey-through-tcp-configuration.

[10] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[11] G. Fairhurst, A. Sathiaseelan, and R. Secchi. Updating TCP to support rate-limited traffic. RFC 7661 (Experimental), 2015.

[12] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[13] M. Handley, J. Padhye, and S. Floyd. TCP congestion window validation. RFC 2861 (Experimental), 2000.

[14] Jeffrey Ichnowski, Kaiyuan Chen, Karthik Dharmarajan, Simeon Adebola, Michael Danielczuk, Víctor Mayoral-Vilches, Nikhil Jha, Hugo Zhan, Edith LLontop, Derek Xu, Camilo Buscaron, John Kubiatowicz, Ion Stoica, Joseph Gonzalez, and Ken Goldberg. FogROS2: An adaptive platform for cloud and fog robotics using ROS 2. arXiv:2205.09778 [cs.RO], 2022.

[15] J. Iyengar and M. Thomson. QUIC: A UDP-based multiplexed and secure transport. RFC 9000 (Proposed Standard), 2021.

[16] Wenqi Jiang, Jason Clemons, Rowland O'Flaherty, Hugo Hadfield, Alperen Degirmenci, Shuran Song, Yashraj Narang, and Christos Kozyrakis. ROSA: A robotics foundation model serving system for robot factories. arXiv:2607.01088 [cs.RO], 2026.

[17] Tobias Jülg, Khaled Gamal, Nisarga Nilavadi, Pierre Krack, Seongjin Bien, Michael Krawez, Florian Walter, and Wolfram Burgard. VLAgents: A policy server for efficient VLA inference. arXiv:2601.11250 [cs.RO], 2026.

[18] Alexander Khazatsky, Karl Pertsch, Suraj Nair, et al. DROID: A large-scale in-the-wild robot manipulation dataset. arXiv:2403.12945, 2024.

[19] Hyeonjun Kim, Jegwang Ryu, Sangbeom Ha, Junhyeok Lee, Jun-Hyuk Kim, Hyemin Ahn, and Jaeho Lee. Learned image compression for vision-language-action models. arXiv:2606.16253 [cs.CV], 2026.

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

[21] Yinhao Li, Weixin Mao, Zihan Lan, Jikun Rong, Qirui Hu, Yiming Zhang, Weipeng Deng, Bowen Shen, Minzhao Zhu, Yiming Mao, Yan Yang, Chenguang Cui, Hongyuan Chen, Xu Huang, Zheyi Zhao, Pinxi Shen, Bozhen He, Zhen Fu, Yifan Wang, Zexin Zhang, Ang Gao, Haoyu Chen, Chengqi Shi, Wenda Wu, and Hua Chen. FluxVLA engine: A one-stop VLA engineering platform for embodied intelligence. arXiv:2609.17210, 2026.

[22] Bo Liu, Yifeng Zhu, Chongkai Gao, et al. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. arXiv:2306.03310, 2023.

[23] Boliang Liu, Wint Yi Poe, Jingyun Di, Riccardo Trivisonno, and Giuseppe Caire. ComVLA: Communication-aware split inference for VLA models in 6G-connected robotics. arXiv:2609.07838 [cs.RO], 2026.

[24] Yunchao Ma, Yizhuang Zhou, Yunhuan Yang, Tiancai Wang, and Haoqiang Fan. Running VLAs at real-time speed. arXiv:2510.26742 [cs.RO], 2025.

[25] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 2022.

[26] NVIDIA. CUDA C++ programming guide, section CUDA graphs. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024.

[27] NVIDIA. Isaac-GR00T: deployment guide, scripts/deployment/README.md. https://github.com/NVIDIA/Isaac-GR00T, commit 8af93e5, 2026.

[28] NVIDIA, Johan Bjorck, et al. GR00T N1: An open foundation model for generalist humanoid robots. arXiv:2503.14734, 2025.

[29] Sungjae Park and Shubham Tulsiani. $\pi\mathrm{R}^2$: Reactive real-time flow policies. arXiv:2607.26055 [cs.RO], 2026.

[30] Physical Intelligence. openpi. https://github.com/Physical-Intelligence/openpi, 2024.

[31] Physical Intelligence, Kevin Black, Noah Brown, et al. $\pi_{0.5}$: a vision-language-action model with open-world generalization. arXiv:2504.16054, 2025.

[32] Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, Simon Alibert, Matthieu Cord, Thomas Wolf, and Remi Cadene. SmolVLA: A vision-language-action model for affordable and efficient robotics. arXiv:2506.01844 [cs.LG], 2025.

[33] Liang Su, David Chen, Jason Li, Kevin Guo, and Teresa Lv. FlashRT-PAI: Fast kernels are not fast pipelines for physical AI inference. arXiv:2610.24349 [cs.AI], 2026.

[34] Jiaming Tang, Yufei Sun, Yilong Zhao, Shang Yang, Yujun Lin, Zhuoyang Zhang, James Hou, Yao Lu, Zhijian Liu, and Song Han. VLASH: Real-time VLAs via future-state-aware asynchronous inference. arXiv:2512.01031 [cs.RO], 2025.

[35] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

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

A Using GS-WAN

Table 17 lists every component of GS-WAN with its layer, the module of `sequence-ai` that implements it, whether it is released, and where this report measures it.

Component	Layer	What it does	Module	Released	Measured in
On the robot					
Observation encoder	network	resizes frames to the policy's input; sends arrays as raw bytes; compresses only when that is faster	`frames`, `msgpack_io`, `compress`	yes	Section 4.6, platform form
Connection keeper	network	keeps the connection's sending window open while the robot pauses	`window`, `connect`	yes	Section 4.3
for existing clients	network	the same for openpi's client and LeRobot's Zenoh transport	`openpi`, `lerobot`	yes	Section 4.5
On the GPU server					
Connection keeper	network	sends small messages (pings) while the robot pauses; sends each response in one write	`ping`, `one_write`	yes	Sections 4.4, 4.6
GS Inference	inference	answers several robots together in one pass, in the group sizes the policy allows, holding each request only as long as its robot can wait; accepts a robot only while the GPU can still answer it in time	`slack`, `batcher`	yes	Sections 3.4 to 3.6
Policy replica	inference	runs one copy of the model per GPU, in its own process, and replaces it when it dies	`replicas`	yes	Section 3.8, platform form
Exact CUDA graphs	inference	records the GPU work of a part of the model that runs many times per inference and replays it with one launch, kept only if its outputs are exactly the same	`cuda_graphs`	yes	Section 3.10, platform form
Hosted (GPU Routing)					
Placement	routing	chooses a nearby region	n/a	no	Section 5.1
Standby replica	routing	keeps a robot's episode on one copy of the model, with a standby connection to a second	n/a	no	Section 5.2
Measurement tools					
Link and latency tools	all	measure a robot's link and a server's latencies	`bench/`, `tcp_link`, `latency_stats`, `wifi_power`	yes	Tables 10 and 11

Table 17 | The components of GS-WAN. *Layer*: what the component acts on: the network between a robot and its GPU, inference on the GPU, or routing a robot to a GPU; the network rows together make up GS Network. *Module*: the file of the released package `sequence-ai` that implements it. *Released*: whether it is in that package. *Measured in*: where this report measures it; "platform form" means the measurement used the platform's code, from which the released module was extracted. A connection's sending window, TCP's congestion window, is how much data it may send before it waits for an acknowledgement, and Linux shrinks it after a pause longer than the connection's retransmission timeout; an episode is one attempt at a task.

A.1 Calls of a robot program

A robot program calls a small set of functions.

- `observe(inputs)` sends an observation as soon as it is available.
- `act(observation)` requests the next action chunk, a short sequence of actions the robot executes one per control step.

- `end_episode()` ends an episode, one attempt at a task; `label(success, notes)` records whether it succeeded.

The policy's author declares, on the policy itself, the control rate `hz`, the number of valid actions in a chunk (`action_chunk_valid_len`, N) and the step at which the robot requests the next chunk (`action_chunk_replan_at`, k). The robot therefore has $N - k$ steps of its current chunk left when it sends a request: its *buffer*, the time within which the next chunk must arrive for the robot to keep moving. A policy may also declare that the server may answer several robots' requests together in one pass of its model, and which numbers of requests, its batch sizes, that pass accepts, with `@seq.concurrent(sizes=...)`.

A program can also run a planner beside its policy (Appendix B.4). A `plan` method runs on its own timer (every 2 s in one of our policy templates, the policies as packaged for our platform) and selects the next subtask for the policy; `act` never waits for it.

A.2 Declared timing, benchmarks and planning

From the declared N and k, the server knows each request's *slack*, how long the request may wait at the server for other robots' requests and still return before its robot runs out of actions (Section 3.5), and each replica's *seats*, how many robots one replica, a loaded copy of the model, can answer in time (Section 3.6). A policy may instead declare a fixed time for which each request may be held for other requests (`max_wait_for_batch`), in which case its robots wait when they run out of actions. The same declaration drives the benchmark harness, which reproduces the order of observations and requests a robot would produce, so the order of observations and requests in a benchmark does not depend on how fast its machines respond (Appendix B.4). In one program run on the platform, a plan ran every 2 s and took several times as long as a policy request, and the policy's requests ran alongside it.

A.3 The open-source framework

The inference and network components are released as one Python package, `sequence-ai`, under Apache-2.0, with the requirements given in Section 3.11: the inference components in `sequence_ai.engine` and the network components in `sequence_ai.network`. Each component is one file, and most can be added on their own to an existing server or robot program; `connect` and `openpi` combine the client-side components.

On the robot, keeping the sending window open across pauses (Section 4.3):

```
from sequence_ai.network import window

transport = httpx.HTTPTransport(socket_options=window.socket_options())
window.keep(sock)                          # or: BBR on your own socket
warm = window.Warm(ping=connection.ping) # where there is no BBR
warm.used()                                # each time you send
window.check()                             # warns once: how to fix the machine
```

On the GPU server, single-write responses (Section 4.6):

```
uvicorn.run(WholeResponse(app), http=one_write_http())
```

On the GPU server, CUDA-graph capture of a module (Section 3.10):

```
from sequence_ai.engine import cuda_graphs

census = cuda_graphs.Census(cuda_graphs.modules_of(model))
model(observation); census.end_act()     # a few acts
census.close()
print(census.candidates())               # which modules are worth capturing
cuda_graphs.install(model, ["denoiser"]) # kept only if it matches eager exactly
```

On the GPU server, answering several robots together in one pass, with one process per GPU (Section 3.4):

```
batcher = Batcher(run, sizes=(1, 2, 4, 8), wait_ms=2.0)
response = await batcher(item)
```

```
pool = ReplicaPool(load)                  # one process per GPU
response = await pool.run(payload)
```

GS Inference (Sections 3.5 and 3.6):

```
from sequence_ai.engine import slack

table = slack.measure(forward_of, sizes=[1, 2, 4])       # forward_of(n), timed
play = slack.playback(hz=20, valid_len=15, replan_at=8)  # the next requested at the 8th
seating = slack.Seating(table, play["buffer_s"] * 1000, sizes=[1, 2, 4],
                        cycle_ms=(play["play_s"] - play["buffer_s"]) * 1000)
seats, over = seating.seat([("near", 30.0), ("far", 200.0)])  # (robot, delay in ms)
```

If answering one, two or four robots together takes 80, 110 or 130 ms, the call reports that this replica has seats for four robots, and that the robot 200 ms away is still the one to move to another GPU, because a robot that far away has too little of its buffer left for the server.

The snippets follow the components' docstrings, which are the reference.

The client side as one call, against the user's own server:

```
import sequence_ai

link = sequence_ai.connect("http://your-gpu:8000")   # BBR, keepalive, a ping while idle
response = link.act({"image": frame, "state": state})   # msgpack both ways
```

`connect()` holds one connection and never runs a ping and a request concurrently: an HTTP/1.1 client that starts a request while a ping is in flight opens a second connection for it, with a new handshake and a sending window that starts small.

On the GPU server, server-side pings (Section 4.4):

```
async with websockets.serve(handler, host, port, **ping.websockets_options()):
    ...
uvicorn.run(app, **ping.uvicorn_options())
```

With existing policy clients:

```
kept = openpi.keep(policy)      # openpi's own client and connection
path = lerobot.keep_alive_file("zenoh-keep-alive.json5")  # LeRobot 0.7
```

The first applies BBR, a TCP congestion control that keeps the window open across a pause, together with TCP keepalive and pings while the connection is idle, to the connection that openpi's own client holds; nothing in openpi is replaced. It was tested against openpi's client package, version 0.1.0, on a local connection. The second writes one Zenoh setting; Zenoh is the transport of LeRobot 0.7.0's remote inference. With it, two idle Zenoh peers (version 1.10.2) on one machine sent a frame in each direction several times a second; without it, less than once a second. Between two cloud VMs, each brought a 100 KB upload after a half-second pause from two or three network delays to one, a network delay being the time a message takes to reach the other end and come back (Tables 11 and 12).

A.4 Reproducing the measurements

The upload measurements can be repeated with the package's `bench/server.py` and `bench/client.py`, which use only the standard library. The client times an upload after an idle period, with server pings, with client pings, back to back and as a stream, and reports each as a multiple of the connection's network delay; `bench/README.md` gives the commands. The idle-window experiment of Section 4.3, which timed an upload after a pause, used the platform's own client, which is not released. The released tool reproduces its setting with `--sizes 24k --idle-ms 530 --samples 40 --modes idle,ping,keepwarm --warm 1m`. The released tool's own run is Table 10; `bench/ws_bench.py` and `bench/zenoh_bench.py` reproduce Tables 11 and 12, and the package's `docs/measurements.md` lists every command.

B Component Details

This appendix describes each component's design in detail. Most of the measurements it quotes were taken while building or operating the platform, before the experiments of the main text; they show what each component was built against, and the main text does not rest on them. Each component is described under the same headings: *motivation*, *design*, *implementation* (the code path and parameters, with file names for released components), *measurements* and *when it helps*.

B.1 Inference: One Model on One GPU

At the inference layer, inference is almost all of a request's time in the server (Table 1), and with one robot the GPU runs no kernel most of the time (Table 2). Two problems follow: making one inference faster without changing its outputs, and sharing one GPU among several robots without any robot running out of actions. The components in this section run on the GPU server. CUDA-graph capture shortens one inference (Section 3.10); batching answers several robots together in one forward pass of the model (Section 3.4); one process per replica, a copy of the model loaded on one GPU, keeps a forward pass from blocking the server (Section 3.8); GS Inference bounds how long a request may wait and how many robots a replica serves (Sections 3.5 and 3.6). All five are released in `sequence_ai.engine`.

B.1.1 Replaying a module's eager kernels from CUDA graphs

Motivation. The action head of a diffusion or flow policy calls one module, its denoiser, several times per inference with the same input shapes. Eager PyTorch, the default mode, runs each operation as soon as Python reaches it: each call launches its kernels one by one from the host, the machine's CPU, and for a small model the GPU spends more time waiting for the next launch than computing. One inference of GR00T launches thousands of kernels, and the GPU spends much of the time waiting between them. A general-purpose compiler removes this overhead by fusing operations, which reorders the arithmetic and therefore changes the outputs. A policy validated on its eager outputs would then have to be validated again.

Design. A CUDA graph records the kernels that a piece of code launches and replays them with a single launch. This component records the kernels of the module's *own eager execution* and replays exactly those, so the arithmetic, and therefore the output, is identical to eager execution. Four rules make it safe to enable on a model that has not been inspected.

- *A census identifies the module.* Over a few inferences, forward hooks count the calls each submodule receives, its input shapes and its GPU time. A candidate is a module called at least twice in every profiled inference, with one shape throughout an inference, costing at least 1 ms per inference; among nested candidates the outermost is chosen, and at most four are returned, slowest first.
- *Every shape is verified at capture time.* The first call with a new shape runs eagerly. The same call is then captured and replayed on the same inputs, and the graph is kept only if its output is exactly the same as the eager output. A shape whose capture fails, cannot be returned, or produces a different output runs eagerly from then on, and the component logs the reason once.
- *The random number generator is swapped during capture.* A model draws its noise from the GPU's generator. The capture runs against a clone of the generator state, and the original generator is restored afterwards, so a capture, even a failed one, leaves the model's subsequent draws unchanged.
- *The module stays in place.* Only its forward method is replaced; it remains the same object of its own class, so a model that checks the type of its submodules, reads their `state_dict`, or has hooks on them sees no change.

Implementation. `cuda_graphs.install(model, paths)` wraps the forward method of each named submodule in a `Graphed` object. A call is keyed by the signature of its arguments (the shapes and types of its tensors and the values of its non-tensor arguments). For a key that already has a graph, `Graphed` copies the inputs into the graph's fixed input tensors, replays the graph, and returns a copy of its output tensors. For a new key, it runs eagerly and then captures: it allocates fixed input tensors, runs the module once on a side stream so that lazy initialization happens outside the capture, and records a second run into a `torch.cuda.CUDAGraph`; all graphs of one module share one memory pool. The generator swap requires `torch` 2.4 or later; on older versions nothing is captured and the model runs unchanged. On a machine without a CUDA GPU, `install` captures nothing and reports this. `Graphed.report()` lists the captured shapes, the reason each remaining shape runs eagerly, and each capture's duration; a module-level switch disables replay so that eager and graph execution can be timed against each other in one process.

The generator swap addresses an observed failure. With `torch` 2.9 on an L4, after a failed capture, restoring the generator's saved state and then drawing from it raised an error about an offset increment outside graph capture for the rest of the process; with the swap, the draws after a failed capture matched those of a run without any capture.

Measurements. We expected replay to remove the launch overhead without changing the outputs, and on the diffusion transformer of GR00T N1.7 on one L4 it did: inference became about one and a half times faster with exactly the same outputs, checked on inputs not used for capture, on a new prompt, and at a batch size of four (Table 18). On an A100, whose host launched kernels more slowly than the L4's, eager GR00T N1.6 was slower than on an L4, and replay made it about three times faster, again with exactly the same outputs (Table 18). Capture took a fraction of a second per input shape on both cards, and for GR00T N1.7 on the L4 it used no additional GPU memory. Capturing the whole denoising loop instead of the module gave the same time, and has to be written for each model. In the same setup, the default `torch.compile` (inductor) gained a little less than replay, only after a long compilation, and changed the outputs; `torch.compile` with its CUDA-graphs backend gained nothing. On a T4, a synthetic denoiser also ran faster from graphs; this run verifies that capture, the byte-level check and the generator handling work on a real GPU, and makes no speedup claim for a real policy.

Execution	Time (ms)	Speedup	Outputs	Setup
GR00T N1.7 DROID on an L4, one call of the policy				
Eager	224			
Module from CUDA graphs	146	1.54×	the same	0.1 s per shape
Whole loop as one graph	145.7			written per model
`torch.compile`, inductor		1.45×	changed	48.5 s of compilation
`torch.compile`, CUDA-graphs backend		1.00×		
`torch.compile`, reduce-overhead mode		1.53×	changed	
GR00T N1.6 DROID, the model's own call				
L4, eager	211.4			
L4, modules from CUDA graphs	162.5	1.30×	the same	under 0.4 s per shape and module
A100-40GB, eager	296.1			
A100-40GB, modules from CUDA graphs	97.0	3.05×	the same	under 0.4 s per shape and module

Table 18 | One inference of GR00T, run eagerly or sped up in other ways, on an L4 and an A100-40GB. Upper part: GR00T N1.7 on an L4, timed as one call of the policy as the platform runs it, input processing included, with its denoiser replayed from CUDA graphs as this component does, the whole denoising loop captured as one graph, or `torch.compile` with its default backend (inductor), its CUDA-graphs backend or its reduce-overhead mode. Lower part: GR00T N1.6 on both cards, timed the same way on each as the model's own call, with the modules this component chose replayed from CUDA graphs. *Time*: the median of 40 calls, in ms. *Speedup*: the eager time over the method's, on the same card. *Outputs*: whether the actions were bit for bit the same as eager execution's; the compiled outputs differed from them by at most 0.043 rad, and by 0.0051 rad on average. *Setup*: what the method costs before its first fast call. A blank cell is a value we do not report. The A100's host launched kernels more slowly than the L4's, so eager GR00T was slower there and replay gained more.

When it helps. When the GPU is waiting for the host. In eager mode, GR00T's GPU computes for only part of the model's time; the graph removes the gaps. On this model the exact method was as fast as `torch.compile`'s fastest mode, which changes the outputs (Table 18). Capturing CUDA graphs for these models is established practice [24, 33, 21]; this component adds the automatic choice of the module, the bit-level check against eager execution and the handling of the random number generator.

B.1.2 Batching: answering several robots in one pass

Motivation. One robot keeps a GPU busy for only a fraction of the time (Table 2); the remainder can serve other robots. If their requests run one after another, each waits for the others' forward passes; answered together in one pass, a *batch*, they share one.

Design. For a policy that declares that its model can take several requests in one call, pending requests are combined into one forward pass. Three rules govern it, each added after a failure on the platform.

- *A batch is dispatched when it is full, or after a short wait from its first call if the model is free.* While the model is busy, the open batch keeps collecting calls and is dispatched as soon as the model is free. Calls that arrive during a forward pass would wait for it anyway, so they wait together. Cutting batches on a timer

instead turned calls that arrived over tens of milliseconds into a series of small batches queued behind each other.

- *Only the batch sizes the policy declares are used.* A model is traced, compiled or graphed for the shapes it has seen, and a batch of an unseen size is a new shape; a JAX policy compiles each batch size it is called with. Pending calls are packed into the largest declared size they fill, in arrival order, and the remainder forms the next batch the same way: with sizes 1, 2 and 4 declared, three calls go as a batch of 2 followed by a batch of 1.
- *A batch that raises an exception is retried call by call,* so only a call that fails on its own returns an error.

Implementation. The released `Batcher(run, sizes=(1, 2, 4, 8), wait_ms=2.0, runners=1)` wraps any function from a list of inputs to a list of outputs; a request handler awaits `batcher(item)`. A plain function runs on a thread, so a forward pass does not block the server's event loop; `runners` is the number of batches that may run concurrently, one per model replica. The platform's batcher follows the same rules and differs in what sets the wait: for a robot it is the robot's own *slack*, how long its request can wait in the server before the robot would run out of actions (Section 3.5); for a benchmark, whose environments send requests together, a batch is dispatched once 20 ms pass with no new request, once it is full, or at the latest 250 ms after its first request, and the requests of a round that identifies itself are dispatched as soon as the round is complete. When a replica starts, each declared batch size is run once untimed and then three times timed, so that the first batch of each size does not include compilation: for one $\pi_{0.5}$ deployment, a first call at three batch sizes had made the slowest 5% of requests take tens of seconds, against a median under a second.

Measurements. On an L4, answering several requests together in one pass raised $\pi_{0.5}$'s throughput by less than twofold, and on an A100-40GB by more than twofold (Table 19). On the L4, the reason is that the vision and prefix computation, over the camera images and the prompt, costs almost as much per request in a batch as alone, while the action expert, the small network that then produces the actions, becomes nearly free. The GPU adds to this: an L4 is capped at 72 W, and a larger batch lowers its clock.

	L4, earlier runs			A100-40GB
	$\pi_{0.5}$ DROID	$\pi_{0.5}$ LIBERO		$\pi_{0.5}$ DROID
Batch size	Requests/s	ms per batch	ms per request	Requests/s
1	5.99	201.5	201.5	14.22
2	8.21	319.0	159.5	18.74
4	9.38	542.4	135.6	24.53
8	10.52	1,055.5	131.9	29.83
16	10.66			33.46
Gain	1.78×		1.53×	2.35×

Table 19 | Answering several $\pi_{0.5}$ requests together in one pass, on one L4 and one A100-40GB. *Batch size*: how many requests one pass answers. *Requests/s*: requests answered per second, the batch size over the time of one pass. *ms per batch*: the time of one pass; *ms per request*: that time over the batch size. *Gain*: the highest throughput over that at batch size 1. DROID and LIBERO are two of $\pi_{0.5}$'s published checkpoints. The L4 columns come from earlier runs; the A100 column is computed from the median of 40 calls of the model alone at each batch size, timed as in Table 3, where $\pi_{0.5}$ gained 1.5× on an L4. A blank cell was not measured.

One replica per GPU. Two replicas of $\pi_{0.5}$ on one L4 are two processes that the GPU runs in turns: when their inferences overlap, each takes 2.1 times as long as alone. The CUDA Multi-Process Service (MPS) made them only slightly faster. One replica answering two robots together in one pass responded faster at the median than two replicas, one per robot. The platform therefore runs one replica per GPU. The open-source `replicas` component runs one process per GPU and restarts it when it dies; it does not raise throughput on a single GPU.

When it helps. For the cheap, repeated part of a model: at batch size 16, the action expert's cost per request almost vanishes. In evaluation, where many environments send requests at once: 64 environments of one task formed large batches.

B.1.3 One process per model replica

Motivation. A forward pass that runs in the server's own process blocks its event loop, and with it every other robot's request and every health check. A model that crashes brings the server down with it.

Design. The `replicas` component starts a fixed set of replicas, each a separate process bound to one GPU, and dispatches each call to a free replica. A call waits in a queue for a free replica, so requests queue per GPU rather than per Python interpreter. The component distinguishes a model that raised an exception from a process that died, because the two need opposite handling: the first is still loaded and takes the next call; the second holds nothing and is replaced. After a few consecutive deaths, the pool stops replacing processes and reports an error, since a model that dies on every call would otherwise be reloaded again and again, each load costing minutes of GPU time.

Implementation. `ReplicaPool(load, replicas=1, devices=None, start_timeout=900.0, deaths_in_a_row=3)` takes a function that a new process can import and that builds the model. Processes are spawned, never forked: a forked child inherits its parent's CUDA state, which CUDA does not support. Each replica is assigned its GPU through `CUDA_VISIBLE_DEVICES`; inputs and outputs are pickled over a pipe. The exception types name the cases: `ReplicaError` (the model raised), `ReplicaDied` (the process is gone), `ReplicaStartFailed`, `PoolHalted`.

When it helps. On a machine with several GPUs, or when the server must stay responsive during a forward pass.

B.1.4 How long each request waits

Motivation. Waiting to form a batch is only worthwhile if the robot can afford the wait. Holding every request a fixed few milliseconds to form a batch can fail in both directions: if the hold is too short, robots that could have shared a forward pass miss each other; if it is too long, the response arrives after the robot has run out of actions. On a stand-in server that replays a card's measured batch times without the model, with every robot 20 ms away, neither happened: holds of 2 and 50 ms served as many robots as GS Inference, with the batch times of an L4 and of an A100 alike (Section 3.5). With robots at different distances, on the A100's $\pi_{0.5}$ batch times, the longest hold served far fewer robots and the shortest as many as GS Inference (Table 5). Before this component, four robots sharing a replica received many of their responses late (see below).

Design. A robot requests its next chunk while part of its current chunk is still unexecuted; the time those remaining actions take is its *buffer*. Its request may wait in the server for that remaining time, minus the time the robot's requests spend outside the server, minus the time the batch will take with one more request in it, minus a margin; this is its slack (Equation 3). The open batch waits for the other robots that the replica serves until each has a request in it, or until its first request can wait no longer, and then runs with the requests that have arrived. No one declares a deadline: it follows from the policy's declared parameters and what the robot reports with each request.

The design has three further details.

- *The wait is counted from the request's arrival,* not from the robot's previous response. A robot sends requests on its own clock, k steps after its previous request, so holding its response does not delay its next request; a wait counted from its previous response would grow in every round in which it was held.
- *Robots are not forced into lockstep.* Robots arrive at arbitrary times. Those that arrive within the first robot's slack share its batch, and the rest go into the next one. The only prediction made about a robot that has not yet arrived is the earliest time of its next request: one period after its last.
- *Each robot is scheduled by its own timing.* A robot farther away than a batch leaves time for is served first, separately, in a batch that fits its own timing, and no batch waits for it; a distant robot neither delays the nearby ones nor is placed in a batch that makes it later.

Implementation. With each request, the robot reports the time its requests spend outside the server and how much of its current chunk remained when it sent the request. The first covers upload, download and the client's encoding and decoding: it is the request latency minus the server's handling time, at the value that only the slowest 10% of its recent requests exceed. The server bounds both values; a robot that reports neither is assumed to have a network delay of 100 ms and its policy's declared buffer. The deadline by which a request must leave the server is its arrival time plus the remaining buffer minus its time outside the server, or the moment the robot runs out of actions, whichever is earlier. The batch time is more than the forward pass alone: once eight requests of a given batch size have been served, it is the measured server time for batches of that size, and before that the forward pass plus the server's own handling time measured so far; the margin is 50 ms. When the pending requests cannot all be served in time in one batch, the splitter sorts them by deadline and dispatches first, separately, the ones that would be late: those that cannot be served in time even alone are dispatched together immediately, and otherwise the earliest goes with as many of the following requests as its own deadline allows. A robot's period and its jitter are smoothed over its requests (a weight of 0.2 for the newest interval), and a robot not heard from for four of its periods is no longer waited for.

Measurements. The controlled measurement is in Section 3.5: with GS Inference, an A100 served twelve $\pi_{0.5}$ robots in time, against at least six answered one by one, and on an L4 it doubled the GR00T robots served in time (Table 4). Before the wait used measured values, four robots at 20 Hz, each with chunks of $N = 15$ actions and a new request every $k = 8$ steps, on a stand-in model on a CPU, received many of their responses late, because the batch wait, the time in the server and the time outside it together exceeded the buffer. In a simulation of one replica, a nearby robot alongside a distant one was served in time more often when the distant robot was dispatched separately, in both of two seeds. A replica met its deadlines clearly less often at 95% load than at 88%, which is why replicas are filled only up to a load limit. The late rate after the change, in the setting of the four robots above, has not been measured.

When it helps. When several robots share a replica and the policy leaves a buffer.

B.1.5 How many robots fit

Motivation. The same quantities answer a second question: how many robots can one model replica serve before responses start arriving late? Taking robots until requests queue up finds the answer only by failing.

Design. We call the number of robots a replica can serve while meeting each robot's deadline its *seats*. Each robot is assigned the largest declared batch size whose server time, plus that robot's own time outside the server, plus the margin, fits within its buffer. Robots assigned the same batch size are answered together in passes of up to that size; all passes together may occupy no more than 90% of the time between a robot's requests; and a replica seats at most 64 robots regardless of how fast its forward pass is. Robots are accepted nearest first, so the farthest robot is the first one a replica cannot accept; that robot is notified, and the platform moves it to another replica.

Implementation. The load limit was determined by measurement: periodic robots offset themselves and rarely wait while their batches complete within one cycle, and pile up once they do not. A replica responded in time clearly less often at 95% load than at 88%. A robot is moved only after it has exceeded a replica's seats for 10 s, so that one slow request does not move it; once moved, the replica does not take it back for 60 s; and a robot's seat is kept for 3 s between two of its episodes. A robot whose response could not arrive in time is notified that its buffer is too short, at most once a minute.

Measurements. One L4 kept at most two $\pi_{0.5}$ robots on time at 15 Hz, and one A100-40GB twelve, with GS Inference (Table 4).

When it helps. When robots at different distances share a replica: when every robot's batch size was set by the farthest robot's latency, one distant robot lowered the batch size of every nearby one, and with it the replica's seats.

B.2 Network: The Connection Between a Robot and Its GPU

At the network layer, every request pays the network delay at least once, and a robot's traffic pattern makes it pay more. Its uploads are periodic, with an *idle period* between them in which the connection sends nothing. After each idle period Linux shrinks the connection's congestion window, the amount of data it may send before it must wait for an acknowledgement, so with the default setting every upload after a pause took one extra network delay, at every network delay we set but the longest (Table 10). Responses written in several pieces, uploads larger than the policy needs, and a client that opens a second connection add further network delays or bytes. The components in this section remove them; all are released in `sequence_ai.network` except the client techniques of Section 4.6. The components are independent; Section 4.5 describes the single call that applies the client-side components to one connection.

B.2.1 Preserving the congestion window across idle periods

Motivation. A robot's uploads are periodic: an observation, then nothing until the next request a fraction of a second later. TCP was tuned for the opposite case. TCP resends a packet that is not acknowledged within its retransmission timeout (RTO). On Linux, a connection that has sent nothing for longer than this timeout is assumed to have stale information about the path, and its congestion window is reduced before it sends again. If the reduced window cannot hold the next observation, the observation is sent in two flights instead of one, and the request pays one extra network delay every time.

Design. The kernel checks for idleness each time data is added to a connection's send queue. With `net.ipv4.tcp_slow_start_after_idle` at 1 (the default), no packet in flight, and more than one RTO elapsed since the last send, the window is reduced: halved once for every RTO that elapsed, down to the initial window of 10

packets. The RTO is at least the network delay plus 200 ms. A robot is therefore affected when three conditions hold together: the gap between its uploads exceeds the network delay plus 200 ms; the upload is larger than the reduced window can hold (10 packets is about 14 KB); and the connection uses a congestion control algorithm that applies the reduction. The last condition allows a mitigation: BBR, a congestion control algorithm that paces its sending by the measured bandwidth and network delay, manages the window itself and skips both the reduction after idle periods and the decay of an underused window.

There are three mitigations, and they differ in who can apply them.

- Setting the kernel option to 0 preserves the window across idle periods. This requires root on the robot.
- Using BBR on the connection. An unprivileged process may select BBR for its own socket, but only if the kernel module is already loaded, which is not the default on Ubuntu.
- Keeping the connection from ever going idle by sending a small request about every 100 ms while the robot is otherwise idle. This keeps the time since the last send below the RTO, so the reduction never triggers.

An underused window also decays over time, and pings do not fill it. In the experiment below this decay did not occur in any group with pings; every acknowledgement of new data refreshes the timestamp from which the decay is measured, and the pings' own acknowledgements arrive every 100 ms. The package documentation warns that a window the pings do not fill may shrink, and that BBR or the kernel option preserves it; the condition under which it shrinks has not been identified.

Implementation. The `window` component is 223 lines of standard-library Python. `bbr_options()` returns the socket option that selects BBR if this machine allows a socket to use it, and first tests it on a throwaway socket: an HTTP client applies its socket options while connecting, and an option it cannot set makes the connect fail. `keepalive_options(idle_s=20, every_s=5, probes=3)` returns TCP keepalive options, each tested the same way; they do not preserve the window, and exist so that a connection silently dropped by a home router is detected before the next observation is sent on it. `socket_options()` returns both lists in the form HTTP client libraries accept, and `keep(sock)` applies BBR to a socket the caller already holds. `Warm(ping, every_s=0.1, for_s=60.0)` implements the third mitigation: the caller calls `used()` after each send, and a thread calls `ping()` whenever the connection has been idle for `every_s`, until `for_s` after the last use; a ping that raises an exception stops the pinging, since the connection is gone. `check(size)` changes nothing on the machine; for an upload over 256 KB on a machine that reduces an idle window and has no BBR, it issues a warning once per process that gives the two commands that fix the problem. On macOS the window did not shrink after a pause in our checks, so none of this was needed there.

Experiment. Two small cloud VMs, with Linux kernels 7.0 and 6.8, were the clients, and a stand-in policy on a CPU was the server; delay was added to the clients' outgoing traffic. Each group opened a new connection, warmed it up with one large transfer, and then sent one 24 KB request (19 packets with the platform's client) every 0.53 s. Table 20 gives, for each group, the median time the server spent receiving a request and how many of the 40 steady-state requests paid at least one extra network delay.

Network delay	≈21 ms		≈101 ms		≈241 ms	
Client's Linux kernel	7.0	6.8	7.0	6.8	7.0	6.8
Default	21.5 / 40	21.5 / 40	0.4 / 0	0.6 / 0	0.4 / 1	0.4 / 0
Kernel option set to 0	0.5 / 0	0.7 / 0	0.4 / 0	0.6 / 0	0.4 / 0	0.4 / 0
Client sends a small request every ≈100 ms	1.4 / 0	1.1 / 0	0.4 / 0	0.4 / 0	0.4 / 0	0.4 / 0
BBR as the system default	0.7 / 0	0.7 / 0	0.6 / 0	0.5 / 0	3.0 / 0	3.0 / 0
Default, 1% loss throughout	22.1 / 39	25.0 / 40	100.6 / 39	101.9 / 40	242.8 / 38	223.6 / 38
Default, 1% loss early, then none	20.2 / 40	19.3 / 40	100.4 / 40	83.6 / 40	239.6 / 20	0.6 / 0
the same, kernel option set to 0	0.5 / 1	0.7 / 0	0.4 / 0	0.6 / 6	0.4 / 0	0.6 / 0
the same, client's small requests	3.8 / 1	2.5 / 0	0.4 / 1	0.4 / 0	0.4 / 1	0.4 / 1
Default, request size varying ±8%				1.3 / 19		0.4 / 10
the same, client's small requests				0.4 / 0		0.6 / 0

Table 20 | The idle-window experiment: a 24 KB request (19 packets) every 0.53 s on a warmed-up connection, from a cloud VM to a stand-in server, with delay added; 40 steady-state requests per cell. Each cell gives two numbers. First, the median time in ms from the first to the last byte of a request at the server: a few ms or less when the request arrived in one flight of packets, about one network delay when it needed a second flight. Second, how many of the 40 requests paid that extra network delay, counted when the time exceeded half the network delay. Columns give the network delay and the client's Linux kernel; in some groups the 6.8 client's base delay was shorter, so they ran at about 84 and 224 ms. *Kernel option set to 0*: `net.ipv4.tcp_slow_start_after_idle=0`, which keeps the window across idle periods. *Client sends a small request*: a request to the server about every 100 ms while the client is otherwise idle. *BBR*: a congestion control that does not reduce the window after an idle period. *Loss*: 1% of packets dropped, throughout the run or only during the warm-up and the first five requests. A blank cell was not run: the varying sizes ran on the 6.8 client only, at the two longer delays.

Why the network delay matters. The request period was 0.53 s. At the shortest network delay the RTO is short enough that the window is halved twice in that time, down to the floor of 10 packets, and a 19-packet request does not fit. At the two longer network delays the RTO is longer, the window is halved once, and the halved window holds a 19-packet request exactly. This is why a slightly larger request, or a window reduced by a loss, brings back the extra network delay at those distances.

When it helps. At the shortest network delay on a clean link, every request paid an extra network delay by default, and none did with any of the three mitigations, on both kernels. After a loss the extra network delay appears at every distance: with 1% loss throughout, nearly every request paid it by default. When request sizes vary, up to half the requests paid it at the two longer network delays, and none did with the ping.

The same upload with the released tool. The released tool (`bench/client.py`) counts the idle period from the response: when the acknowledgement of one upload arrives, it waits 0.53 s and uploads again, as a robot does when it waits for a response before acting. The connection has then sent nothing for the idle period plus one network delay. Between two small cloud VMs 12 ms apart (kernel 7.0, with CUBIC, Linux's default congestion control), with delay added, the 24 KB upload took two network delays every time at all but the longest network delay, where it took one (Table 10). With the server pinging, the client sending a byte every 100 ms, BBR on the client socket, or the kernel option at 0, no upload took more than one network delay at any network delay. This matches the mechanism. Before each default upload the tool recorded the window and the idle time, which it reads from the kernel as the time since the connection last sent data; Table 21 sets the idle time against the RTO computed from Linux's rule. Up to about 100 ms the idle time spans two timeouts, so the window is halved twice, down to its floor of 10 packets, and the upload no longer fits: it is sent in two flights. At 242 ms it spans one timeout, the window is halved once, and the halved window still holds the upload, which is sent at once. At about 101 ms this is the opposite of Table 20, and both results hold. There, the 0.53 s ran from the start of one upload to the start of the next, an idle time that spans one timeout; here it runs from the response, and with the network delay added it spans two. Whether a robot at that distance pays therefore depends on where its pause starts: its pause, its network delay and its upload size decide it together. A robot that sends a request, waits for the response and executes part of it before sending the next request is in the second case, and its idle period also includes the model's inference time; openpi's example client for one of its robots works this way. The window before them had been reduced by the losses rather than by the idle period; a ping prevents the reduction after a pause but does not grow the window. With BBR on the client socket, run at 22 ms only, no upload took more than one network delay.

				Window (packets)		
Network delay (ms)	Idle (ms)	Timeout (ms)	Idle, in timeouts	Before	Left	Upload, in network delays
12.4	542	212	2.6	28	10	1.99
21.4	552	221	2.5	28	10	2.00
40.5	571	241	2.4	28	10	1.99
65.5	596	266	2.2	28	10	2.00
81.3	612	281	2.2	28	10	2.00
100.7	631	301	2.1	28	10	2.00
242.2	773	442	1.7	36	18	1.00

Table 21 | Why the default upload of Table 10 took two network delays at every delay we set but the longest: medians over its 40 uploads, from what the tool recorded before each one. *Network delay*: the path's, as in Table 10. *Idle*: how long the connection had sent nothing before the upload, as the kernel reported it; it is the tool's 0.53 s wait plus about one network delay. *Timeout*: the connection's retransmission timeout (RTO), how long TCP waits for an acknowledgement before it resends a packet; Linux halves an idle connection's window once for each whole timeout it stays idle, but not below 10 packets. *Window*: how many packets the connection may send before it must wait for an acknowledgement; *before*: as the kernel reported it before the upload, which is before the halving, since Linux applies it only when the upload starts; *left*: after the halving, computed by Linux's rule. The upload, as the released tool sends it, was about 18 packets: it fit in a window of 18, and with 10 it needed a second flight. *Upload*: its median time in network delays.

B.2.2 Server-side pings

Motivation. The mitigations of Section 4.3 are applied on the robot, and a robot's software is often the hardest component to change: it may be someone else's client library, or firmware. What reduces the robot's window is the robot's own idleness, and the server can end that idleness.

Design. A WebSocket library answers a ping with a pong automatically. If the server on the GPU sends an application-level ping about every 100 ms (a WebSocket ping, or an HTTP/2 or gRPC PING), the robot's library answers each one, so the robot sends a few bytes in every interval and its kernel never considers the connection idle; neither the robot's program nor its library changes. A WebSocket server sends its next ping one interval after the previous pong arrives, so the robot is idle for the interval plus one network delay. The robot's RTO is about its network delay plus 200 ms; an interval under 200 ms therefore works on a path of any length, and at 100 ms one lost ping still leaves the next one within the RTO.

Implementation. The `ping` component does not implement pinging; it returns the options that existing servers already accept. `ping.websockets_options(every_s=0.1, timeout_s=20.0)` returns the `ping_interval` and `ping_timeout` arguments of `websockets.serve`, and `ping.uvicorn_options()` the corresponding uvicorn arguments. The timeout is long: it is how long a pong may take before the server drops the connection, and a slow link is not a dead one. The uvicorn options were verified with uvicorn 0.54 on three of its WebSocket implementations. A server that answers each HTTP/1.1 request with one response cannot send anything unprompted; there the ping has to come from the robot (Section 4.3).

Measurements between cloud VMs. We used three cloud VMs, one connection per pair, at a near and a far network delay. Table 22 gives the number of network delays an upload took. A cold connection, including the handshake, took more network delays still, and more for larger uploads. The interval that matters is set by the client's timeout: on the near pair, an interval just under the timeout still avoided the extra network delay, one just over it did not, and shorter intervals made no difference. We use 100 ms, so that even after one lost ping the gap stays within the timeout. The ping does not slow requests: on one connection, uploads with pings took as long as uploads sent continuously. In an evaluation with the simulator and the policy 226 ms apart, pinging between requests reduced the server's median time from the first to the last byte of a request from more than one network delay to under 1 ms.

Upload	After 500 ms idle	Server pings every 100 ms	Sent continuously
100 KB	3.0	1.1	1.05
500 KB	3.0	1.1	1.1
1 MB	3.0	1.25	1.3

Table 22 | How long an upload from the client took, in network delays, on a connection already open between cloud VMs, with a near network delay of 11.7 ms and a far one of 69.2 ms. *After 500 ms idle*: by default, each upload sent after the connection had sent nothing for 500 ms. *Server pings every 100 ms*: the same idle time, with the server pinging the client during it. *Sent continuously*: uploads back to back, with no idle time, for comparison.

When it helps. Between wired machines: three network delays became about one for uploads of 100 KB to 1 MB.

B.2.3 One connection, and existing policy clients

Motivation. Combined without care, the client-side components can undo one another. An HTTP/1.1 client that starts a request while another is still waiting for its response opens a second connection for it. So if a ping is still waiting for its answer when a request starts, the request goes on a new connection, with a new handshake and a congestion window that starts small again, which is what the ping is meant to prevent.

Design. `connect(url)` opens one connection, sets the socket options of Section 4.3 on it, and holds a lock between pings and requests: a ping is skipped while a request awaits its response, and a request that starts while a ping awaits its answer waits for it, at most one network delay and only when the two coincide. `act(observation)` encodes the observation as msgpack (Section 4.6), optionally compresses its large arrays first where that reduces latency (Section 4.6), posts it, and returns the response with arrays decoded as arrays. The latency of each request is recorded (Appendix B.2.9).

Implementation. `Link(url, act_path="/act", ping_path="/ping", ping_every_s=0.1, ping_for_s=60.0, compress=False, timeout=10.0)` builds an `httpx` client whose pool is limited to one connection; by default it pings after every 0.1 s of idle time, for up to 60 s after the robot last sent something. The server needs a POST endpoint that reads a msgpack body and any cheap GET endpoint for pings. Compression is off by default because it requires a server that restores the arrays. To decide whether compression pays off, the link must know what its bytes cost, but a request's latency also includes the model. If the server reports its own time in the standard `Server-Timing` header, that time is subtracted; if it does not, a slow model is counted as a slow link, and compression is judged more beneficial than it is. On first use, the link runs two checks that only print advice: whether Wi-Fi power saving is on, and whether the kernel reduces the window after idle periods.

openpi's client. Our adapter, `openpi.keep(policy)`, takes the WebSocket connection that openpi's own client opens, requests BBR and enables TCP keepalive on its socket, and pings the connection while it is idle between two `infer()` calls, waiting up to 5 s for each pong. Nothing in openpi is replaced: its frames, its msgpack layout and its server are unchanged. On the server side, the same effect needs one argument, `ping_interval=0.1`, in the call that starts openpi's WebSocket server; this requires a change to openpi's server file, and openpi's server has not been run with that change. The client side was tested against version 0.1.0 of openpi's client package on a local connection.

LeRobot's remote inference. LeRobot 0.7.0 communicates over Zenoh, which opens its own sockets, so their options cannot be set from Python. Zenoh has a setting for how many keep-alive frames a link sends per lease period (10 s by default); `lerobot.keep_alive_file(path)` writes a configuration that sets it to 100, so that an idle link sends a frame every 100 ms in each direction. With it, two idle Zenoh peers on one machine sent a frame about every 100 ms in each direction; without it, they sent one every 1.5 s or less often. The lease itself is unchanged: it is how long a silent peer is waited for before the link is dropped, and a short lease drops a robot whose Wi-Fi stalls briefly.

When it helps. Wherever the mechanisms of Sections 4.3 and 4.4 help; these adapters apply the same mechanisms to connections that the user does not own.

B.2.4 Single-write HTTP responses

Motivation. uvicorn, a widely used Python HTTP server, writes a response in pieces: the status line and headers when the application starts the response, then the body, then the end of the message. On a direct connection the pieces leave together or nearly so. Behind some tunnels and proxies they do not: the first piece is forwarded, and each later piece is held until the client has acknowledged the first, so every request pays one extra network delay.

Design. The `one_write` component gives uvicorn's HTTP protocol a transport that buffers everything written to it in one event-loop iteration and sends it as a single write in the next iteration. A response's start, body and end are written consecutively with nothing awaited in between, so they are sent together, as one TLS record on a TLS server. A streamed response is still sent part by part, each one iteration later.

Implementation. `uvicorn.run(WholeResponse(app), http=one_write_http())`. The transport alone is not sufficient for every application. Middleware that passes a response between tasks, as Starlette's base middleware class does, puts the start and the body in different event-loop iterations and therefore in separate writes again. `WholeResponse` is an outermost wrapper that holds a response's start until its first body part arrives. In one deployment, middleware added after the transport split responses again, and a probe found every request back at two network delays (the second measurement below).

Measurements. Behind one type of tunnel, a GET took about one network delay longer at the median when headers and body were written separately than when they were written together. When later middleware split responses again, requests through the same tunnel took two network delays, and less than half that time once the response was held and written as a whole (30 requests each).

When it helps. Behind a tunnel or proxy that forwards a later piece of a response only after the earlier piece has been acknowledged; there it saves one network delay on every response.

B.2.5 Compression only when it reduces latency

Motivation. An observation can be megabytes of raw camera frames, and a robot's uplink may carry only a few megabytes per second. Compressing every array is counterproductive on a fast link, where compression and decompression take longer than the upload time they save, and for data that does not compress.

Design. `compress.Squeeze` decides for each large array or byte string in an observation, by its path in the observation, whether to compress it. It compresses as long as the upload time saved at the uplink's rate exceeds the robot's compression time plus the server's decompression time. The uplink rate is measured from the connection's own uploads. The decision is made per path because compressibility depends on the content: camera frames compress, JPEG bytes never do, and a given path holds the same kind of content every time. On the server, `compressed.expand` restores every compressed item to its original array or bytes before anything else reads the observation, so the policy receives exactly the bytes the robot built.

Implementation. Only items of 16 KiB or more are considered. The codec is zstd at level 1 where it is available on the robot (Python 3.14's standard library, or the `zstandard` package), and zlib at level 1 otherwise. A path whose compression ratio is below 1.05, or for which compression stopped paying off, is sent uncompressed for its next 64 observations and then tried again. Until a connection has measured its own uplink, a rate of 6 MB/s is assumed. A compressed item is sent as a small map containing the codec, the uncompressed size and, for an array, its type and shape; `expand` checks the stated size against a limit before decompressing and against the actual decompressed size, so a map that understates its contents is rejected rather than expanded.

Measurements. We measured what compression saves and what it costs on four real camera frames of $320\times180\times3$: zstd at level 1 nearly halved the bytes, took well under a millisecond to compress and to decompress, and was lossless. These frames were decoded from video and compress better than raw camera frames, so real workloads may compress less. Bytes cost time on the robot's uplink: an upload takes an extra (size $\times$ 8) / (uplink rate), which is 400 ms for 1 MB at 20 Mbit/s.

When it helps. On a slow uplink carrying raw arrays (uncompressed frames, depth, tactile data).

B.2.6 Resizing frames before upload

Motivation. Cameras produce more pixels than policies use. A frame resized on the robot is a smaller upload, and the robot knows what input its policy was trained on.

Design. `frames.resize_image(frame, width, height, mode=..., filter=...)` resizes a frame to the policy's input, and `frames.encode` encodes a frame as JPEG. Nothing has a default: the caller specifies the mode (padded or stretched) and the resampling filter. A default filter once caused frames resampled one way to reach a policy trained on frames resampled another way; the filter is part of what a policy was trained on.

Implementation. The padded mode is adapted from openpi's client and reproduces its padded resize byte for byte, so a frame resized here is identical to the frame openpi's own client would send. The default JPEG quality is 95. The image library is imported only when a frame is actually resized or encoded, and a frame's size is read from its header without it.

Measurements. We asked how much JPEG changes $\pi_{0.5}$'s actions, on 50 real DROID observations with the policy's sampling noise fixed. In that earlier measurement, two identical requests without fixed noise gave chunks that differed by about as much as the actions themselves.

When it helps. When cameras produce larger images than the model's input: one robot in our data sent two 1,280×720 frames to a policy from a family whose templates take 224×224 images.

B.2.7 Arrays as raw bytes

Motivation. JSON encodes bytes as base64, a third larger on the wire, and arrays as lists of numbers.

Design. The `msgpack_io` component encodes observations and responses as msgpack, with bytes as bytes and numeric arrays as their raw data. Two ecosystems, msgpack-numpy and openpi, send arrays over msgpack in different layouts; the component writes the first and reads both, so a server that uses it accepts either client. It rejects anything whose decoding could execute code: object arrays (which msgpack-numpy writes with pickle), structured types, and every msgpack extension type.

Implementation. It is written in plain Python, so that neither side needs to install a msgpack package; on a machine without numpy, arrays are decoded as the nested lists that the same value would be in JSON. Nesting is limited to 64 levels.

When it helps. For large raw arrays, which JSON encodes as base64, a third larger.

B.2.8 Sending observations as they are produced

Motivation. A policy that uses several frames, or video, needs many frames before it can start. If they are all uploaded with the request, the request waits for the whole upload.

Design. The platform's own client uploads each observation as soon as it is produced, on the same connection, with up to eight uploads on their way at once, so that only the newest frame is on the request's critical path. A camera frame marked as video is coded against the frames before it as low-latency H.264, with a keyframe every second; when the *worker*, the server process on the GPU machine that runs the policy, refuses a frame, it drops the frames up to the next keyframe and the client forces one, so about one network delay of frames is lost and the episode goes on. The same client connects directly to the machine that runs the policy, and on each new connection it first sends 1 MB to open the congestion window. These techniques are part of the client `sequence-ai` uses to reach the hosted service, not of the components for a user's own server.

Measurements. We timed frames sent as video, from capture until the worker had decoded them, between two containers on one machine with an emulated network delay and no model. Without loss, at an 80 ms network delay, a stream that sends the frames on one TCP connection without waiting for responses was as fast as WebRTC, and took about half as long as the earlier client, which sent its uploads one by one, each after the previous one was answered. Sending up to eight uploads at once instead of one by one nearly halved the median at an 80 ms network delay without loss, and also halved it at a 200 ms network delay and 30 Hz.

When it helps. When the observation is large and the link is clean or lightly lossy.

B.2.9 Measurement tools

Four components measure rather than change anything. Whether a component above helps a given robot depends on that robot's link, which these tools measure.

Link statistics from the server's kernel. `tcp_link` reads, for each client connection, what the server's own kernel knows: the smoothed network delay and its variance, retransmitted segments, the congestion window and the delivery rate. It also times each upload from the arrival of the packets that carried its header to the arrival of those that carried

the end of its body, which is about one network delay when the upload was sent in two flights. Because these are arrival times, not the times the server read the bytes, a busy server does not mistake a two-flight upload for one. The server is the right place to read these values: a client's operating system may not expose these counters to an unprivileged process, and the server reads every client the same way. A client at a private address is a tunnel or proxy in front of the server, and its numbers are labeled as that hop's, not the robot's.

Latency statistics. `latency_stats` keeps counters, gauges and, for each latency metric, its latest 256 samples, and reads them as one snapshot. Percentiles use the nearest-rank method, so each one is a measured value. A percentile is reported only once the window holds enough samples for it: the 99th percentile of twenty samples would simply be the largest of them, so it waits for 100. All percentiles in this report that we computed ourselves use the nearest-rank method.

Wi-Fi power saving. With power saving on, a Wi-Fi adapter sleeps between its access point's beacons, and incoming data, including a policy's response, waits until it wakes. `wifi_power` finds the interface of the default route, checks whether it is wireless and whether power saving is on, and if so issues a Python warning with the command that disables it; it changes nothing.

Link measurement tool. `bench/server.py` and `bench/client.py` measure one TCP connection between the GPU's location and the robot's location, using only the standard library. On that one connection, the client times: an empty frame (the baseline network delay); a frame after an idle period; the same with the server pinging during the idle period; the same with the client sending a byte during the idle period; back-to-back frames; frames at a fixed rate without waiting for acknowledgements; and, optionally, a frame written in two parts with Nagle's algorithm enabled. It reports each as a multiple of the baseline network delay and records the kernel's connection state around each upload. It can request BBR on its socket and reports which congestion control the connection uses, and it can warm the window with one large upload first, as a client that widens its window on connect does. All times come from one machine's monotonic clock; the clocks of two machines are never compared.

B.3 The Platform: Episodes, GPUs and Locations

Obtaining GPUs. Each robot is given the closest GPU that is available. The same mechanism obtains machines for deployed policies and for users who run their own server (Appendix A).

At the routing layer, two problems remain that a library cannot address. The network delay, paid on every request, depends on where the robot's GPU is (Section 5.1). When a policy keeps state across an episode, the whole episode must be served by one *replica*, a copy of the policy running on one GPU. Without a *standby*, a second replica kept ready to continue the episode, the released client reconnects at once when its connection is reset, but when its replica goes silent it waits until the request's deadline has passed: tens of seconds with a 30 s deadline and minutes with the platform's default (Section 5.2). The components in this section are part of the platform and are not released.

B.3.1 Pinning an episode to one replica, with a standby

Motivation. Some policies keep state across the requests of an episode: a history of frames, a plan, a cache on the GPU. Such an episode cannot be served by whichever replica is free, and when its replica or its connection fails, an episode that starts over has lost the task.

Design. The router serves a request from a free replica if the policy is stateless, from the episode's own replica if the policy is stateful, and from the robot's place on a shared replica if the policy answers several robots together in one pass. A replica is released from an episode that has been idle past its timeout, or when the same robot starts its next episode. An episode whose replica died is not silently restarted: for an hour afterwards its requests are rejected with a response stating that the episode was lost, so the robot's program can decide what to do. To survive a failure instead, the episode is stored outside the machine as it runs. Each frame a robot sends and each response are appended to an episode log, and periodically, off the critical path, the open part of the log and the policy's own state for the episode are written to storage. The robot holds a second, idle connection to a standby replica. When the main connection fails, the next request goes directly to the standby, without passing through the platform's *gateway*, the server that tells a robot which GPU machine to connect to; the standby reads the episode back and continues it.

Implementation. The log is saved every 2 s when the episode has a standby and every 300 s when it is kept for storage only; the last 30 segments are kept, about a minute at 2 s. The policy's state is saved key by key, and a key is uploaded only when its bytes have changed since the last save, as determined by its hash: most of a policy's state (a

model's settings, a task's plan) does not change from one save to the next. Each key alternates between two storage locations, and an index naming the current one is written last, so a machine that stops between writing a key and writing the index leaves the previous save intact. An episode's saved state is limited to 16 MiB; large data that belongs to the machine alone, such as a frame history or a GPU cache, is declared local and is not saved. Saving is done by the server process and never by the policy's process, which holds no storage credentials.

Measurements. We timed how long the released client waits for its next answer after a failure (Section 5.2), against a stand-in gateway and worker on one machine with no model. Every trial recovered. When the worker went silent, the standby answered in under a second, while reconnecting through the gateway waited out the request's deadline (Table 14). In a later check on a laptop, where the worker's connection resets did reach the robot, reconnecting took a fraction of a second.

When it helps. For failures of one connection or one replica.

B.4 Program Structure

A server that sees only requests cannot tell how much time each robot has left. The calls in this section let a program stream observations, request the next chunk while it keeps moving, declare its timing so that the server can derive each request's slack, and run a slower planner without blocking control.

B.4.1 Observe and act, with declared timing

Motivation. A server that sees only requests cannot distinguish an urgent request from one that could wait half a second. The information exists in the policy's chunk parameters and in the robot's control loop.

Design. A robot program calls `observe(inputs)` to send an observation as soon as it is available and `act(observation)` to request the next chunk; `end_episode()` ends an episode and `label(success, notes)` records its outcome. The policy's author declares the control rate, the number of valid actions N and the step k at which the robot sends the next request (Section 2). From these, the server derives each request's slack (Section 3.5) and how many robots a replica can accept (Section 3.6); a policy may instead declare a fixed time that each request may wait to be answered together with others, and its robots then stop whenever their actions run out before the response arrives.

Implementation. The same declaration drives the benchmark harness. A benchmark episode runs on a virtual timeline that reproduces the order of observations and requests a robot would produce: each chunk executes its first N actions; the request for the next chunk is sent right after the observation returned by its k-th action; the environment is paused until the response arrives. The switch to a new chunk happens at the N-th action and never "when the response happens to arrive", so the same policy produces the same episode regardless of how fast its machines respond, and a benchmark runs as fast as its simulator and policy allow. Waiting costs the episode's clock nothing: an episode's duration is its step count divided by the control rate. Every observation, every complete response with its latency, every plan and the outcome are written to the episode log.

When it helps. The server learns each request's time budget without the robot stating a deadline, and a benchmark replays the same order of observations and requests on every run.

B.4.2 Planning

Motivation. A vision-language model that selects the next subtask takes hundreds of milliseconds to seconds. A control loop cannot wait for it.

Design. A policy may have a `plan` method. It runs on its own timer: a plan starts a fixed interval after the previous one started, only if the robot has sent something since, and never while another plan is running, so a plan that overruns its next tick is followed by the next one as soon as it returns. Its output is appended to the episode together with the index of the latest observation at that moment; requests read the newest plan and never wait for one. A failed plan changes nothing for any robot: the error is logged, the robots keep their newest plan, and the next tick retries.

Measurements. In one program run on the platform, a plan ran every 2 s and took far longer than the policy's requests that ran alongside it.

When it helps. A slow model can steer a fast one without slowing it down.

B.4.3 Human handoff and teleoperation

Design. A plan may request a human operator instead of selecting a subtask; the operator then controls the robot by teleoperation. A robot under teleoperation is not served by any policy machine, so it keeps the episode log itself: each step it observed and each of the operator's chunks that it executed, written to a file on the robot's own machine; the platform keeps none of it. An episode is labeled as a whole, with success or failure and a note.

When it helps. When a plan decides that a person should take over.

Authorship

Yongzhe Wang, General Sequences.