



Serving Vision-Language Models: Image Tokens and GPU Memory

Plan and build private AI compute with GPU Smith. We connect workload requirements with GPU hardware, networking, deployment and operating decisions so your infrastructure fits the work it must perform.

Published by GPU Smith · 2026-10-29 · vision language model gpu memory requirements · vlm inference vram · qwen2.5-vl vllm memory · image tokens per image vlm · limit_mm_per_prompt vllm · multimodal llm serving gpu sizing

Original article: <https://gpusmith.com/articles/en/vision-language-model-serving-memory-image-tokens>



In this article

1. Why text-only sizing breaks for VLMs
 2. Anatomy of a VLM request
 3. Counting image tokens
 4. Memory components
 5. Token budget worksheet
 6. vLLM knobs that bound memory
 7. Prefill pressure and TTFT
 8. GPU fit scenarios
 9. Video and multi-image caveats
 10. Operational checklist
 11. Sources
-

Why text-only sizing breaks for VLMs

A vision-language model (VLM) has roughly the weight footprint of its text-only sibling, but each request carries images that become hundreds or thousands of extra tokens before the language model reads the prompt. A text-only calculator sized from prompt length, output length and concurrency therefore undercounts KV cache, prefill compute and time to first token, sometimes by an order of magnitude, and for dynamic-resolution models the error depends on the pixel dimensions users upload.

This guide treats image resolution as a capacity variable: it counts visual tokens for fixed- and dynamic-resolution models, converts them into KV cache bytes, lists the extra memory pools a multimodal engine allocates, and maps the result onto vLLM flags. Flags and defaults were checked against the vLLM **v0.32.0** source (released 2026-10-18, the newest release on [PyPI](#)) and the "latest" docs on 2026-10-28; they change often and should be rechecked against the deployed version.

Anatomy of a VLM request

Most open VLMs have three stages. A vision encoder, typically a Vision Transformer (ViT), cuts the image into patches and treats each as a token; the ViT paper ("An Image is Worth 16x16 Words") showed that "a pure transformer applied directly to sequences of image patches" performs well on image classification ([Dosovitskiy et al., arXiv:2010.11929](#)). A projector (often a small MLP) maps encoder outputs into the language model's embedding space, where they are processed as ordinary prompt tokens and occupy KV cache slots exactly like text.

LLaVA-1.5, the canonical fixed-resolution example, uses "CLIP-ViT-L-336px with an MLP projection" ([Liu et al., arXiv:2310.03744](#)). The 7B checkpoint pairs `openai/clip-vit-large-patch14-336` with an `mlp2x_gelu` projector and a Llama-architecture model of 32 layers, 32 key-value heads and hidden size 4096 ([liuhaotian/llava-v1.5-7b config.json](#)).

Qwen2-VL and Qwen2.5-VL use a ViT with **14x14** patches and then merge each 2x2 group of adjacent patches into one token, so one language-model token represents a 28x28-pixel area ([Qwen2-VL paper](#); [Qwen2.5-VL technical report](#)). The Qwen2-VL encoder has "approximately 675 million parameters" ([Qwen2-VL paper](#)); the Qwen2.5-VL report gives the ViT's shape (32 layers, hidden size 1280) but no parameter total ([Qwen2.5-VL technical report](#)). Summing the `visual.*` tensors in the 7B checkpoint's safetensors headers gives 676,550,144 parameters, encoder plus merger (our count). The encoder is small next to the language model, but its activations scale with image size.

Counting image tokens

Fixed resolution: LLaVA-1.5

For a fixed-resolution encoder the token count is constant. The LLaVA-1.5 processor config sets `patch_size` 14 and `num_additional_image_tokens` 1 ([llava-hf processor_config.json](#)); the latter accounts for a CLS token added by the backbone ([Transformers LLaVA docs](#)). The arithmetic is:

- $336 \text{ px} / 14 \text{ px per patch} = 24 \text{ patches per side}$
- $24 \times 24 = \mathbf{576}$ patch tokens (the CLS token is not passed to the language model; the original checkpoint selects patch features, per its [config.json](#))

This derived value matches the `image_seq_length` default of 576 in Transformers' `LlavaConfig` ([Transformers LLaVA docs](#)). Every image costs 576 tokens whatever its original size, because the processor resizes it to 336x336: capacity planning is easy, but detail in large documents is lost.

Dynamic resolution: Qwen2-VL and Qwen2.5-VL

Qwen2-VL's "Naive Dynamic Resolution" maps different resolutions to different token counts. Its worked example: a 224x224 image "encoded with a ViT using `patch_size=14`, will be compressed to 66 tokens before entering LLM" ([Qwen2-VL paper](#)). Arithmetic: $224/14 = 16$ patches per side, $16 \times 16 = 256$ patches, $/ 4 = 64$ merged tokens, plus `<|vision_start|>` and `<|vision_end|>` special tokens, for 66.

For Qwen2.5-VL the processor resizes the image to multiples of 28 per side, with total pixels between `min_pixels` and `max_pixels` and aspect ratio roughly preserved (the `smart_resize` function in [Transformers' Qwen2-VL image processor](#)). The token count is then $(\text{height} / 28) \times (\text{width} / 28)$. The model card gives "the default range for the number of visual tokens per image" as 4-16384 and an example of `min_pixels = 256*28*28` and `max_pixels = 1280*28*28` for 256-1280 tokens ([Qwen2.5-VL-7B-Instruct model card](#)). The shipped `preprocessor_config.json` sets `min_pixels` to 3136 (4 x 784) and `max_pixels` to **12,845,056** (16,384 x 784) ([Qwen2.5-VL-7B preprocessor_config.json](#)).

One inconsistency matters. The vLLM memory guide comments that the Qwen2.5-VL `max_pixels` "Default is $1280 * 28 * 28$ " ([vLLM docs: Conserving memory](#)). That is the Transformers class default, but the processor overrides it with any `max_pixels` in the checkpoint's config ([Transformers' Qwen2-VL image processor](#)), and the official checkpoint ships 12,845,056. Capacity work should therefore assume up to 16,384 tokens per image unless capped, and confirm by sending a known-size image and reading `prompt_tokens` in the response usage.

Table 1: Derived Qwen2.5-VL tokens per image

Computed with the `smart_resize` logic above (excluding the two vision special tokens). Values are derived, not measured.

Input (W x H)	Tokens, checkpoint max_pixels	Tokens, max_pixels = 1280x28x28
448 x 448	256	256
1024 x 768	999	999
1024 x 1024	1,369	1,225
1920 x 1080	2,691 (resized 1932 x 1092)	1,222 (1316 x 728)
3840 x 2160	10,549	1,222
4032 x 3024 (12 MP photo)	15,552	1,230

A 12 MP photo costs about 27 times the tokens of a LLaVA-1.5 image under the shipped defaults, and about twice as many once `max_pixels` is capped; the cap is the largest single capacity lever.

Memory components

Table 2: Memory pools in a vLLM VLM deployment

Pool	Location	Scales with	Bounded by
Weights (LLM, encoder, projector)	GPU	Parameters x bytes each	Model, quantization
KV cache	GPU	Text + visual tokens in flight	<code>gpu_memory_utilization</code> , <code>max_model_len</code> , <code>max_num_seqs</code>
Encoder activations (peak)	GPU	Largest item processed at once	Profiling; <code>limit_mm_per_prompt</code> hints
Encoder cache	GPU	In-flight vision embeddings	Scheduler budget (not configurable)
GPU video decode reserve	GPU	Operator setting	<code>mm_ipc_gpu_memory_gb</code>
Processor cache	Host RAM	Preprocessed items	<code>mm_processor_cache_gb</code> x process count

Weights

Qwen2.5-VL-7B-Instruct has 8,292,166,656 BF16 parameters ([Hugging Face API: Qwen2.5-VL-7B](#)), or **15.4 GiB** (x 2 bytes / 2³⁰); Qwen2.5-VL-72B-Instruct has 73,410,777,344 ([Hugging Face API: Qwen2.5-VL-72B](#)), or **136.7 GiB**. Both totals include the vision encoder.

KV cache per visual token

KV bytes per token = 2 (K and V) x layers x KV heads x head dimension x bytes per element. With BF16 cache (2 bytes):

- Qwen2.5-VL-7B: 28 layers, 4 KV heads, head dimension 3584 / 28 = 128 ([config.json](#)). $2 \times 28 \times 4 \times 128 \times 2 = 57,344$ bytes = **56 KiB per token**.
- Qwen2.5-VL-72B: 80 layers, 8 KV heads, head dimension 8192 / 64 = 128 ([config.json](#)). $2 \times 80 \times 8 \times 128 \times 2 = 327,680$ bytes = **320 KiB per token**.
- LLaVA-1.5-7B: 32 layers, 32 KV heads (no grouped-query attention), head dimension 128 ([config.json](#)). $2 \times 32 \times 32 \times 128 \times 2 = 524,288$ bytes = **512 KiB per token**.

Per image: LLaVA-1.5-7B, 576×512 KiB = 288 MiB; Qwen2.5-VL-7B, $1,369 \times 56$ KiB $\approx$ 74.9 MiB at 1024x1024 and $10,549 \times 56$ KiB $\approx$ 576.9 MiB for an uncapped 4K frame; Qwen2.5-VL-72B, $10,549 \times 320$ KiB $\approx$ 3.2 GiB for the same frame. Attention layout matters as much as token count: LLaVA-1.5's full multi-head attention makes its 576 tokens cost as much KV as a roughly 4-megapixel image on Qwen2.5-VL-7B. Block allocation details are in the [KV cache memory calculator](#) and the [vLLM KV cache capacity planner](#).

Encoder activations and profiling

At startup vLLM profiles peak memory with dummy inputs and gives the remainder (within `gpu_memory_utilization`, default **0.92** in v0.32.0 source ([vLLM cache.py](#))) to the KV cache, logged as "Available KV cache memory" ([vLLM gpu_worker.py](#)). For a VLM the dummy inputs include maximum-size images, so a model accepting 16,384-token images reserves activations for 65,536 pre-merge patches through a 32-layer ViT. Qwen2.5-VL's window attention means "computational cost scales linearly with the number of patches", but "only four layers employ full self-attention" ([Qwen2.5-VL technical report](#)). `--skip-mm-profiling` skips this step and "shifts the responsibility to users" for estimating encoder peak memory ([vLLM multimodal.py](#)).

Encoder cache

vLLM holds encoder outputs in an encoder cache, sized in embeddings, so an image's embeddings can be consumed across chunked-prefill steps and shared between requests carrying the same image ([vLLM encoder_cache_manager.py](#)). Its size is the larger of `max_num_batched_tokens` and the largest per-item token count, and "is not currently configurable" ([vLLM scheduler.py](#); [encoder_cache_manager.py](#)). Each Qwen2.5-VL-7B embedding has the language model's width (`out_hidden_size` 3584 in [config.json](#)), so a 16,384-embedding cache is $16,384 \times 3,584 \times 2$ bytes = **112 MiB** (derived). It is modest, but the docs warn that encoder cache size "is determined by the actual inputs at runtime and is not limited by these hints" ([vLLM docs: Conserving memory](#)).

Processor cache (host RAM)

`mm_processor_cache_gb` (default **4 GiB**) avoids reprocessing repeated inputs ([vLLM engine args](#)). The source notes it "is duplicated for each API process and engine core process", for a total of `mm_processor_cache_gb * (api_server_count + data_parallel_size)` ([vLLM multimodal.py](#)). One API server at DP=1 therefore uses 8 GiB of host RAM by default.

Token budget worksheet

The worksheet multiplies tokens per image (Table 1), images per request and concurrent sequences.

1. Tokens per request = (images x (tokens per image + 2 special tokens)) + text prompt tokens + output tokens.
2. KV tokens in flight = tokens per request x concurrent sequences.
3. KV bytes = KV tokens x KV bytes per token.

Illustrative workload, Qwen2.5-VL-7B: each request carries four 12 MP phone photos, a 300-token text prompt and up to 500 output tokens; 32 requests run concurrently.

- Uncapped (checkpoint max_pixels): $4 \times (15,552 + 2) + 300 + 500 = \mathbf{63,016 \text{ tokens per request}}$. $\times 32 = 2,016,512 \text{ tokens}$. $\times 56 \text{ KiB} = \mathbf{\approx 107.7 \text{ GiB}}$ of KV cache.
- Capped at max_pixels = 1280*28*28: $4 \times (1,230 + 2) + 800 = \mathbf{5,728 \text{ tokens per request}}$. $\times 32 = 183,296 \text{ tokens}$. $\times 56 \text{ KiB} = \mathbf{\approx 9.8 \text{ GiB}}$.

The same model, prompt and concurrency differ by about 11x in KV demand purely through the resolution cap. The uncapped request also exceeds the 32,768-token context that the model card says the shipped config targets without YaRN ([Qwen2.5-VL-7B-Instruct model card](#)). vLLM will not reject it by default, because it derives max_model_len from max_position_embeddings, which is 128,000 in [config.json](#); quality past 32K is simply unvalidated. Set --max-model-len 32768 or enable YaRN deliberately. The operator should run the worksheet on 50th and 99th percentile image sizes from production logs, not one "typical" image.

vLLM knobs that bound memory

Table 3: Multimodal flags

Defaults per the [vLLM engine args](#) page and [multimodal.py](#) at tag v0.32.0.

Flag	Default	Effect on memory
--limit-mm-per-prompt	999 per modality	Caps items per prompt; shrinks the profiled dummy batch
--mm-processor-kwags	{}	Forwards e.g. max_pixels to the processor; caps real token counts
--mm-processor-cache-gb	4	Host RAM processor cache, per process
--mm-ipc-gpu-memory-gb	0	GPU reserve for frontend video decoding, taken from KV cache memory
--mm-encoder-tp-mode	weights	data replicates encoder weights per rank, splits images
--skip-mm-profiling	False	Faster startup; operator must estimate encoder peak memory
--language-model-only	False	Sets all modality limits to 0

The vLLM memory guide notes that fewer "multi-modal items per prompt" reduce "the memory footprint of the model", and that a limit of zero disables a modality (vLLM docs: [Conserving memory](#)); the server CLI form is `--limit-mm-per-prompt.image 2` (vLLM docs: [Multimodal inputs](#)). The per-modality form also accepts size hints, e.g. `{"image": {"count": 5, "width": 512, "height": 512}}`. These hints "affect memory profiling only" and "do not change how inputs are actually processed at inference time" (vLLM docs: [Conserving memory](#)). A 512x512 hint frees activation reserve at startup, but uncapped 12 MP uploads are still processed at full token count, so the hint and the processor cap should be set together to the same effective size.

Per-request processor overrides are rejected by default on the server; enabling them requires `--trust-request-mm-kwarg`s (vLLM docs: [Multimodal inputs](#)). That flag lets tenants raise their own resolution. Transformers now deprecates call-time `min_pixels/max_pixels` in favor of a size dictionary (Transformers' [Qwen2-VL image processor](#)), so the cap's spelling may change.

Prefill pressure and TTFT

Visual tokens are prompt tokens, so they inflate prefill: one uncapped 4K frame on Qwen2.5-VL-7B adds over 10,000 prompt tokens plus the ViT pass. Two latency consequences follow (metrics as defined in [TTFT and goodput metrics](#)).

First, chunked prefill spreads a long multimodal prompt over several steps bounded by `max_num_batched_tokens`, protecting other sequences' inter-token latency but lengthening the image request's TTFT. The scheduler option `disable_chunked_mm_input` avoids splitting a multimodal item, and errors if the largest item exceeds `max_num_batched_tokens` (vLLM [scheduler.py](#); [encoder_cache_manager.py](#)). Second, the per-step encoder compute budget is also the larger of `max_num_batched_tokens` and the largest item, so one very large image can monopolize a step. Capping resolution reduces tail TTFT as well as memory.

GPU fit scenarios

Table 4 applies the worksheet to single-image chat: one image, 300 text tokens, 500 output tokens. "Capped" uses `max_pixels = 1280*28*28` (1,222 tokens for a 1920x1080 input, Table 1); "4K uncapped" uses 10,549 tokens. Assumptions, all illustrative: card capacity taken as nominal GiB, `gpu_memory_utilization` 0.92, BF16 weights and KV cache, and a flat **1.5 GiB per GPU** for activations, encoder profiling reserve and CUDA graphs. The real reserve should be read from the "Available KV cache memory" log line.

Table 4: Illustrative fit, Qwen2.5-VL in BF16

Model / GPUs	0.92 x total	KV pool	KV tokens	Concurrent, capped (2,024 tok)	Concurrent, 4K (11,351 tok)
7B on 1 x 24 GiB	22.1 GiB	5.1 GiB	≈96,000	≈47	≈8
7B on 1 x 48 GiB	44.2 GiB	≈27.2 GiB	≈510,000	≈250	≈45

Model / GPUs	0.92 x total	KV pool	KV tokens	Concurrent, capped (2,024 tok)	Concurrent, 4K (11,351 tok)
7B on 1 x 80 GiB	73.6 GiB	≈56.7 GiB	≈1,060,000	≈520	≈93
72B on 2 x 80 GiB (TP=2)	147.2 GiB	7.5 GiB	≈24,000	≈12	≈2
72B on 4 x 48 GiB (TP=4)	176.6 GiB	≈33.9 GiB	≈111,000	≈54	≈9
72B on 4 x 80 GiB (TP=4)	294.4 GiB	≈151.7 GiB	≈497,000	≈245	≈43

KV pool = budget - weights - overhead (e.g. 22.1 - 15.4 - 1.5 = 5.1 GiB); KV tokens = pool x 2^{20} / KiB per token (56 or 320). Concurrency figures are KV-bound ceilings; `max_num_seqs`, compute and latency targets usually bind first. The 72B on two 80 GiB cards fits but, at about two concurrent 4K-image requests, is a test rig rather than a serving configuration. Weight quantization moves these numbers considerably; the [inference hardware sizing](#) guide covers that trade-off.

Video and multi-image caveats

Qwen2-VL samples video at two frames per second and caps "the total number of tokens per video to 16384" (Qwen2-VL paper). Qwen2.5-VL groups "two consecutive frames" together and samples at dynamic FPS (Qwen2.5-VL technical report), matching `temporal_patch_size 2` in the processor config (`preprocessor_config.json`). Illustratively, vLLM's profiling example of 32 frames at 640x640 (Conserving memory) resizes to 644x644, or $23 \times 23 = 529$ tokens per two-frame group; 16 groups give **8,464 tokens**, or ≈463 MiB of KV on Qwen2.5-VL-7B. Real counts also depend on video pixel settings such as `total_pixels` (vLLM docs: Multimodal inputs).

GPU-side video decoding requires a positive `--mm-ipc-gpu-memory-gb` "to reserve VRAM for video decoding" (vLLM docs: Multimodal inputs), and that reserve "is carved out of the engine's KV-cache memory" (vLLM multimodal.py). Video token pruning (`--video-pruning-rate`, method `evs` by default in source) reduces tokens, but "enabling video pruning also disables encoder CUDA graphs, since the retained token count becomes data-dependent" (vLLM docs: Multimodal inputs). The KV saving may be partly paid back in encoder latency.

For multi-image prompts the default limit is 999 items per modality (vLLM engine args), so without an explicit `limit_mm_per_prompt` a single client can submit a document's worth of page images in one request.

Operational checklist

Table 5: Pre-production checklist for VLM serving

#	Check	Why
1	Set <code>max_pixels</code> (or equivalent) explicitly	Checkpoint default may allow 16,384 tokens per image
2	Set matching <code>limit_mm_per_prompt</code> count and size hints	Hints only shape profiling
3	Disable unused modalities (<code>{"video": 0}</code>)	Removes video profiling reserve
4	Also downscale at the gateway	Policy enforced before the engine
5	Keep <code>--trust-request-mm-kwarg</code> off	Clients cannot raise resolution
6	Size from production image-size percentiles	Token counts follow pixels, not request counts
7	Check <code>prompt_tokens</code> on known images	Catches processor default changes
8	Record "Available KV cache memory" per version	Profiling reserve shifts with updates
9	Budget host RAM for processor caches	4 GiB x (API servers + DP ranks)
10	Load-test TTFT with mixed traffic	Large images inflate shared prefill

GPU Smith can review a VLM serving configuration against representative traffic.

Sources

1. [An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale](#) - arXiv (Dosovitskiy et al.), ViT patch-token concept (accessed 2026-10-28)
2. [Improved Baselines with Visual Instruction Tuning](#) - arXiv (Liu et al.), LLaVA-1.5 uses CLIP-ViT-L-336px with MLP projection (accessed 2026-10-28)
3. [liuhaotian/llava-v1.5-7b config.json](#) - Hugging Face, vision tower, projector type, language model layers/heads (accessed 2026-10-28)
4. [llava-hf/llava-1.5-7b-hf processor_config.json](#) - Hugging Face, patch_size 14, num_additional_image_tokens, feature strategy (accessed 2026-10-28)
5. [LLaVA model documentation](#) - Hugging Face Transformers, image_seq_length default 576, CLS token handling (accessed 2026-10-28)
6. [Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution](#) - arXiv (Qwen team), Naive Dynamic Resolution, 2x2 merge, 66-token example, 675M encoder, video sampling and 16384-token cap (accessed 2026-10-28)
7. [Qwen2.5-VL Technical Report](#) - arXiv (Qwen team), 14x14 patches, window attention, four full-attention layers, two-frame grouping, dynamic FPS (accessed 2026-10-28)
8. [Qwen2.5-VL-7B-Instruct model card](#) - Hugging Face, 4-16384 token range, min/max_pixels example, 32,768-token config context (accessed 2026-10-28)

9. [Qwen2.5-VL-7B-Instruct processor_config.json](#) - Hugging Face, min_pixels 3136, max_pixels 12845056, temporal_patch_size 2 (accessed 2026-10-28)
10. [Qwen2.5-VL-7B-Instruct config.json](#) - Hugging Face, layers, KV heads, hidden size, out_hidden_size (accessed 2026-10-28)
11. [Qwen2.5-VL-72B-Instruct config.json](#) - Hugging Face, layers, KV heads, hidden size (accessed 2026-10-28)
12. [Hugging Face API: Qwen2.5-VL-7B-Instruct safetensors metadata](#) - Hugging Face, BF16 parameter count 8,292,166,656 (accessed 2026-10-28)
13. [Hugging Face API: Qwen2.5-VL-72B-Instruct safetensors metadata](#) - Hugging Face, BF16 parameter count 73,410,777,344 (accessed 2026-10-28)
14. [image_processing_qwen2_vl.py](#) - Hugging Face Transformers source, smart_resize logic, class defaults, max_pixels override and deprecation (accessed 2026-10-28)
15. [Conserving Memory](#) - vLLM docs (latest), limit_mm_per_prompt, size hints, encoder cache warning, mm_processor_kwargs example (accessed 2026-10-28)
16. [Multimodal Inputs](#) - vLLM docs (latest), CLI limit syntax, mm-ipc-gpu-memory-gb, video pruning, trust_request-mm-kwarg (accessed 2026-10-28)
17. [Engine Arguments](#) - vLLM docs (latest), multimodal flag defaults (accessed 2026-10-28)
18. [vllm/config/multimodal.py](#) - vLLM source (tag v0.32.0), processor cache duplication, mm_ipc reserve, encoder TP mode, skip profiling, language_model_only, pruning method (accessed 2026-10-28)
19. [vllm/config/scheduler.py](#) - vLLM source, encoder cache size and compute budget, disable_chunked_mm_input (accessed 2026-10-28)
20. [vllm/v1/core/encoder_cache_manager.py](#) - vLLM source, encoder cache semantics and budget computation (accessed 2026-10-28)
21. [vllm/config/cache.py](#) - vLLM source, gpu_memory_utilization default 0.92 (accessed 2026-10-28)
22. [vllm/v1/worker/gpu_worker.py](#) - vLLM source, "Available KV cache memory" startup log (accessed 2026-10-28)
23. [vllm on PyPI](#) - Python Package Index, latest release 0.32.0 at time of writing (accessed 2026-10-28)

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. <https://pypi.org/project/vllm>
2. <https://arxiv.org/abs/2010.11929>
3. <https://arxiv.org/abs/2310.03744>
4. <https://huggingface.co/liuhaotian/llava-v1.5-7b/blob/main/config.json>
5. <https://arxiv.org/html/2409.12191>
6. <https://arxiv.org/html/2502.13923>
7. https://huggingface.co/llava-hf/llava-1.5-7b-hf/blob/main/processor_config.json
8. https://huggingface.co/docs/transformers/en/model_doc/llava
9. https://github.com/huggingface/transformers/blob/144ecf007b8b4f2910da104c84a79801db6985a6/src/transformers/models/qwen2_vl/image_processing_qwen2_vl.py

10. <https://huggingface.co/Qwen/Qwen2.5-VL-7B-Instruct>
11. https://huggingface.co/Qwen/Qwen2.5-VL-7B-Instruct/blob/main/preprocessor_config.json
12. https://docs.vllm.ai/en/latest/configuration/conserving_memory
13. <https://huggingface.co/api/models/Qwen/Qwen2.5-VL-7B-Instruct?expand%5B%5D=safetensors>
14. <https://huggingface.co/api/models/Qwen/Qwen2.5-VL-72B-Instruct?expand%5B%5D=safetensors>
15. <https://huggingface.co/Qwen/Qwen2.5-VL-7B-Instruct/blob/main/config.json>
16. <https://huggingface.co/Qwen/Qwen2.5-VL-72B-Instruct/blob/main/config.json>
17. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/config/cache.py>
18. https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/v1/worker/gpu_worker.py
19. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/config/multimodal.py>
20. https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/v1/core/encoder_cache_manager.py
21. <https://github.com/vllm-project/vllm/blob/v0.32.0/vllm/config/scheduler.py>
22. https://docs.vllm.ai/en/latest/configuration/engine_args
23. https://docs.vllm.ai/en/latest/features/multimodal_inputs.html

THE PUBLISHER

About GPU Smith

GPU Smith provides independent engineering and research for private AI infrastructure. We help technical buyers and operators reason about GPU workload sizing, hardware selection, procurement, deployment and inference operations, from the compute system to the networking, power and cooling requirements around it.

Start with the workload

Useful infrastructure decisions begin with the models, data, throughput, latency and operating constraints a team actually has. GPU Smith helps connect those requirements with system design and deployment choices. The goal is a privately controlled AI environment whose capacity and operational demands are understood before a purchasing decision.

Hardware and supplier research

Our public reference library covers GPUs, complete systems and networking components. The server-vendor directory and manufacturing research help teams investigate suppliers, compare options and follow sources behind technical claims. We distinguish manufacturer specifications from measured benchmarks and advertised capabilities from tested configurations. Reference pages are research resources, not a live inventory listing or a binding equipment quote.

Deployment and operations

GPU Smith's engineering scope includes infrastructure integration, networking, power, cooling and the practical operation of inference workloads. Our published articles explain the tradeoffs behind deployment, procurement and ongoing operations so teams can ask better questions and document decisions.

Work with GPU Smith

Explore the [hardware library](#), [GPU references](#), [networking references](#) and [vendor research](#). [Contact GPU Smith](#) with your workloads and deployment constraints to discuss a project and confirm current scope, pricing and availability.

Inclusion of a manufacturer or product in our research does not imply a partnership, certification or endorsement.

Website: <https://gpusmith.com>

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.