DeepSeek-V4.1-Flash: Pushing the Limits of KV Cache Compression

DeepSeek-AI
research@deepseek.com

Abstract

The widespread adoption of long-horizon agents has made model workloads increasingly input-heavy. Although prior work has substantially reduced the cost of long-context computation, prefill remains computationally expensive, and large KV caches continue to strain HBM and SSD capacity and data-transfer bandwidth. Together, these compute, storage, and bandwidth demands constitute the primary bottleneck to further lowering deployment costs. To address this challenge, we introduce DeepSeek-V4.1-Flash, a multimodal Mixture-of-Experts (MoE) model with 552B backbone parameters and support for contexts of up to one million tokens. With its Causal Encoder-Decoder (CED) architecture, the model activates 16B parameters per token during decode but only 8B parameters during prefill, substantially improving cost efficiency for agentic workloads. To push the limits of KV cache compression, DeepSeek-V4.1-Flash combines cross-layer KV cache reuse in Compressed Sparse Attention 2 (CSA2) with FP4 KV caching. These designs reduce its global KV cache footprint (always in HBM) to 890 bytes per token, roughly 1/4 of the corresponding footprint of DeepSeek-V4-Flash. Further, through a dedicated deployment optimization known as SWA Bounded Replay, DeepSeek-V4.1-Flash reduces its persistent KV cache footprint (always on SSD or in host memory) to roughly 1/8 of that of DeepSeek-V4-Flash. Despite its much smaller KV cache footprint, the model delivers substantially better performance than the baseline. In addition, we streamline the DeepSeek-V4 architecture and introduce several efficient architectural extensions. We pretrain DeepSeek-V4.1-Flash on a multimodal corpus comprising 45T tokens and conduct comprehensive post-training, yielding strong performance across diverse text-based and multimodal agentic scenarios. Model checkpoints are available at https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash.

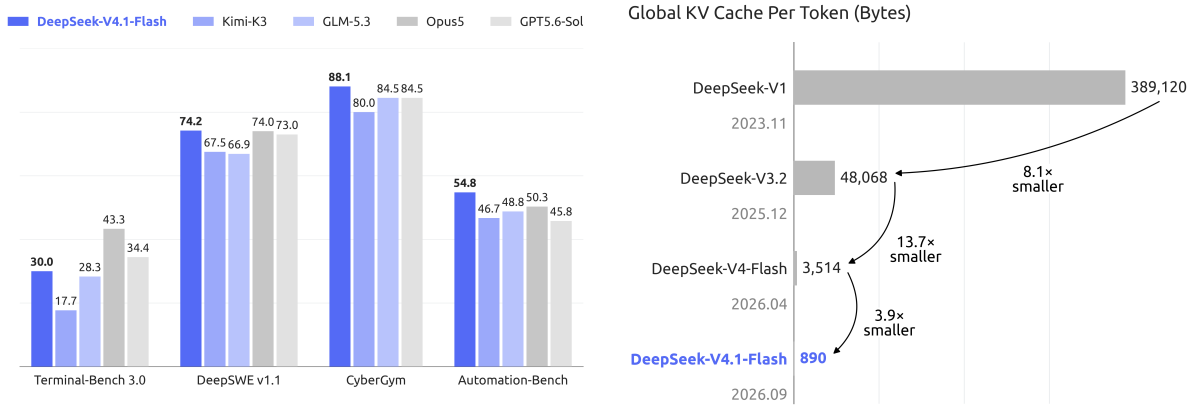

Figure 1 | **(a)** Performance of DeepSeek-V4.1-Flash and its counterparts on agentic benchmarks. **(b)** Global KV cache size per token (in bytes) across generations of DeepSeek models, highlighting DeepSeek's sustained efforts to reduce context memory requirements. DeepSeek-V4.1-Flash achieves approximately 4-fold and 437-fold reductions in per-token global KV cache size relative to DeepSeek-V4-Flash and DeepSeek-V1, respectively.

Contents

1. Introduction

Applications of long-horizon agents have expanded rapidly in recent years, making ultra-long-context processing an increasingly important model workload. Supporting such workloads requires not only efficient long-sequence processing, but also the persistent storage, reuse, and transfer of large KV caches. KV cache management has therefore become a foundational capability for model deployment, while introducing substantial challenges across computation, storage, and communication. Prior advances in sparse attention (DeepSeek-AI, 2025, 2026b) have significantly reduced the computational cost of long-sequence processing, making persistent storage and data movement increasingly prominent bottlenecks.

Specifically, DeepSeek-V4 (DeepSeek-AI, 2026b) combines a global attention branch spanning the full context with local Sliding-Window Attention (SWA). The global branch maintains global KV, comprising main KV and indexer K, while SWA maintains local KV states. For a fixed window size, SWA KV storage is bounded independently of sequence length. For sufficiently long sequences, global KV therefore dominates the **runtime KV** footprint, which is constrained by HBM capacity. In addition, certain KV are persisted for prefix reuse, referred to as **persistent KV** caches, which are constrained by SSD and host memory capacity. I/O and interconnect bandwidth also limit cache migration and loading. Together, these constraints limit serving throughput, increase deployment costs, and ultimately hinder the deployment and adoption of agents over longer task horizons and across broader application scenarios.

Further reducing the KV cache footprint is therefore critical to alleviating storage and communication bottlenecks and lowering the cost of long-context serving. To this end, we develop DeepSeek-V4.1-Flash, a multimodal Mixture-of-Experts (MoE) model designed for more aggressive KV cache compression. DeepSeek-V4.1-Flash has 552B backbone parameters, natively supports multimodal inputs, and accommodates contexts of up to one million tokens. We adopt a Causal Encoder-Decoder (CED) architecture, in which decoder global KV is projected from the final encoder hidden states. This design enables the model to activate 8B parameters per token during prefill and 16B during decode, which is particularly cost-effective for input-heavy agentic scenarios. Despite being considerably larger than DeepSeek-V4-Flash, DeepSeek-V4.1-Flash requires only approximately 1/4 as much runtime KV cache storage and 1/8 as much persistent KV cache storage at the same sequence length. Moreover, DeepSeek-V4.1-Flash delivers better overall performance than DeepSeek-V4-Flash.

This level of KV cache compression is achieved through joint optimizations in model architecture, cache precision, and deployment strategy. Conceptually, DeepSeek-V4 can be viewed as an SWA-based local-processing backbone augmented with compressed global context. This perspective motivates us to focus on simplifying the global branch while largely preserving the local attention design. At the architectural level, we design Compressed Sparse Attention 2 (CSA2), which applies cross-layer reuse to global KV (including main KV and indexer K) and Top-K indices to substantially reduce KV cache storage. CSA2 has three statically assigned modes: Full, Reindex, and Reuse. Full Mode generates global KV and performs indexing. Reindex Mode reuses the global KV from a preceding layer, and uses its own indexer Q to rescore the shared indexer K and select fresh Top-K indices. Reuse Mode reuses both global KV and the Top-K indices in a preceding layer, and directly performs sparse attention. In all three modes, each layer retains its own global Q and SWA KV. Sharing global KV and indexer K reduces duplicated cache storage. In addition, different from DeepSeek-V4 that employs the Compressed Sparse Attention (CSA)–Heavily Compressed Attention (HCA) hybrid architecture, DeepSeek-V4.1-Flash uses pure CSA2. At the cache-precision level, we use FP4 global KV caches during training with only marginal performance degradation. Together, CSA2 and FP4 KV

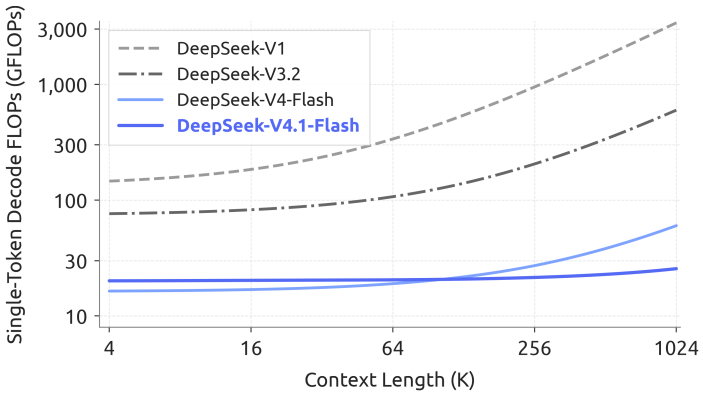

Figure 2 | Single-token Decode FLOPs versus context length across generations of DeepSeek models. We account for compute precision by weighting BF16, FP8, and FP4 operations by 1, 0.5, and 0.25, respectively. DeepSeek-V4.1-Flash maintains nearly constant Decode FLOPs as context length increases, substantially reducing the computational cost of long-context scenarios.

caching reduce global KV cache storage to approximately 1/4 of that of DeepSeek-V4-Flash, as shown in Figure 1(b). At the deployment level, DeepSeek-V4.1-Flash, like DeepSeek-V4, uses Sliding-Window Attention (SWA) in every layer. In DeepSeek-V4, we use a hybrid strategy to balance the storage cost of persisting SWA KV caches against the computation required for exact reconstruction. Exact reconstruction requires replaying the most recent $L \times n_{\text{win}}$ tokens, where L is the number of layers and n_{win} is the SWA window size. In DeepSeek-V4.1-Flash, we introduce SWA Bounded Replay, which approximately reconstructs the required SWA KV states by replaying only the most recent n_{win} tokens. Our experiments show that this incurs only negligible performance degradation. This finding establishes a new storage–computation trade-off, allowing us to avoid persisting SWA KV cache to SSD while incurring a small amount of prefill recomputation. With SWA Bounded Replay, the persistent KV cache footprint is further reduced to approximately 1/8 of that of DeepSeek-V4-Flash. Together, these optimizations greatly ease pressure on HBM and SSD capacity, reduce deployment costs, and pave the way for deployment at a larger scale.

Complementing CED and CSA2, we further streamline the original DeepSeek-V4 architecture. Additionally, we upgrade the original *m*HC (Xie et al., 2026) design to Single-Pass *m*HC, with an accompanying Mega-*m*HC deployment kernel that halves activation memory traffic relative to the original four-kernel implementation. Furthermore, we integrate the Engram (Cheng et al., 2026b) conditional memory module to strengthen model capabilities. We also introduce the DSpark (Cheng et al., 2026a) speculative decoding architecture to improve decoding efficiency through semi-autoregressive draft generation and confidence-scheduled verification. With all the architectural improvements combined, the single-token Decode FLOPs of DeepSeek-V4.1-Flash remain nearly constant across context lengths. Figure 2 shows that extending the context length 256-fold, from 4K to 1M, increases its Decode FLOPs by only 1/4, significantly less than the growth observed for DeepSeek-V4-Flash.

To fully realize the KV cache compression benefits of these architectural designs and further improve training and inference efficiency, we systematically co-optimize the training infrastructure and inference system for DeepSeek-V4.1-Flash, ensuring efficient and scalable large-scale multimodal training and long-context deployment. Training infrastructure supports disaggregated vision-encoder execution, balanced image sharding for long sequences, and cross-stage shared-state management for attention reuse. The inference system implements Encoder and De-

coder SWA Bounded Replay paths. Further optimizations include communication–computation overlap, sharded Engram embedding tables, and inference kernel fusion. In particular, each CSA2 Reuse Mode layer executes with only 15 kernels during prefill and 11 during decode. We also separate long-lived global KV storage from short-lived encoder SWA KV in host memory, using bounded replay to approximately reconstruct missing encoder SWA states.

During pre-training, we train DeepSeek-V4.1-Flash on a large-scale multimodal corpus comprising 45T tokens. Sparse attention is trained from scratch at a sequence length of 64K, without any dense attention warmup stages. After pre-training, the model possesses native multimodal capabilities and supports contexts of up to one million tokens. In our evaluations, DeepSeek-V4.1-Flash-Base achieves world knowledge, reasoning and coding abilities comparable to DeepSeek-V4-Pro-Base, and delivers 5%–10% improvements on held-out evaluations, using only 1/3 total parameters and 1/4 activated parameters. Together, these results highlight its strong parameter efficiency and reflect improvements in training data quality for real-world deployment.

Building on this base model, we conduct post-training to elicit its reasoning and agentic capabilities. In contrast to the architectural innovations described above, our post-training introduces no algorithmic innovation: the recipe follows the standard paradigm of supervised fine-tuning (SFT) followed by reinforcement learning (RL) and on-policy distillation (OPD), without any modification beyond well-established practice used in DeepSeek-V4 development (DeepSeek-AI, 2026b). All substantive changes lie instead in the data pipeline. We develop large-scale automated pipelines for data synthesis and environment construction, and progressively scale the data, tasks, and rollouts employed during RL, thereby extending the model's capabilities across textual, multimodal, and agentic domains. Figure 1(a) summarizes DeepSeek-V4.1-Flash's performance on core agentic benchmarks. Our evaluation shows that, despite its compact size, DeepSeek-V4.1-Flash exhibits a distinctive capability profile:

- **Reasoning.** The model delivers strong reasoning ability, sustaining high accuracy on reasoning-intensive benchmarks such as mathematics and competitive programming, showing comparable performance with top open-source models, such as Kimi-K3(Team et al., 2026a) and DeepSeek-V4-Pro.
- **Agent.** DeepSeek-V4.1-Flash achieves performance on par with closed-source frontier models across standard agentic benchmarks like Terminal-Bench 2.1 (Merrill et al., 2026), DeepSWE v1.1 (DataCurve, 2026), and AutomationBench (Shepard and Salimans, 2026). It has proven fully capable of handling everyday coding tasks and white-collar workflows. However, a gap with giant models remains on science-oriented agentic tasks, such as Terminal-Bench 4.0 (Marten et al., 2026a), that require expert-level domain knowledge.
- **Multimodal.** Within the multimodal domain, the model surpasses top-tier open-source competitors like Kimi-K3 specifically on benchmarks evaluating visual reasoning and the interpretation of professional charts. Beyond formal metrics, it also exhibits practical utility in real-world visual agentic workflows, such as frontend development and office automation, where it can utilize rendered screen captures for visual inspection and self-correction. Nevertheless, we acknowledge that a distinct overall performance gap remains when compared to giant closed-source systems.

These results indicate that DeepSeek-V4.1-Flash can already match closed-source frontier models on the vast majority of benchmarks, and is capable of completing over 95% of real-world tasks. Meanwhile, its small activation footprint yields low inference latency and serving cost. We therefore believe that DeepSeek-V4.1-Flash offers a favorable trade-off between capability and efficiency, and can serve as a fast, affordable assistant supporting the daily work of a

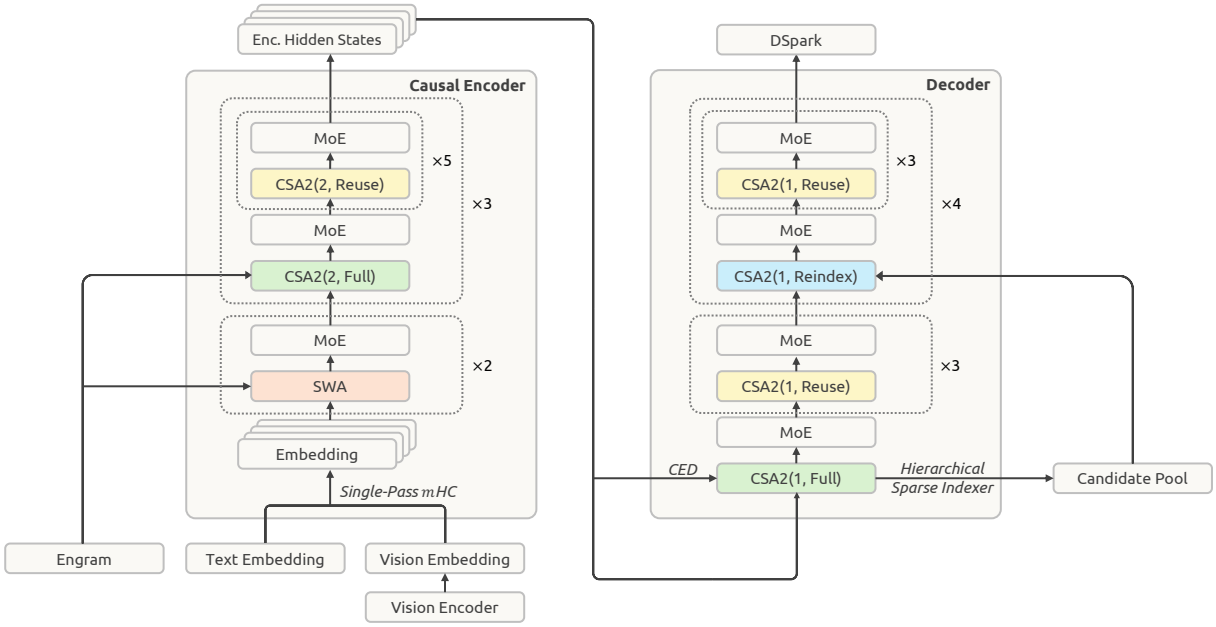

Figure 3 | **Overall architecture of DeepSeek-V4.1-Flash.** The 40-layer network is divided into a causal encoder and a decoder, each with 20 layers. All feed-forward layers use standard DeepSeekMoE. The first two encoder layers use sliding window attention (SWA); the rest use Compressed Sparse Attention 2 (CSA2), with CSA2(ratio, mode) specifying the compression ratio and mode. The model also uses Single-Pass *m*HC, Engram, DSpark, and a Hierarchical Sparse Indexer.

broad population of users. In summary, DeepSeek-V4.1-Flash simultaneously improves model intelligence and inference efficiency while reducing deployment costs. It substantially lowers the cost barrier to deploying long-horizon agents at scale and creates new opportunities for their adoption across a broader range of scenarios. DeepSeek-V4.1-Flash also serves as a new starting point for our continued scaling efforts. Building on this foundation, we will pursue the joint scaling of model architecture, pre-training, and post-training to further explore the frontier of model intelligence.

2. Architecture

2.1. Overview

DeepSeek-V4.1-Flash is a multimodal mixture-of-experts (MoE) Transformer that takes images and text as input and generates text autoregressively. Its language backbone comprises 40 causal Transformer layers, organized into a 20-layer causal encoder followed by a 20-layer decoder. Each layer incorporates both global attention and sliding window attention (SWA), except for the first two layers, which use SWA only. A vision encoder and an MLP projector convert images into visual embeddings that are processed jointly with text embeddings, with multimodal data incorporated from the start of language-model pre-training. Overall, DeepSeek-V4.1-Flash has 552B backbone parameters and 196B Engram parameters, activating 8B parameters per token during prefill and 16B during decode. Figure 3 illustrates the overall architecture of DeepSeek-V4.1-Flash.

The Causal Encoder–Decoder (CED) architecture and Compressed Sparse Attention 2 (CSA2)

address complementary costs of long-context inference. CED constructs the decoder's global key-value (KV) cache from encoder outputs, allowing most prompt tokens to bypass full decoder computation while retaining layer-local sliding-window attention. This nearly halves prefill computation, lowering the cost of processing new or uncached inputs in agentic workloads with growing contexts. CSA2 shares global KV across layers to reduce cache storage and reuses sparse selections to reduce indexing work. In the decoder, a Hierarchical Sparse Indexer restricts later indexers to a candidate pool selected by an earlier indexer, further reducing the number of entries scored per query.

We retain the shared and fine-grained routed experts of DeepSeekMoE (Dai et al., 2024), and introduce modality-specific load balancing (Wang et al., 2024a) for image and text tokens. Single-Pass *m*HC (Xie et al., 2026) revises residual-stream mixing to enable more efficient kernel fusion, and Engram (Cheng et al., 2026b) adds sparsely accessed conditional memory. We omit the MTP module during backbone pre-training and use DSpark (Cheng et al., 2026a) for speculative decoding. We train DSpark separately after the backbone pre-training stage. Additionally, we compress the main KV cache to FP4 to further reduce storage overhead. The following sections describe these components and the corresponding optimization changes.

2.1.1. Multimodal Architecture

The multimodal input pathway comprises a vision encoder and an MLP projector. For each input image, the vision encoder produces a spatial grid of visual features. A 3×3 pixel-unshuffle operation then rearranges each local neighborhood along the channel dimension, reducing the spatial resolution before the MLP projector maps the features to the hidden dimension of the language backbone. Finally, the resulting visual embeddings are inserted at the corresponding image-token positions in the input embedding sequence and processed jointly with text embeddings by the language backbone.

DeepSeek-ViT We train a vision encoder named DeepSeek-ViT from scratch to natively process images at varying resolutions. We build DeepSeek-ViT on the Vision Transformer (Dosovitskiy et al., 2021) architecture with several modifications. To accommodate inputs of arbitrary resolutions, we replace standard absolute positional embeddings with 2D-RoPE. To align the ViT more closely with LLM design principles, we replace the patch embedding layer's convolution with a linear projection to ensure compatibility with the Muon optimizer. We also adopt RMSNorm (Zhang and Sennrich, 2019) for normalization and SwiGLU (Shazeer, 2020) as the activation function. Before feeding visual features into the LLM, we apply a pixel-unshuffle operation with 3×3 downsampling to reduce the visual token count by a factor of nine, effectively supporting input resolutions up to approximately 1344×1344 pixels.

Multimodal Auxiliary-loss-free Load Balancing for MoEs Image and text tokens exhibit distinct representation distributions and may induce different expert-routing preferences in MoEs. Balancing their aggregate load may therefore obscure modality-specific imbalance. To address this issue, we extend auxiliary-loss-free load balancing (Wang et al., 2024a) by maintaining separate expert-wise correction biases for text and image tokens. During routing, each token uses the correction biases associated with its modality for expert selection, while the original routing scores are retained for weighting the selected expert outputs. After each training step, the two sets of biases are updated independently according to their respective expert loads. This design balances expert utilization within each modality and contributes to stable and efficient multimodal training.

2.2. Causal Encoder-Decoder (CED)

In agentic workflows, frequent tool calls generate extensive prefill requests, imposing severe computational overhead when KV caches miss. To alleviate this prefill bottleneck, we propose the Causal Encoder-Decoder (CED) architecture, inspired by YoCo (Sun et al., 2024). YoCo reduces prefill computation by allowing the upper half of the layers to directly share the KV cache generated by the lower half. Building upon this concept, CED introduces a series of structural improvements to enhance both the overall KV cache capacity and the computational depth of KV generation. Consequently, CED successfully reduces nearly half of the prefill computation while maintaining performance comparable to the baseline.

For global attention, CED treats the bottom $L/2$ layers of the Transformer as the causal encoder. For the upper half layers (i.e., the decoder, $l > L/2$), the KV entries are not derived from their respective hidden states H_l. Instead, they are projected directly from the hidden state of the $(L/2)$-th layer, $H_{L/2}$, using layer-dependent projection weights (W_l^{KV} and W_l^Z):

$$C_l = H_{L/2} W_l^{KV}, \quad Z_l = H_{L/2} W_l^Z, \quad l > \frac{L}{2}, \tag{1}$$

where C and Z represent the KV entries and their corresponding compression weights, respectively. This design allows CED to compute only the first half of the layers during the prefill phase, acquiring the upper-layer global KV cache with minimal computational cost.

For sliding window attention (SWA), CED maintains the conventional layer-wise computation across all layers. Specifically, for any layer l, the local keys and values are derived directly from the current layer's hidden state H_l. This design effectively increases the computational depth of local KV generation. However, maintaining this layer-wise computation necessitates an SWA replay process. During the prefill phase, computing the SWA KV cache for the decoder requires processing an additional $n_{\text{win}} \times L/2$ tokens (where n_{win} denotes the window size). For multi-turn interactions with short prompts per turn, this computational overhead in the decoder becomes non-negligible. Fortunately, prior work (Chen et al., 2025) has shown that the actual effective receptive field of SWA is much smaller than the theoretical $n_{\text{win}} \times L/2$. Motivated by this observation, we introduce Decoder SWA Bounded Replay, which only prefills the last n_{win} tokens of the prompt for the SWA computation, thereby significantly reducing the computational cost. Further details are provided in Section 3.2.2.

Overall, for a sequence length $N \gg n_{\text{win}}$, CED reduces the prefill complexity from $O(NL)$ to $O(NL/2 + n_{\text{win}} \times L/2) \approx O(NL/2)$, effectively halving the overall computation.

2.3. Compressed Sparse Attention 2 (CSA2)

Serving long contexts requires controlling both KV cache storage and attention computation. These costs can be reduced along three multiplicative dimensions: the entry size, where GQA (Ainslie et al., 2023) reduces the number of KV heads and MLA (DeepSeek-AI, 2024) shares a small latent across heads; the sequence dimension, where every m tokens are compressed into one entry, like CSA and HCA in DeepSeek-V4 (DeepSeek-AI, 2026b); and the layer dimension, where some layers reuse the caches (Brandon et al., 2024) and selections of other layers instead of keeping their own, or are replaced altogether by more efficient layers. Prior work has shown that compression along the layer dimension is effective: IndexCache (Bai et al., 2026) reuses Top-K indices across layers to cut indexer computation; YOIO (Sun et al., 2026b) computes the sparse routing once and shares it across all layers; and HySparse (Gao et al., 2026) lets sparse layers reuse the KV cache of dense layers. However, index reuse alone saves no main

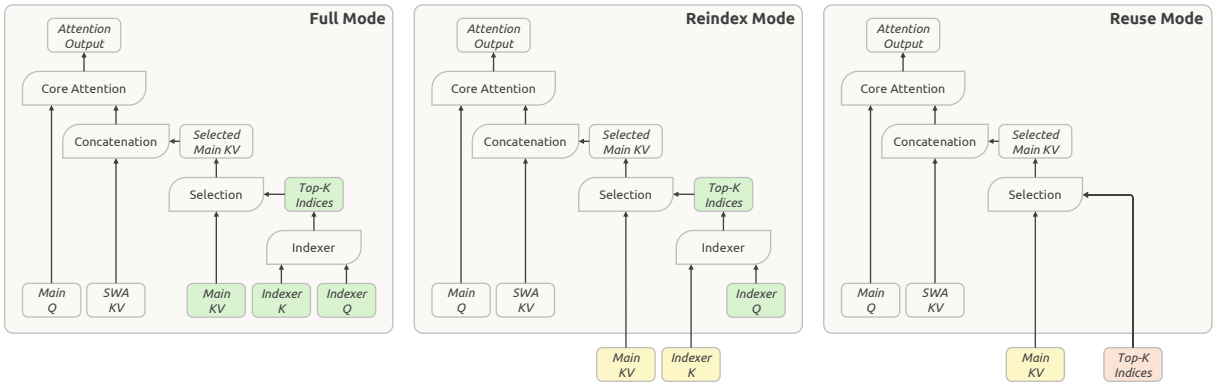

Figure 4 | **Three operating modes of CSA2.** The modes differ in how they obtain main KV, indexer K, and Top-K indices. Green blocks indicate quantities computed in the current layer; yellow blocks indicate main KV and indexer K reused from the most recent Full Mode layer; while red blocks indicate Top-K indices reused from the most recent index-producing (Full or Reindex Mode) layer. All three modes compute main Q and SWA KV in the current layer.

KV storage, network-wide routing sharing limits performance, and hybrid designs still retain full attention layers; more importantly, none of these methods covers all three multiplicative dimensions.

CSA2 exploits the three dimensions jointly: it shares main KV and indexer K across layers and allows layers to reuse Top-K indices, with cache sharing and index reuse decoupled. It combines these reuse strategies with a simplified compressor and a Hierarchical Sparse Indexer that narrows the search domain of subsequent indexing layers in the Decoder.

Similar to CSA, CSA2 includes a lightweight indexer that scores the main KV entries using indexer Q and indexer K and selects the Top-K entries for each query. Each Q attends to the selected entries together with the layer-local sliding-window KV (SWA KV). CSA2 also includes the uncompressed main KV setting as a special case with a compression ratio of 1. Meanwhile, CSA2 simplifies both the compressor and the indexer. In CSA, a compression ratio of m produces each main KV entry from $2m$ original KV cache entries, with overlapping source entries for adjacent compressed entries. It also includes absolute positional embedding to encode the positions of these $2m$ entries during compression. CSA2 removes this overlap and absolute positional embedding. In addition, CSA2 obtains indexer K by projecting main KV entries, replacing CSA's separate compression path from hidden states. Both designs simplify the implementation and increase the training efficiency.

Sections 2.3.1 and 2.3.2 describe the cross-layer reuse strategies and the Hierarchical Sparse Indexer, respectively.

2.3.1. Cross-Layer KV and Index Reuse

Each CSA2 layer is statically assigned one of three modes: Full, Reindex, or Reuse. In all three modes, the layer computes its own query and SWA KV and uses them together with the selected main KV entries to produce a new attention output. The modes differ in how they obtain main KV, indexer K, and Top-K indices. Figure 4 illustrates the three modes.

Full Mode. The layer computes its own main KV and indexer Q, projects indexer K from that main KV, and runs the indexer to produce fresh Top-K indices. It therefore executes the complete CSA2 computation path and has the same component responsibilities as a complete

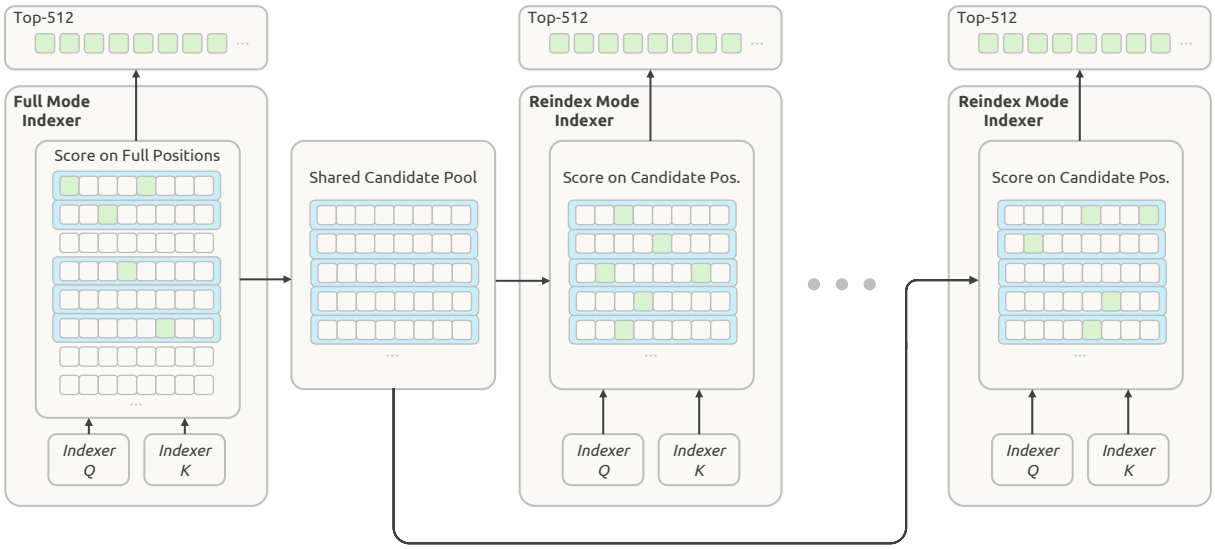

Figure 5 | **Hierarchical Sparse Indexer.** Each square represents a position; green squares mark selected indices, and blue rectangles mark blocks selected based on their maximum indexer scores. The decoder's first CSA2 layer in Full mode selects its own Top-512 indices and builds a shared candidate pool from the selected blocks for subsequent layers. CSA2 layers in Reindex mode then select their Top-512 indices from this pool.

CSA layer in DeepSeek-V4.

Reindex Mode. The layer reuses the most recent available main KV from a preceding layer together with its corresponding indexer K. The indexer computes its own query, rescores the reused keys, and produces fresh Top-K indices. This allows the sparse selection to change across layers while main KV and indexer K remain shared.

Reuse Mode. The layer reuses the most recent available main KV and the latest Top-K indices computed against that main KV by a preceding layer in Full or Reindex Mode. It performs attention using this selection without computing indexer Q or evaluating index scores.

Sharing main KV and indexer K reduces cache storage, while reusing Top-K indices avoids additional indexer computation. Reindex Mode preserves cache sharing while allowing the selected entries to change across layers. When CSA2 is combined with CED, the decoder layer assigned to Full Mode computes its own global KV from the hidden state of the $(L/2)$-th layer, i.e. the last layer of the causal encoder. The Reindex and Reuse Modes are unchanged.

2.3.2. *Hierarchical Sparse Indexer*

Cross-layer index reuse reduces the number of indexer evaluations, but the remaining indexers still score the full causally visible context. For extremely long contexts, this cost remains a major computational bottleneck. Prior work introduced indexer sparsity by scoring and pruning pooled block representations before token-level indexing (Xu et al., 2026b). We find that in the decoder, information from shallower indexers can naturally be used to restrict the candidates considered by deeper indexers without adding any extra state. We therefore introduce the Hierarchical Sparse Indexer, which is used only in the decoder of CED to reduce this repeated scoring during decode. For each query, the first layer assigned to Full Mode constructs a candidate pool that later re-indexing layers use as their search domain. For a fixed candidate-pool size, this changes the per-query cost of deeper indexers from linear in context length to

constant. The mechanism is training-aware and introduced in post-training: the candidate restriction is applied identically during training and inference, so deeper indexers are optimized under the same search domain they use at inference. Figure 5 illustrates this process.

This first Full Mode layer scores all causally visible main KV positions and produces the Top-K indices for its own attention. It also performs blockwise candidate selection: each block is assigned the maximum index score among its positions, and the blocks with the highest scores are selected. It then collects the positions covered by the selected blocks into a candidate pool larger than the final Top-K set. For example, selecting 2,048 blocks with 8 positions each yields 16,384 candidate positions. This pool defines where later indexers search; the final Top-K selection determines which main KV entries each layer reads.

Subsequent layers in Reindex Mode score only the candidate positions for the corresponding query and select their own Top-K entries within that pool. Layers in Reuse Mode perform no new indexing and use the latest Top-K indices computed against the main KV they reuse. Thus, the candidate pool is shared across indexing layers, while their final selections can differ.

For a fixed candidate-pool size, the number of positions scored per query by each subsequent indexer is bounded independently of context length. The first Full Mode layer still scans the entire causally visible range. Hierarchical indexing therefore reduces the cost of later indexer evaluations while retaining the initial full-range pass.

2.4. Efficient Architectural Extensions

2.4.1. *Single-Pass mHC*

In DeepSeek-V4, we introduced mHC (Xie et al., 2026), which maintains n residual streams between adjacent Transformer blocks. For each token, we denote these streams by $X_l \in \mathbb{R}^{n\times d}$, where l is the block index and d is the hidden dimension. The streams are updated as follows:

$$X_{l+1} = B_l X_l + C_l\, \mathcal{F}_l(A_l X_l), \qquad (A_l, B_l, C_l) = \mathcal{H}(X_l), \tag{2}$$

where $A_l \in \mathbb{R}^{1\times n}$, $C_l \in \mathbb{R}^{n\times 1}$ and $B_l \in \mathbb{R}^{n\times n}$ are token-wise coefficients predicted from X_l. The coefficient predictor $\mathcal{H}$ includes normalization and projection.

Ideally, the residual transformation between two blocks is a single map from (X_{l-1}, Y_{l-1}) to $(X_l, \hat{X}_l)$, where $\hat{X}_l = A_l X_l$ is the current block input and $Y_{l-1} = \mathcal{F}_{l-1}(\hat{X}_{l-1})$ is the previous block output. Such a map requires $(n+1)d$ reads and $(n+1)d$ writes, giving a lower bound of $(2n+2)d$ on activation memory traffic. In practice, DeepSeek-V4 uses a multi-pass implementation of (2), with three kernels that execute sequentially due to data dependencies:

$$X_l = B_{l-1} X_{l-1} + C_{l-1} Y_{l-1} \qquad \text{Residual update, contraction over } n \tag{3}$$

$$(A_l, B_l, C_l) = \mathcal{H}(X_l) \qquad \text{Coefficients, contraction over } nd \tag{4}$$

$$\hat{X}_l = A_l X_l \qquad \text{Input mixing, contraction over } n \tag{5}$$

The three kernels read $(n+1)d$, nd and nd values respectively and write $(n+1)d$ in total. Including the pre-norm in $\mathcal{F}_l$, the total activation memory traffic is $(4n+4)d$, twice the lower bound.

In $\mathcal{H}$, the normalization weights are folded into the projection weights offline, and the RMS division is applied after the projection. Two of the three stages can therefore share one traversal of the residual: the residual update does not require a reduction across the hidden dimension, so each tile of X_l can be computed and immediately used to accumulate the projection outputs

and the sum of squares needed to compute the RMS. Input mixing cannot be fused into this pass because A_l is not available until the reduction over all hidden tiles is complete. It therefore requires a second read of X_l. This second pass can also incorporate input pre-norm. This two-pass implementation would require $(3n+2)d$ activation reads and writes in total, one additional read of X_l compared with the lower bound.

We therefore introduce **Single-Pass *m*HC**, which shifts the input-mixing coefficients by one block, i.e. every block consumes the mixing coefficients produced by the previous one, so that the above dependency disappears:

$$X_{l+1} = B_l X_l + C_l \mathcal{F}_l(A_{l-1} X_l), \qquad (A_l, B_l, C_l) = \mathcal{H}(X_l). \tag{6}$$

Input mixing now uses A_{l-1} instead of A_l, so it no longer depends on the coefficients computed from X_l. Each tile of X_l can therefore be used immediately for both input mixing and coefficient prediction, without waiting for the full reduction. Empirically, this shift incurs negligible performance degradation.

For pre-training, we keep the existing multi-kernel implementation, since the shift only changes which mixing coefficients each block applies. For deployment, we fuse residual update, input mixing, and coefficient prediction into a single kernel, **Mega-*m*HC**. The kernel implements *m*HC with $(3n+2)d$ activation reads and writes and Single-Pass *m*HC with $(2n+2)d$ activation reads and writes. Mega-*m*HC processes X_l in tiles along the hidden dimension. Each tile is used to compute the mixed input and to accumulate the quantities needed to predict (A_l, B_l, C_l) for the next block. The kernel also incorporates input pre-norm and FP8 conversion. The residual is thereby read once and written once, attaining the $(n+1)d$ reads and $(n+1)d$ writes of the ideal map and halving the activation memory traffic of our original implementation.

2.4.2. Engram

We augment DeepSeek-V4.1-Flash with Engram (Cheng et al., 2026c), the conditional memory module introduced in our previous work to decouple memorization from computation. We follow the original Engram design—tokenizer compression, multi-head hashing, context-aware gating, and multi-branch integration—with two modifications. First, we omit the short causal convolution because its performance gains do not justify the added complexity in our inference stack. Second, we optimize the Engram embedding with momentum-based update followed by Sinkhorn balancing, as detailed in Section 2.5.

We allocate 196B Engram parameters evenly across two modules. Each module uses N-gram orders $\{2, 3, 4\}$, with 8 hash heads and a total embedding dimension of 2048 per order. Each head indexes a table of approximately 16M entries, with table sizes chosen to be distinct primes. Both the embedding tables and the key/value projections use FP8 precision. The modules are placed at layers 1 and 14 (zero-indexed) to balance memory usage across training pipeline stages. During inference, deterministic addressing enables embeddings to be prefetched from host memory via background RDMA transfers, with prefetching for the first module overlapping computation in the first Transformer block. Further implementation details for Engram training and inference are discussed in Section 3.1.3.

2.4.3. DSpark

We equip DeepSeek-V4.1-Flash with DSpark (Cheng et al., 2026a), a speculative decoding module that combines semi-autoregressive drafting with confidence-scheduled verification.

The drafter comprises three Transformer blocks with a sliding attention window of 128 tokens. A single forward pass through these blocks computes base logits for five draft positions in parallel, while a lightweight Markov head models dependencies among the draft tokens. A confidence head predicts per-position conditional acceptance probabilities, which are used to estimate prefix survival probabilities. The scheduler combines these estimates with profiled engine throughput curves to dynamically select the verification length for each request, aiming to maximize expected system-wide token throughput under the current system load.

Unlike the MTP module in DeepSeek-V3 (DeepSeek-AI, 2024), which is trained jointly with the backbone throughout pre-training, DSpark is introduced in a dedicated stage after pre-training. In this stage, we train only DSpark while keeping the backbone frozen. During post-training, we continue to train DSpark alongside the backbone, without propagating gradients from the DSpark objective into the backbone. This keeps DSpark aligned with the evolving policy, enabling it to accelerate both online serving and rollout generation for RL and OPD.

2.4.4. FP4 Main KV Cache

Long-context agent workloads require large per-request KV caches, increasing serving costs. DeepSeek-V4 already uses quantization-aware training (QAT) (Jacob et al., 2018) for FP4 indexer queries and keys, accelerating index computation and reducing the indexer cache size. We adopt the OCP-standard MXFP4 format (Rouhani et al., 2023) to support as many hardware platforms as possible, despite the higher accuracy of alternative formats in our experiments. We now extend QAT to the main KV cache, where FP4 reduces storage rather than accelerates matrix multiplication. Dequantizing cached values before attention allows us to use a more accurate format without requiring native matrix-multiplication support for that format, preserving compatibility across hardware platforms.

Among the approximately four-bit formats evaluated, we select E2M1 with one E4M3 scale per 16 channels, following NVFP4 (Alvarez et al., 2025) but omitting its second-level global scale to balance accuracy and simplicity. Omitting this scale leaves ample dynamic range for the main KV cache: the format supports magnitudes up to $448 \times 6 = 2688$, far above the cache's magnitude bound. In DeepSeek-V4.1-Flash, the largest trained RMSNorm weight magnitude is approximately 1. After RMS normalization, the L2 norm of the 512-channel KV latent is at most approximately $\sqrt{512}$. RoPE preserves this norm, so the maximum absolute value across channels after rotation is also bounded by approximately $\sqrt{512} \approx 22.6$. Besides, the maximum magnitude observed during training is around 10. Therefore, omitting the global scale causes no measurable decrease in accuracy and simplifies the cache layout.

To enable FP4 main KV cache storage in DeepSeek-V4.1-Flash, we introduce QAT during post-training. The non-RoPE and RoPE components use the same quantization format. We quantize the cache after RoPE: quantizing before RoPE yields only a marginal accuracy improvement in our experiments and would introduce additional overhead during decoding. We retain FP8 for the SWA KV cache due to its sensitivity to quantization. Compared with the FP8 main KV cache in DeepSeek-V4, this format nearly halves the storage footprint, both in HBM and when offloaded to SSD.

2.5. Optimization

Building upon the optimization configuration used in DeepSeek-V4, we make some new modifications to better align with the architectural design.

First, we use head-wise Muon, where Query weights are split by head before applying the

Muon update. Here, we briefly discuss the motivation for such a design. By viewing Muon as a preconditioned gradient descent, vanilla Muon uses one preconditioner for all heads, whereas head-wise Muon provides different preconditioners for different heads. This design can better handle the heterogeneity across attention heads (Zhang et al., 2024; Zhang, 2026, Section 3). As a result, we observe that head-wise Muon outperforms vanilla Muon. The empirical advantage of head-wise Muon is also validated in GLM 5 (Zeng et al., 2026) and Kimi-K3 (Team et al., 2026a).

Second, applying Adam to the newly introduced Engram parameters substantially increases the optimizer-state memory footprint. To reduce memory usage during training, we instead optimize the Engram embedding tables, token embedding, and prediction head using a momentum-based update followed by Sinkhorn balancing. Sinkhorn balancing has previously been applied to linear-layer weight matrices in SinkGD (Scetbon et al., 2025); here, we extend it to these large parameter matrices. Like Muon, this approach requires only a momentum buffer while empirically outperforming Adam.

Basic Configurations. We retain AdamW (Loshchilov and Hutter, 2019) for normalization-layer weights and other non-matrix parameters, including biases and scaling factors. We use Muon (Jordan et al., 2024) for the weight matrices of linear transformations in the language-model backbone, the Engram projection layers, and the vision-language projector. We use head-wise Muon for Query and Key weights. We apply decoupled weight decay and Nesterov momentum to Muon (Nesterov, 1983; Liu et al., 2025); normalization-layer weights are also subject to weight decay, whereas biases and scaling factors are not. The Sinkhorn-balanced update also uses Nesterov momentum but does not apply weight decay. During pre-training, we keep the vision encoder frozen until the learning-rate decay stage, while its final normalization layer and the vision–language projector remain trainable. At the onset of learning-rate decay, we unfreeze the vision encoder and optimize it jointly with the LLM with a smaller learning rate.

Sinkhorn-Balanced Updates for Engram / Embedding / Prediction Head. The complete procedure is summarized in Algorithm 1. At a high level, it follows the same workflow as Muon, with Sinkhorn balancing taking the place of Newton–Schulz orthogonalization. We denote the larger matrix dimension by m, which corresponds to the vocabulary size for embedding tables and prediction heads, and denote the hidden dimension by n.

Given the Nesterov momentum update $\widehat{G}_t$, Sinkhorn balancing finds diagonal scaling matrices D_r and D_c such that

$$\Delta_t = \sqrt{n}\, U^{(K)} = \sqrt{n}\, D_r \widehat{G}_t D_c, \qquad \frac{1}{n}\sum_{j=1}^{n} (\Delta_t)_{ij}^2 \approx 1, \qquad \frac{1}{m}\sum_{i=1}^{m} (\Delta_t)_{ij}^2 \approx 1, \tag{7}$$

Thus, the procedure approximately equalizes the row-wise and column-wise RMS of the update matrix. Here, one row corresponds to one token index or n-gram identity; and one column encodes one hidden feature. Sinkhorn balancing exploits this token–feature structure by normalizing along both rows and columns. For numerical stability, rows satisfying $\rho_i \leqslant \tau\bar{\rho}$ are masked. The factor $\sqrt{n}$ converts unit row ℓ_2 norm into unit row-wise RMS. Separately, we adjust the effective learning rate as $\widetilde{\eta}_t = \gamma\eta_t$ to match the update magnitude of Adam. We set $\gamma = 0.18$, which is close to the factor 0.2 used in Moonlight (Liu et al., 2025).

More broadly, Sinkhorn balancing is closely related to optimizers that exploit matrix or tensor axis structure (Shazeer and Stern, 2018; Zhang et al., 2025a; Wen et al., 2025; Glentis et al., 2025; Deng et al., 2026; Yuan et al., 2026; Xu et al., 2026a). For example, Adafactor (Shazeer and

Algorithm 1 Momentum update with Sinkhorn balancing

Require: Weight $W_t \in \mathbb{R}^{m\times n}$, gradient G_t, momentum M_{t-1}, momentum coefficient β, base learning rate η_t, learning-rate correction γ, numerical constants ε and τ, and an odd number of normalization steps K

$M_t \leftarrow \beta M_{t-1} + (1-\beta)G_t$
$\widehat{G}_t \leftarrow \beta M_t + (1-\beta)G_t$ ▷ Nesterov momentum
$\rho_i \leftarrow \|\widehat{G}_{t,i,:}\|_2$ and $\bar{\rho} \leftarrow \frac{1}{m}\sum_{i=1}^{m}\rho_i$
$U^{(0)} \leftarrow \widehat{G}_t$
Set $U^{(0)}_{i,:} \leftarrow 0$ if $\rho_i \leqslant \tau\bar{\rho}$ ▷ Mask near-zero rows
for $k = 1, \ldots, K$ **do**
  if k is odd **then**
    for all $i = 1, \ldots, m$ **do**
      $U^{(k)}_{i,:} \leftarrow U^{(k-1)}_{i,:} / (\|U^{(k-1)}_{i,:}\|_2 + \varepsilon)$
    end for
  else
    for all $j = 1, \ldots, n$ **do**
      $U^{(k)}_{:,j} \leftarrow U^{(k-1)}_{:,j} / (\|U^{(k-1)}_{:,j}\|_2 + \varepsilon)$
    end for
  end if
end for
$\Delta_t \leftarrow \sqrt{n}\, U^{(K)}$ ▷ Convert unit row ℓ_2 norm to unit row RMS
$\widetilde{\eta}_t \leftarrow \gamma\eta_t$ ▷ Match the update magnitude of Adam
$W_{t+1} \leftarrow W_t - \widetilde{\eta}_t\Delta_t$

Stern, 2018) conducts row- and column-wise normalization in a different manner, and Adam-mini (Zhang et al., 2025a) uses an alternative row-wise normalization for embedding tables and prediction head. These normalization strategies may differ in optimization performance and communication overhead. We leave more detailed investigation as a future direction.

3. General Infrastructures

3.1. Training Infrastructure

3.1.1. Multimodal Training Infrastructure

Communication-Computation Overlap in Contrastive Learning. The vision encoder is first optimized with a contrastive objective before being fine-tuned with a generative next-token prediction loss. In the contrastive phase, the loss is computed over a full batch of text and vision pairs, so the features of both modalities must be all-gathered across data-parallel ranks, incurring substantial communication. Because the gradient of the text features depends only on the gathered visual features—and, symmetrically, the gradient of the visual features depends only on the gathered text features—each all-gather can be overlapped with the forward or backward pass instead of stalling the pipeline:

$$\begin{aligned}\text{Forward}(V) \;\rightarrow\; \big(\text{Forward}(T) \parallel \text{AllGather}(V)\big) \;\rightarrow\; \nabla_{\text{Text}} \\ \rightarrow\; \big(\text{Backward}(T) \parallel \text{AllGather}(T)\big) \;\rightarrow\; \nabla_{\text{Vision}} \;\rightarrow\; \text{Backward}(V),\end{aligned}$$

where V and T denote the visual and text features, $(A \parallel C)$ denotes the overlap of computation A with communication C, and ∇ denotes the gradient computation. In this schedule, the visual

features are gathered during the text forward pass and the text features during the text backward pass, so that both all-gathers are hidden entirely behind useful computation.

End-to-End Parallelism. To handle the model and data heterogeneity between the vision encoder and the LLM (Zhang et al., 2025b), we adopt the disaggregated encoder design used in recent training systems (Team et al., 2025, 2026b). The vision encoder is replicated outside the LLM parameter tree, and each training step is divided into three phases: vision encoder forward, LLM forward/backward, and vision encoder backward. This separation prevents interference between vision encoder and LLM computation. Load-balanced vision processing is confined to the first and last phases, while the LLM phase remains free of vision computation and preserves the parallel strategy of text-only training.

Long-Sequence Multimodal Training Optimization. DeepSeek-V4.1-Flash is trained on sequences of up to one million tokens, with a substantial share of training occurring at ultra-long sequence lengths in both the pre-training and post-training stages. At these sequence lengths, multimodal samples create heavy I/O, CPU, and memory bottlenecks.

- **Balanced image sharding.** During pre-training, a single ultra-long, image-dense sequence can exhaust one host's I/O, CPU, and memory during loading, so the images of each sequence are sharded across the CP ranks with load balancing, and each image is loaded exactly once. With images read once, loading stays hidden behind compute whenever

$$\frac{N \times \rho}{B_{\text{IO}}} < \frac{N \times C}{B_{\text{GPU}}} \quad \Longleftrightarrow \quad \rho < \frac{B_{\text{IO}}}{B_{\text{GPU}}} C,$$

  where N is the token count, ρ the raw bytes per token, C the per-token compute, and B_{IO}, B_{GPU} the file-system and GPU bandwidths. Since N cancels, the criterion involves only per-token quantities (ρ and C), independent of sequence length and cluster size; ρ is set by the vision-module configuration (e.g., the resolution cap or the spatial downsample). Storage throughput therefore becomes a bottleneck only for small models with low per-token compute, as in ablations, while production-scale models remain compute-bound.
- **Incremental image transfer.** Besides the balanced sharding above, the reinforcement-learning rollout transfers images to the inference engine only incrementally, and caches the engine's CPU-side decoding and preprocessing outputs on a distributed file system for reuse across rollouts and subsequent training.

3.1.2. Attention Sharing Training for CSA2

In Section 2.3, we introduce CSA2, an attention sharing method with three modes, some of which involve sharing one or more of the main KV, the indexer K, and the Top-K indices across multiple layers. Supporting CSA2 in large-scale distributed training requires additional coordination beyond the attention computation itself. In particular, layers that share attention components may be placed on different pipeline stages, making direct module reuse incompatible with conventional stage-local execution. Therefore, we adopt several designs to support CSA2 training.

Shadow indexers address this issue by placing a lightweight executable replica on each participating stage while retaining a single logical owner for the shared parameters. The owner remains responsible for optimization and checkpointing, whereas parameter synchronization

and gradient aggregation keep the shadow replicas consistent throughout training. This design preserves the original model semantics without requiring the pipeline scheduler to treat shared layers as a special execution unit.

Pipeline payload extensions provide the intermediate representations and sparse routing information required by downstream consumers when the source and consumer layers cross a pipeline boundary. These states are incorporated into the existing point-to-point communication path and partitioned consistently with context parallelism, avoiding unnecessary replication while maintaining the corresponding gradient flow.

Micro-batch-level shared-state management tracks the states associated with concurrently active pipeline micro-batches and coordinates their lifetimes across forward execution, activation recomputation, and backward propagation. Shared states are retained until their final consumer has completed and are then released promptly to limit additional memory overhead. The same runtime abstraction also handles stage placement and source–consumer relationships, allowing the attention implementation to access shared states without depending on the physical pipeline layout.

Together with lightweight adaptations to the optimizer, checkpointing, warm-up, and computation-graph tracing workflows, these mechanisms enable CSA2 to operate transparently under the existing distributed training interface and pipeline schedules.

3.1.3. Engram

Engram embedding tables are partitioned by row across dedicated process groups of engram parallel size. The group size controls the trade-off between per-device memory usage and the communication scope of embedding lookups. Optimizer states are further sharded across replicas of each table partition. Engram lookup indices depend solely on the input token sequence. Embedding prefetch is therefore initiated for the entire local batch before each pipeline stage begins processing microbatches for the current training step, minimizing interference with pipeline execution. Embedding gradients are buffered during backward and returned to their owning ranks after the backbone backward pass. For efficient integration with multimodal training, embedding prefetch and gradient transfers are scheduled to overlap with the vision encoder's forward and backward computation. Embeddings are stored and fetched in FP8, with the retrieved values and scaling factors passed directly to the following GEMM. For Engram table updates, Sinkhorn normalization maintains row and column scaling vectors across iterations to avoid repeated writes of the full normalized matrix. Row normalization and the accumulation of partial column statistics are fused into a single kernel to further reduce memory traffic. During RL rollouts, Engram embedding tables remain resident in GPU memory. This placement reduces host memory pressure and helps avoid out-of-memory failures caused by host memory fragmentation.

3.2. Inference System

DeepSeek-V4.1-Flash is designed with inference efficiency as a first-class concern. Although its architecture is conceptually complex, the resulting inference kernel flow is remarkably concise. Through reasonable kernel fusion, we encapsulate the intricate operations and keep hardware resources fully pipelined inside a small number of fused kernels—including the fused-RoPE-attention-RoPE-cast kernel in FlashMLA (Li and Liu, 2025), the Mega-Gate, Mega-*m*HC, and Mega-MoE kernels in DeepGEMM (Zhao et al., 2025), the kernels in TileKernels (Wang et al., 2026a), and the TopK kernel in DeepSelect (Qian et al., 2026). As a result, the vast majority of

Transformer layers—those whose CSA2 operates in Reuse Mode—execute with only 15 kernels during prefill and 11 during decode, thereby achieving both high-throughput and low-latency inference.

At the deployment level, we adopt Encoder–Prefill–Decode (EPD) disaggregation, enabling vision encoding, prefill, and decoding to scale independently and overlap in execution.

3.2.1. Persistent KV Cache Management

Under identical workloads, V4.1 reduces the persistent KV cache footprint to 1/8 of that of V4. Two multiplicative factors account for this reduction: the persistent KV cache no longer stores SWA KV, which almost halves its size, and the global KV retained in it is further compressed to 1/4 of V4's footprint through architectural and precision optimizations.

In the V4 deployment, SWA KV accounts for nearly half of the persistent KV cache capacity. Within this cache, global KV and SWA KV are managed independently, governed by an LRU eviction policy. Global KV is stored in its entirety, and upon a hit, the complete prefix is reused. In contrast, SWA KV is cached at two specific points—the end of the prompt and the end of the output—to facilitate regeneration and multi-turn sessions; a hit allows computation to resume from that cached position. To maintain a high hit rate, we configured a sufficiently large persistent KV cache on SSD such that, under typical workloads, both types of KV remain resident for over 72 hours. Despite retaining only n_{win} KV entries at designated positions, the uncompressed SWA KV cache still incurs a substantial storage overhead, especially in multi-turn conversations with short turns.

Persistently storing SWA KV is both costly and ineffective, because its access pattern does not match the persistent KV cache's long retention policy. Unlike global KV, which exhibits long-tail reuse, SWA KV is reused only within a narrow, minute-scale window inside an active session and becomes dead once the session ends or the next turn begins. The V4 technical report proposed *Zero SWA Caching*, which avoids the storage overhead by recomputing missing SWA KV. Exact recovery, however, requires a full forward pass over $L \times n_{\text{win}}$ tokens, whose cost proved prohibitive in production deployments.

V4.1 therefore revises persistent KV cache management as follows:

1. SWA KV is no longer cached in the persistent KV cache and instead stored in a distributed memory pool provisioned from 10% of the host DRAM on each machine. Although this pool is far smaller in aggregate capacity, its short TTL (only minutes) allows expired entries to be recycled immediately for new sessions; under real-world workloads, this high turnover suffices to serve the vast majority of concurrent active sessions. Global KV remains in the persistent KV cache with a guaranteed lifetime of at least 72 hours.
2. Evicting SWA KV inevitably causes misses, which stay affordable thanks to a lightweight fallback, *Encoder SWA Bounded Replay* (detailed in Section 3.2.2). For the inevitable but infrequent requests that hit global KV but miss SWA KV, it recovers the missing state by recomputing only n_{win} tokens instead of a full $L \times n_{\text{win}}$-token forward pass. This bounded replay is the cornerstone of the design: it turns a catastrophic miss into a graceful, inexpensive degradation, thereby justifying the removal of SWA KV from the persistent KV cache.

3.2.2. *SWA Bounded Replay*

Since SWA dependencies accumulate across layers, exactly reconstructing the SWA KV of L layers would require replaying $L \times n_{\text{win}}$ tokens. SWA Bounded Replay instead replays only the most recent n_{win} tokens and truncates SWA to the replay segment, accepting approximate states: for a replay starting at position s, a query at position i attends to SWA keys in $\left[\max(s, i - W + 1), i\right]$.

Encoder SWA Bounded Replay. Encoder SWA Bounded Replay makes prefix caching depend only on global KV, allowing SWA KV to be removed from the persistent KV cache.

When the encoder SWA KV is missing, we replay the last n_{win} tokens of the cached prefix and process them together with the uncached suffix. The replayed tokens regenerate only SWA KV, reusing the cached global KV without recomputation or overwriting, while the uncached suffix generates both global KV and SWA KV.

By design, the replayed prefix state is approximate, so the global KV and SWA KV computed for the uncached suffix depend on the cache-hit position and are not mathematically identical across positions. Encouragingly, our experimental evidence confirms that this bounded replay strategy barely compromises response quality.

Decoder SWA Bounded Replay. Decoder SWA Bounded Replay bounds the decoder forward pass to n_{win} tokens, nearly halving total prefill computation.

Under CED, decoder global KV is projected from the final encoder hidden states. The only obstacle to ending prefill at the encoder is decoder SWA KV, which is generated from each decoder layer's own hidden states and is needed by the first decode steps. Since we never cache decoder SWA KV, exactly reconstructing it requires running the $\frac{L}{2}$ decoder layers over the last $\frac{L}{2} \times n_{\text{win}}$ prompt tokens, which is expensive when a short uncached suffix follows a long cached prefix. Therefore, we also apply the bounded replay strategy to this scenario: at every prefill, we replay the last n_{win} tokens of the prompt, feed their encoder outputs through the decoder layers under the same SWA truncation, and use the resulting decoder SWA KV only for decoding, not for prefix caching.

By design, the reconstructed decoder SWA KV is not mathematically equivalent to that from a full decoder forward pass. Also, we find that this strategy has only a negligible impact on response quality. For added safety, we additionally simulate the same replay during post-training for train-aware adaptation.

4. Pre-Training

4.1. Data Construction

Text Data Curation In pursuit of higher intelligence, we go beyond the general, sample-level quality reflected by small-scale data experiments and focus more on the holistic interactions among diverse corpora that offer unique information gains. We adopt a more systematic and standardized data construction pipeline to improve data quality and optimize the data mixture. Specifically, based on more comprehensive evaluations, a scaling ladder over model parameters and training data is carefully designed to guide large-scale training runs. We filter out model-generated content with limited information gain, including outputs from less capable models and low-quality machine-translated text. We regard such content as implicit duplication, as

it largely reformulates existing information and may become detrimental over long training horizons. We also explore model-in-the-loop data iteration approaches as a foundation for future large-scale synthetic data. In addition, we involve more domain experts to construct fine-grained data quality evaluation dimensions. Compared with the previous version, the new corpus incorporates more recent code from newly released open-source repositories, commits, libraries, and emerging frameworks to cover a broader range of programming languages and better reflects contemporary real-world software engineering scenarios.

Multimodal Data Curation Our multimodal pre-training dataset primarily comprises three types of data: image-text pairs, interleaved image-text data, and domain-specific data. Operating on the premise that raw web data naturally provides rich multimodal knowledge, we refrained from large-scale data synthesis; instead, we prioritized cleaning and utilizing the data in its native form to achieve the most direct and scalable visual knowledge compression during pre-training. During the initial data collection, we found that our crawling system was overly biased toward text-centric web content; we therefore re-bootstrapped it from Common Crawl to improve its coverage of multimodal sources. For image-text data, we extract images together with their associated alt text from webpages, filter them by applying an image-text relevance threshold, and deduplicate them based on image semantics. For interleaved data, we build this subset predominantly from webpages and PDFs. Processing large-scale multimodal corpora usually incurs higher CPU and disk storage costs than processing text-only corpora, which motivated us to organize the interleaved-data construction into progressively more expensive stages. Before image retrieval, we apply heuristic and statistical filtering, deduplication, and quality models to select high-value documents. The surviving documents are then assembled into interleaved image–text sequences, where filtering and deduplication are applied again in an image-aware manner. Finally, we employ SmolVLM (Marafioti et al., 2025) to conduct strict quality scoring on the image-text content, thereby extracting high-quality interleaved data. Documents filtered out during this process are partly recycled into additional image-text pairs via screening and recombination. To compensate for the inherent limitations of web-gathered data, we also incorporate domain-specific datasets to boost the model's capabilities in fine-grained visual perception (e.g., visual grounding and pointing), optical character recognition (OCR), and the acquisition of long-tail knowledge. We also collect extensive image-code pairs and computer-use trajectories to improve multimodal agentic understanding.

Data Integration and Deduplication As our text-only and multimodal data were processed through distinct pipelines, we constructed the final training corpus as the union of both data sources. For overlapping samples, we replace the text-only versions with their multimodal counterparts and use the larger epoch count of the two configurations. After this substitution, the resulting corpus uses a 7:1 token ratio of text-only to multimodal data. We minimize sample overlap during pre-training and context extension by jointly prefetching and assigning training samples. Ultra-long documents are deterministically pre-split before mixing to ensure a uniform distribution of training tokens across data shards and training steps. We further enhance our best-fit packing algorithm, achieving a padding rate of at most 10^{-4}.

4.2. Pre-Training Setups

4.2.1. Model Setups

We set the number of Transformer layers to 40 and the hidden dimension d to 5120. We adopt a Causal Encoder-Decoder architecture, with 20 layers in the encoder and 20 layers in the

decoder. For the first two layers, we use pure sliding window attention. The remaining 18 encoder layers use CSA2 with a compression rate of $m = 2$. These layers are divided into three identically configured groups of six layers. In each group, the first layer operates in Full Mode, and the remaining five layers operate in Reuse Mode. The 20 decoder layers use CSA2 with a compression rate of $m = 1$. These layers are divided into five groups of four layers. In the first group, the first layer operates in Full Mode, and the remaining three layers operate in Reuse Mode. The remaining four groups share the same configuration: the first layer operates in Reindex Mode, and the remaining three layers operate in Reuse Mode. For all CSA2 layers, we set the number of indexer query heads to 32, the indexer head dimension to 128, and the number of KV entries selected for sparse attention (i.e., attention top-k) to 512. We set the number of query heads to 64, the head dimension to 512, and the query compression dimension to 1280. For the Hierarchical Sparse Indexer, we select a maximum of 2,048 blocks with 8 positions, yielding up to 16,384 candidate positions in total. The number of output projection groups is set to 8, and the dimension of each intermediate attention output is set to 1024. For the additional branch of sliding window attention, the window size n_{win} is set to 128. We employ MoE layers in all Transformer blocks, using SwiGLU activation function with clamping (OpenAI, 2025) at a threshold of 10. Each MoE layer consists of 1 shared expert and 384 routed experts, where the intermediate hidden dimension of each expert is 2304. Among the routed experts, 6 experts will be activated for each token. As for *m*HC, the expansion factor is set to 4, and the number of Sinkhorn-Knopp iterations is set to 20. For the vision encoder, we set its number of layers to 32, the hidden dimension to 1024, the number of attention heads to 16, and the image patch size to 14. The vision MLP projector has 2 layers with a hidden dimension of 5120. Under this configuration, DeepSeek-V4.1-Flash comprises 552B backbone parameters, with 8B activated per token during prefill and 16B during decode.

4.2.2. Training Setups

We employ the Muon optimizer (Jordan et al., 2024; Liu et al., 2025) for the parameters of linear transformations, use AdamW optimizer (Loshchilov and Hutter, 2019) for the weights of all RMSNorm modules and other non-matrix parameters, and use Sinkhorn-balanced update for all embeddings and prediction head. For AdamW, we set its hyper-parameters to $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\varepsilon = 10^{-20}$, and weight_decay $= 0.1$. For Muon, we set the momentum to 0.95 and the weight decay to 0.1, and rescale the RMS of each update matrix to 0.18 for reutilization of the AdamW learning rate. For Sinkhorn-balanced updating, we use the same momentum coefficient and learning-rate correction factor as for Muon and set $K = 11$, $\tau = 10^{-3}$, $\varepsilon = 10^{-20}$. Following (Cheng et al., 2026b), the learning rate of Engram is scaled by 5×. We train DeepSeek-V4.1-Flash on 45T tokens of multimodal data with no instability. We keep the batch size fixed at 100.6 million tokens throughout training. The learning rate is linearly warmed up over the first 2000 steps and then maintained at 2.6×10^{-4} until 28T tokens. Between 28T and 40T tokens, we decay the learning rate to 2.6×10^{-5} following a cosine schedule. We keep the learning rate at this value from 40T to 45T tokens. We train the model from scratch with sparse attention at a sequence length of 64K and extend the sequence length to 1M at 34T tokens. For auxiliary-loss-free load balancing, we set the bias update speed to 0.001 for both image and text tokens, while retaining a small sequence-level balance loss with a loss weight of 0.0001 to avoid extreme imbalance within single sequences. Similar to DeepSeek-V4, we employ sample-level attention masking during pre-training.

Vision Encoder Training. Our DeepSeek-ViT undergoes a separate training stage before being integrated with the language backbone. The training pipeline consists of two stages: *contrastive*

pre-training and *autoregressive fine-tuning*. During contrastive pretraining, we optimize the model using the sigmoid contrastive loss introduced by SigLIP (Zhai et al., 2023) on approximately 47B image-text pairs sourced from alt-text data. To efficiently learn visual representations from such massive datasets, we restrict the maximum input resolution to 224×224 pixels by downscaling larger images while preserving their aspect ratios. Although using higher resolutions in this phase yields notable gains, empirical results show that these benefits contribute little to the final model. Because the subsequent autoregressive stage specifically handles high-resolution extrapolation, scaling up resolutions during contrastive pretraining significantly increases computational overhead without much overall improvement. In the autoregressive fine-tuning stage, we connect the vision encoder to a 4B MoE LLM and train on 236B tokens across datasets including image captions, alt text, charts, and OCR, using a next-token prediction objective. This stage aims to enhance the encoder's ability to model fine-grained visual features. We therefore constrain the input resolution between 544×544 and 1344×1344 pixels by proportionally scaling out-of-bound images. After this stage, we discard the LLM and retain only the optimized vision encoder for the subsequent pre-training pipeline, where the same input-resolution policy is maintained.

4.3. Evaluations

4.3.1. Evaluation Benchmarks

We compare DeepSeek-V4.1-Flash-Base with its predecessor models DeepSeek-V4-Flash-Base and DeepSeek-V4-Pro-Base. We report benchmarks spanning five key dimensions: world knowledge, language understanding and reasoning, coding and mathematics, long context, and multimodal abilities.

World knowledge benchmarks include AGIEval (Zhong et al., 2023), MMLU-Pro (Wang et al., 2024b), C-Eval (Huang et al., 2023), MultiLoKo (Hupkes and Bogoychev, 2025), SimpleQA-Verified (Haas et al., 2025) and SuperGPQA (Du et al., 2025),

Language understanding and reasoning benchmarks include BigBench Hard (BBH) (Suzgun et al., 2022), BigBench Extra Hard (BBEH) (Kazemi et al., 2025), DROP (Dua et al., 2019) and HellaSwag (Zellers et al., 2019),

Coding and mathematical benchmarks include BigCodeBench (Zhuo et al., 2025), HumanEval (Chen et al., 2021), GSM8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021) and MGSM (Shi et al., 2023),

Long context benchmark includes LongBench-V2 (Bai et al., 2025).

Multimodal benchmarks include MMMU-Pro (Yue et al., 2025), DocVQA (Mathew et al., 2021), CVBench (Tong et al., 2024) and RefCOCO/RefCOCO+/RefCOCO-g (Kazemzadeh et al., 2014; Nagaraja et al., 2016; Mao et al., 2016; Yu et al., 2016).

4.3.2. Evaluation Results

In Table 1, we provide a detailed comparison of the base models for DeepSeek-V4-Flash, DeepSeek-V4-Pro and DeepSeek-V4.1-Flash, all evaluated under our internal evaluation framework using strictly controlled and reproducible settings. Compared with DeepSeek-V4-Flash-Base and DeepSeek-V4-Pro-Base, our latest base model reveals a compelling efficiency gain. DeepSeek-V4.1-Flash activates a substantially smaller number of parameters than DeepSeek-V4-Pro-Base and occupies a heavily-reduced KV cache, yet its performance is fully on par

Table 1 | Comparison among DeepSeek-V4-Flash-Base, DeepSeek-V4-Pro-Base, and DeepSeek-V4.1-Flash-Base. All models are evaluated in our internal framework and share the same evaluation setting. Scores with a gap not exceeding 0.3 are considered to be at the same level. The highest score in each row is in **bold font**, and the second is underlined.

	Benchmark (Metric)	# Shots	DeepSeek-V4-Flash Base	DeepSeek-V4-Pro Base	DeepSeek-V4.1-Flash Base
	Architecture	-	MoE	MoE	MoE
	# Activated Params	-	13B	49B	8B/16B
	# Backbone Params	-	284B	1.6T	552B
World Knowl.	AGIEval (EM)	3-5-shot	83.9	**84.4**	83.4
	MMLU-Pro (EM)	5-shot	68.3	73.5	**74.1**
	C-Eval (EM)	5-shot	92.1	**93.1**	92.1
	MultiLoKo (LLM-Judge)	5-shot	42.6	**50.9**	45.5
	Simple-QA verified (EM)	25-shot	30.1	**55.2**	42.3
	SuperGPQA (EM)	5-shot	46.5	**53.9**	53.1
Lang. & Reas.	BBH (EM)	3-shot	86.9	**87.5**	86.1
	BBEH (EM)	1-shot	25.4	**29.8**	27.2
	DROP (F1)	1-shot	**88.6**	**88.7**	87.9
	HellaSwag (EM)	0-shot	85.7	**88.0**	87.2
Code & Math	BigCodeBench (Pass@1)	3-shot	56.8	59.2	**60.6**
	HumanEval (Pass@1)	0-shot	69.5	76.8	**79.4**
	GSM8K (EM)	8-shot	90.8	92.6	**93.0**
	MATH (EM)	4-shot	57.4	**64.5**	61.1
	MGSM (EM)	8-shot	**85.7**	84.4	80.2
Long Context	LongBench-V2 (EM)	1-shot	44.7	**51.5**	45.2
Multimodal	MMMU-Pro (EM)	4-shot	-	-	56.5
	CVBench (EM)	4-shot	-	-	77.9
	DocVQA (LLM-Judge)	4-shot	-	-	95.6
	RefCOCO-avg (Acc@0.5)	0-shot	-	-	86.0

with its predecessors. These results also reflect the substantial improvements we made to our pre-training data curation pipeline. In this version, we introduce native multimodal training and validate its effectiveness through corresponding multimodal evaluations. Trained on a more diverse and multimodal corpus, DeepSeek-V4.1-Flash achieves world knowledge and comprehension capabilities comparable to those of DeepSeek-V4-Pro. In reasoning and coding benchmarks, DeepSeek-V4.1-Flash shows consistent progress, reaching performance close to or better than DeepSeek-V4-Pro across multiple benchmarks.

To further assess the model's capabilities in real-world R&D scenarios, we additionally perform perplexity tests on dedicated internal corpora. Since perplexity tests are impossible through model APIs, we mainly focus on our own pretrained base models. For corpus selection, a separate evaluation set is collected based on our daily development, including internal documentation, proprietary code repositories and academic materials, which targets for reasoning, attribution and problem-solving on complex scientific problems and frontier research. The results are shown in Figure 6, where we report the bits-per-byte(BPB) of different models, with lower values indicating better performance.

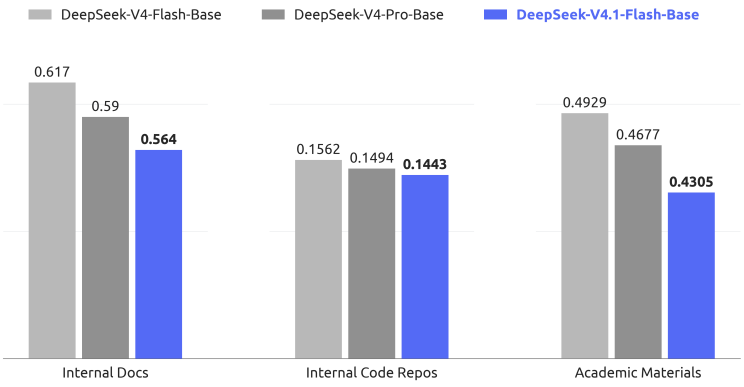

Figure 6 | Bits-per-bytes (BPB) comparison of DeepSeek-V4-Flash-Base, DeepSeek-V4-Pro-Base and DeepSeek-V4.1-Flash-Base on our held-out evaluation sets. DeepSeek-V4.1-Flash-Base achieves lowest BPB on all tasks and demonstrates greater potential to serve as a strong base-model.

5. Post-Training

5.1. Post-Training Pipeline

In this release, we refrain from introducing novel post-training algorithms. The overall recipe follows the standard paradigm of supervised fine-tuning (SFT) followed by reinforcement learning (RL) and on-policy distillation (OPD; Gu et al., 2024; Lu and Lab, 2025), without algorithmic modifications beyond well-established practices. Instead, our efforts are concentrated almost entirely on what the model is trained on rather than how it is optimized: we invest in large-scale, automated pipelines for data synthesis and environment construction. Concretely, the pipeline (i) synthesizes diverse, verifiable training tasks together with their reference solutions and reward signals, (ii) procedurally constructs and scales interactive agent environments in which trajectories can be collected and evaluated at low cost, and (iii) applies rigorous filtering, deduplication, and difficulty calibration to ensure data quality and curriculum balance. We find that, under a fixed and unremarkable optimization procedure, systematic improvements in the scale, diversity, and verifiability of synthesized data and environments account for essentially all of the observed gains. This observation echoes a broader lesson: at the current stage, the marginal return of engineering the data and environment pipeline substantially exceeds that of algorithmic novelty in post-training.

5.1.1. Large-Scale Agent Task Synthesis

Tasks serve as the fundamental fuel for agent learning. However, constructing high-quality training tasks has traditionally required substantial manual effort. We observe that the model is already beginning to exhibit the ability to construct its own training tasks, though this capability remains far from perfect. Recognizing this potential, we have invested considerable effort in strengthening the model's task-construction and quality-verification abilities.

We formalize each task as a triplet (problem, environment, verification system) and evaluate its quality along two dimensions: difficulty—ensuring the task is non-trivial—and correctness—guaranteeing that no critical flaws exist among the three components. Using difficulty and correctness as reward signals, we iteratively train the model to construct better tasks. We also monitor RL tasks across their full lifecycle. Whenever a task is used in a new RL run,

the resulting trajectories provide fresh evidence for quality re-auditing. Under this general framework, we have built dedicated training environment production pipelines for two core scenarios: general agents and coding agents.

General Agent. For general agents, we encourage internal employees and external partners to incorporate our latest model into their routine workflows and, on a voluntary basis, return interaction data and feedback. Based on the interfaces observed in the returned data, we construct a large set of mocked tools that reproduce the interfaces and behaviors of real-world tools and systems, including their input formats, output structures, API schemas, and behavioral constraints, covering both commonly used SaaS and enterprise applications as well as more specialized business back-end systems. In parallel, we collect negative feedback and model failure cases submitted by internal employees at scale, and incorporate them into the pipeline to generate both single-turn and multi-turn agent environments grounded in real workflows. By reconstructing the relevant tool context, user interaction patterns, and failure conditions, the pipeline enables systematic replay of failures and targeted reinforcement learning against observed model weaknesses.

Coding Agent. Coding agent training environments are built from two sources: 1) coding-agent sessions from internal employees and external partners, filtered to retain highly complex tasks or tasks on which model performance is poor, then deduplicated by trajectory; and 2) public GitHub repositories that meet a star-count threshold. Environment construction is carried out collaboratively by multiple specialized agents. First, an agent determines whether the project can be built and fully run inside a container and whether it can be automatically verified; if so, it selects a specific turn or commit as the task starting point, designs several sufficiently complex implementation directions, and produces concrete evaluation points, including both fail-to-pass and pass-to-pass points, along with a construction report, fetching external resources from the web as needed. Next, a separate agent sets up dependencies, the initial working directory, test code, and task descriptions in an isolated container, performs self-testing, removes any traces that could leak the task solution, and packages the environment as a new image layer. Then, multiple distinct agents attempt the task, and an independent quality-inspection agent reviews the environment together with the solving agents' trajectories, checking for environment issues, factual errors, mismatches between evaluation points and task descriptions, and hackability risks. If the inspection does not pass, a repair agent fixes all identified errors, adjusts evaluation points that are too easy or too difficult, and the task re-enters verification.

Through these pipelines, we can automatically and batch-produce RL training data that is correct, discriminative, and controllable in length and difficulty. Whether through the faithful reconstruction of real workflows in general agent environments or the precise construction of coding tasks in coding agent environments, both ultimately feed into a unified training system, driving iterative improvement of model capabilities under continuous quality monitoring.

5.1.2. RL in Synthesized Tasks

We use large-scale asynchronous RL in synthesized tasks to improve model performance and shape its behavior in complicated scenarios. We scale RL runs in two dimensions, i.e., training compute and the number of scaffolds. As shown in Figure 7 and Figure 8, performance continues to improve as we increase compute with cumulative RL steps, whether scaling within a single scaffold, jointly across variants of the same scaffold, or across heterogeneous scaffolds.

For RL training across diverse scaffolds, we decouple agent rollout execution into an agent sandbox and a worker container. The sandbox runs the scaffold and its tools, while the worker

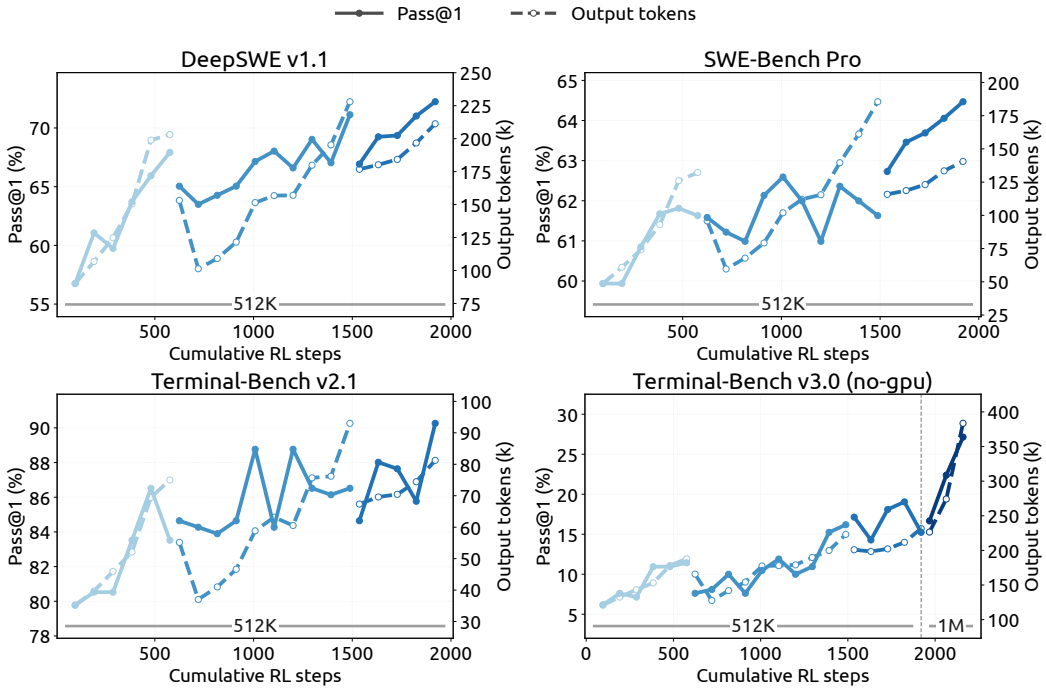

Figure 7 | Performance improves on various code agent benchmarks as RL training scales in the Minimal mode of DeepSeek Harness. Further extending maximal context length to 1M tokens continues to improve performance on extremely long-horizon tasks, e.g., Terminal-Bench v3.0.

provides a scaffold-agnostic control layer that orchestrates the rollout, normalizes heterogeneous interactions into a common trajectory schema, and communicates with the trainer. Both run on DSec (Section 5.1.3), outside the preemptible GPU training pool, separating long-lived rollouts from fine-grained training scheduling. During trainer preemption, rollout execution can be suspended and offloaded while preserving its full state for later resumption and releasing CPU and GPU resources. This design enables stable and efficient RL across heterogeneous scaffolds without modifying the underlying algorithms.

To extend effective RL compute beyond a single training run, we use model merging to reinitialize successive RL runs. Specifically, we merge checkpoints from runs across different scaffolds or configurations, combining improvements acquired along different optimization paths. In Figure 7 and Figure 8, disconnected curve segments reflect successive RL runs after model reinitialization. This yields further gains in both task performance and token efficiency, providing a simple and practical way to aggregate parallel RL compute and continue scaling across successive runs.

5.1.3. Running Agents at Massive Scale: DSec

As we transitioned from DeepSeek-V3 to V4, the rapidly growing number and diversity of agentic training environments motivated us to build *DeepSeek Elastic Compute (DSec)*, a production-grade sandbox platform for large-scale agentic training and evaluation. Its initial design addressed heterogeneous execution environments, scalable image distribution, multiple isolation backends, high-density resource management, command trajectory logging, and preemption-safe resumption.

V4.1 training further increased demand to millions of concurrent sandbox instances span-

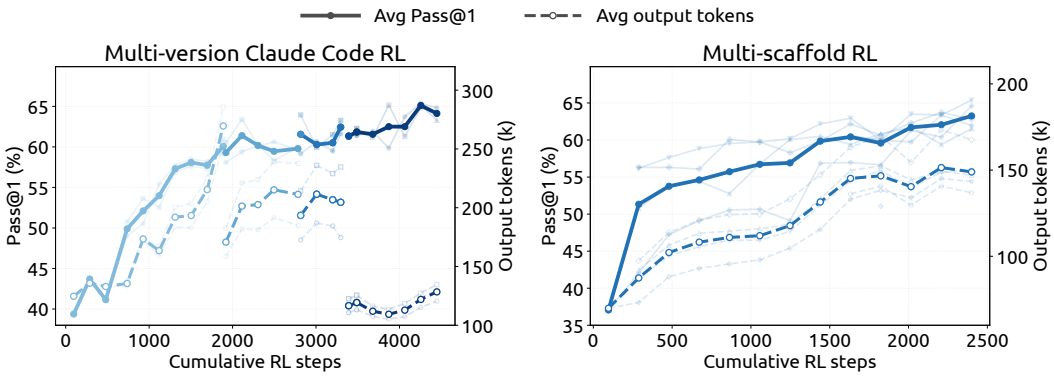

Figure 8 | Performance improves with cumulative RL steps when jointly training across multiple versions of Claude Code (left) and across heterogeneous scaffolds, including OpenCode, Pi, and DeepSeek Harness in Standard and PTC modes (right). Performance is evaluated on DeepSWE v1.1. Lighter curves show the evaluation of individual scaffold versions or scaffolds.

ning diverse harnesses, platforms, code repositories, software dependencies, and task-specific services. At this scale, the primary bottlenecks shifted toward datacenter scalability, workload isolation, per-node compute density, and the containment of misbehavior by increasingly capable agents. We briefly describe our key design aspects below.

Horizontally Scaling Compute at Scale. DSec scales through two complementary mechanisms: sharding and scheduling with relaxed consistency. To accommodate a large number of machines, we partition compute nodes into multiple shards (so-called scale units). Such sharding also allows us to reduce blast radius by isolating workloads from different experiments, preventing a single memory-intensive task from exhausting resources shared by unrelated work.

Instead of using off-the-shelf orchestrators like Kubernetes, DSec employs a custom placement engine to schedule the large number of sandboxes, by trading strong global consistency for scalability. This design is based on a key observation: agentic sandbox placements only demand eventual consistency as long as each compute node enforces local safety constraints. To do so, the placement engine deploys in multiple independent replicas, without synchronized coordination. Each replica predicts resource availability from recent measurements and makes good-enough placement decisions. To compensate for this loss in consistency, each node is responsible for validating the final placement decisions, enforcing a hard admission constraint that rejects new placements if it exceeds a local warning threshold. This design as a whole allows DSec to scale to millions of containers without bottlenecking on central coordination.

Running Sandboxes at High Density. At the node level, we use hardware-supported sub-NUMA partitioning and bind each worker VM to an individual NUMA domain. Containers run within these worker VMs, with their CPU and memory allocations confined to the VM's local NUMA resources. This setup balances aggressive memory overcommitment against Linux kernel lock contention, while localizing memory pressure and runtime failures. Under comparable workload configurations, it increases the supported density from roughly 1,000 to more than 2,500 concurrent live containers per physical node before measurable end-to-end degradation appears.

Such high-density deployment can nevertheless distort time-sensitive evaluations through interference from background workloads. DSec therefore introduces a latency-sensitive (LS) execution class. We apply `SCHED_IDLE` to non-LS tasks to minimize their scheduling priority,

and use core scheduling to ensure only tasks of the same priority class execute simultaneously on sibling hyperthreads to eliminate interference.

Mitigation of Misbehaving Agents. During RL training, we frequently observe agents attempting to perform reward hacking or unintentionally crashing the environment. In certain attempts, our agents exploited recently disclosed vulnerabilities, including permission issues from the XFS driver, illegal memory access in AppArmor, leaking answers from package mirror services, and so on. Agents have also been notorious for deleting critical binaries, breaking system files, or even removing the filesystem. We use per-sandbox AppArmor profiles and fine-grained eBPF-based network policies to prevent such attempts. If an agent crashes its environment, we treat the crash as a failed trajectory and report a "repercussion" signal to the RL framework.

5.1.4. Controllable Reasoning Effort in RL

Alongside advances in model architecture and hardware, the number of output tokens is another key determinant of serving cost and, consequently, of the cost–quality trade-off in real-world applications. We therefore introduce a scalar effort level b as an explicit conditioning signal during reinforcement-learning. This mechanism is applied to both single-turn reasoning and multi-turn agentic tasks. Specifically, we prepend the following instruction to the system prompt:

```
Reasoning Effort: {effort} (range 1-100; higher values request more
                  thorough reasoning)
```

Here, $b \in \{1, \ldots, 100\}$ denotes the requested effort level.

For each training prompt x, we sample M_b responses at each effort level $b \in \mathcal{B}$:

$$z_{b,j} \sim \pi_\theta(\cdot \mid x, b), \qquad b \in \mathcal{B}, \quad j = 1, \ldots, M_b. \tag{8}$$

Here, j indexes the responses sampled at effort level b. Responses sharing the same (x, b) form a subgroup, within which rewards are mean-centered to compute group-relative advantages. Thus, responses from different effort levels are not directly compared. Instead, effort-dependent behavior is induced within each subgroup by making the length component of the reward depend on b. Specifically, we add the length-penalty term $r^{\text{len}}_{b,j}$ to the reward of response $z_{b,j}$:

$$r^{\text{len}}_{b,j} = -\min\left\{C_{\max},\ k(b)\frac{\ell_{b,j}}{L_{\text{norm}}}\right\}, \tag{9}$$

where $\ell_{b,j}$ is the number of reasoning tokens, L_{norm} is a reference length, and $C_{\max}$ caps the maximum deduction applied to a trajectory. The token-penalty coefficient decreases exponentially with the requested effort:

$$k(b) = k_0 \exp\left(-\frac{b - b_{\min}}{\tau}\right), \qquad \tau = \lambda\overline{\Delta b}, \tag{10}$$

where k_0 is the basic penalty coefficient at different effort levels, b_{min} is the minimum value of $\mathcal{B}$, $\overline{\Delta b}$ is the average spacing between training effort levels, and λ controls the rate of penalty decay. Increasing b by τ multiplies the penalty coefficient by e^{-1}. The parameter k_0 controls the overall pressure toward shorter reasoning, whereas a smaller τ causes the penalty to decay more rapidly and tends to produce greater behavioral separation between effort levels. Appendix C provides a marginal-utility motivation for the exponential parameterization of $k(b)$.

At deployment time, the scalar b provides a flexible control interface over the model's reasoning strength. By varying b, a single model checkpoint can move between different operating regimes along the learned cost–quality frontier, adapting to different latency, token-budget, and solution-quality requirements. Although training uses only a finite set of effort levels, intermediate values can be used at deployment to elicit interpolated reasoning behaviors, providing a fine-grained and efficient mechanism for test-time resource allocation.

In our production deployment launched in September 2026, the public API exposes three preset reasoning-effort tiers—`max`, `high`, and `low`—which map directly onto this scalar interface. As summarized in Table 2, the three tiers correspond to effort values of $b = 100$, $b = 75$, and $b = 50$, respectively, so that API users select an operating point on the learned cost–quality frontier without any change to the model weights or decoding configuration.

Table 2 | Mapping between the public API reasoning-effort tiers and the underlying scalar effort values b.

API tier	Effort value b
`max`	100
`high`	75
`low`	50

5.2. Asynchronous Post-training Infrastructure

The long-tail problem during the rollout phase of RL for LLMs has consistently been a major bottleneck for training efficiency. To address this, we extend our post-training infrastructure to allow asynchronous generation of samples (Zeng et al., 2026; Team et al., 2026b), which significantly mitigates the long-tail issue in the rollout phase by maintaining a sufficiently high level of concurrency. Asynchronous training is now enabled for nearly all our RL and OPD tasks, and rollout efficiency has improved substantially.

5.2.1. Overall Workflow

We colocate rollout and training on the same physical devices and time-share their execution, eliminating the need to manually tune resource allocation between the two phases. Each task specifies an upper bound on the number of in-flight samples, and the system maintains this bound throughout the rollout phase.

We evaluated three dispatch granularities for maintaining the target rollout concurrency. Our final approach is sample-level dispatch: once the number of newly completed samples reaches the GRPO group size assigned to the next prompt, we dispatch that prompt regardless of which groups produced those completions. This helps maintain a steady rollout concurrency throughout training. Before settling on this dispatch strategy, we experimented with two alternative dispatch granularities. In the first attempt, we dispatched several extra batches at the beginning and supplemented a full batch after each training iteration; however, this caused severe oscillations in training metrics, indicating that batch-level granularity was too coarse. In the second attempt, we switched to prompt-level dispatch, where a new prompt was dispatched after one GRPO group finishes, but found that it stalled easily on long-tail samples within a GRPO group, making it difficult to smoothly maintain the target rollout concurrency.

Once enough training samples have accumulated, training preempts ongoing rollouts. Dur-

ing training, we use concatenated routing-replay: for samples that span multiple checkpoints, we concatenate the expert routing produced at each rollout segment rather than discarding and recomputing the routing information with new checkpoints.

5.2.2. *Mitigating Length Bias and Off-Policy Effects*

Asynchronous generation, while effectively improving rollout efficiency, introduces two side effects that can degrade training quality. First, it creates a length-distribution bias, especially in the early training stage, because shorter sequences tend to complete first and thus dominate the initial training batches. Second, it inevitably produces off-policy samples, i.e., samples whose tokens are partially or entirely generated by earlier checkpoints. These two issues require different handling strategies.

To address the length bias, we employ two mechanisms. First, the dispatcher can limit concurrency on a per-dataset basis, which helps regulate the proportion of each dataset in the steady-state training batch, indirectly mitigating the length skew by controlling the sources of incoming samples. Second, we support discarding early-returned short samples to smooth the transition into the steady-state length distribution, and prevent the model from overfitting to overly short sequences.

For the off-policy issue, we implement two additional mechanisms. First, by tuning the logic that controls sample dispatching and the waiting condition for training samples, we can bound the maximum off-policy ratio, ensuring that the training data does not deviate excessively from the current model. Second, during training, we add a loss masking scheme that eliminates the contribution of tokens with excessive staleness, thereby mitigating the adverse impact of stale samples on gradient updates.

5.2.3. *Performance Optimization*

In our asynchronous RL framework, the rollout phase is periodically interrupted to switch to updated policy checkpoints. We aim to make this process seamless: interruptions should be near-instantaneous, and interrupted rollouts should resume as if never stopped.

To stop rollouts promptly, we support token-level interruption: generation can be halted at any token boundary. Once sufficient training data has been collected, all in-flight samples stop almost immediately, allowing the system to enter the training phase without delay.

To preserve rollout progress across checkpoint switches, rollout states such as KV cache and expert routing are persisted at token granularity during generation. Upon resumption with a new checkpoint, the persisted states are directly reused, eliminating the cost of re-prefilling and allowing interrupted samples to continue exactly where they left off. Since this requires retaining the states of all in-flight samples, we perform sample-grained garbage collection, releasing each sample's states as soon as it completes.

Beyond checkpoint switching, the same fast-interruption and seamless-resumption machinery allows the training jobs to respond promptly to cluster scheduling preemption signals without losing progress, thereby improving overall cluster utilization.

5.2.4. *Large-Scale On-Policy Distillation*

As the last stage of post-training, the final full-vocabulary OPD task is trained on datasets from all domains using over 40 teacher models. It also adopts asynchronous generation to improve

rollout efficiency. Due to differences in training procedures across domains, the best teacher for each domain may come from a different stage of model development. Moreover, the teacher models may differ architecturally from one another and from the student. Our post-training infrastructure readily accommodates this setting, supporting full-vocabulary OPD with an effectively unbounded number of architecturally heterogeneous teachers, and efficient switching among them at negligible cost (DeepSeek-AI, 2026b).

The OPD stage also requires dynamic reconfiguration during training. We continuously track model capabilities and may adjust the training recipe accordingly, including the dataset mixture, per-dataset concurrency limits, and active teachers. Such changes are straightforward in synchronous training where rollout batches provide explicit configuration boundaries. In the asynchronous setting, however, samples generated under different configurations may coexist in flight. Our infrastructure supports consistent transitions between configurations without disrupting rollout or training.

5.3. Evaluation

5.3.1. Evaluation Setup

Our post-training evaluation focuses primarily on reasoning and agentic capabilities, while knowledge-intensive performance is largely determined by pretraining and reported in Table 1. For reasoning, we evaluate on GPQA Diamond (Rein et al., 2023), Humanity's Last Exam (Phan et al., 2025), Codeforces (internal benchmark), and MathArena Apex (Dekoninck et al., 2025), using temperature and top-p of 1.0. For agentic capabilities, we evaluate across four categories:

- **Code agent**: Terminal-Bench 2.1 (Merrill et al., 2026), Terminal-Bench 3.0 (Marten et al., 2026b), Terminal-Bench 4.0 (Marten et al., 2026a), DeepSWE v1.1 (DataCurve, 2026), ProgramBench (Yang et al., 2026), NL2Repo-Bench (Ding et al., 2025).
- **Cyber security**: SEC-Bench Pro version 260505 (Lee et al., 2026), CyberGym (Wang et al., 2026c), and ExploitGym (Wang et al., 2026b).
- **General agent**: the public evaluation set of AutomationBench v1.0.6 (Shepard and Salimans, 2026), Agents' Last Exam (Sun et al., 2026a) (ALE-CLI).
- **Visual agent**: Chartography (Garre et al., 2026), BabyVision (Chen et al., 2026), main set of ZeroBench (Roberts et al., 2025).

For code agents, we evaluate DeepSeek-V4.1-Flash using the Minimal mode of DeepSeek Harness with a 1M-token context window, temperature set to 1.0, and top-p set to 0.95. To align with official setup requirements, we employ the mini-SWE harness for DeepSWE v1.1. For SEC-Bench Pro, we utilize the Claude Code harness specifically for its session compact design. For visual agent tasks, we evaluate using the Claude Code harness with a 512k-token context window, temperature set to 1.0, and top-p set to 0.95. Agents' Last Exam and AutomationBench are evaluated with their official scaffolds. Model performance with other coding scaffolds is reported in Table 4.

To mitigate reward hacking in coding agent evaluations, we restrict internet access and strip Git histories from the environment. Additionally, we automatically purge transient build and package caches across diverse environments, including Go module caches (go/mod), node modules dependency artifacts, compiled .jar files, and Python pycache directories. Despite these precautions, we still observe instances of exploit-seeking behavior during testing—such as decompiling core Ubuntu Linux packages to uncover vulnerabilities in CyberGym. As models grow increasingly capable, standard evaluation infrastructure (e.g., Docker containers

and validation scripts) becomes more susceptible to model gaming. We urge the broader research community to prioritize detecting and mitigating these behaviors when designing next-generation benchmarks.

5.3.2. Evaluation Results

Table 3 | Comparison between DeepSeek-V4.1-Flash with closed/open source models. † denotes text-only subset of HLE. The best results are highlighted in bold; the second-best results are underlined.

	Benchmark (Metric)	Opus-5 Max	GPT-5.6 Sol Max	K3 Max	GLM-5.3 Max	DS-V4-Pro Max	DS-V4-Flash Max	DS-V4.1-Flash Max
Reasoning	GPQA Diamond (Pass@1)	93.4	**94.1**	92.9	88.1	92.4	89.9	90.9
	HLE (Pass@1)	**56.3**	44.5	43.5	42.0†	42.7†	37.8†	36.8 (39.1†)
	Codeforces (Rating)	-	-	-	-	3348	3289	**3471**
	MathArena Apex (Pass@1)	-	-	**65.6**	-	65.3	58.6	**65.6**
Agentic	Terminal-Bench 2.1 (Pass@1)	89.1	88.8	88.3	88.2	87.9	82.7	**90.6**
	Terminal-Bench 3.0 (Pass@1)	**43.3**	34.4	17.7	28.3	11.8	7.6	30.0
	Terminal-Bench 4.0 (Pass@1)	**51.8**	39.9	12.6	37.9	12.4	7.0	31.2
	DeepSWE v1.1 (Resolved)	74.0	73.0	67.5	66.9	62.7	54.4	**74.2**
	ProgramBench (Almost@1)	**37.0**	23.0	17.5	19.0	15.5	-	20.3
	NL2Repo-Bench (Score)	**75.3**	56.8	58.0	58.0	61.5	54.2	65.4
	CyberGym (Pass@1)	-	84.5	80.0	84.5	83.3	76.7	**88.1**
	SEC-Bench Pro (Pass@1)	-	**74.3**	-	-	56.4	30.9	62.8
	ExploitGym (Pass@1)	22.1	**33.7**	-	15.0	5.4	1.8	15.3
	HLE w/ tools (Pass@1)	63.6	-	59.8	62.5	60.0	51.5	**63.9**
	Automation-Bench (Pass@1)	50.3	45.8	46.7	48.8	43.2	37.7	**54.8**
	Agents' Last Exam(Pass@1)	28.6	26.7	27.6	28.5	25.7	25.2	**31.8**
	Chartography w/ tools (Pass@1)	**84.0**	79.9	68.1	-	-	-	78.9
	BabyVision w/ tools (Pass@1)	**94.1**	88.9	85.7	-	-	-	89.6
	ZeroBench-main w/ tools (Pass@5)	52.0	**53.0**	41.0	-	-	-	49.0

As detailed in Table 3, DeepSeek-V4.1-Flash exhibits significant performance upgrades across both reasoning and agentic benchmarks over its predecessor, DeepSeek-V4-Flash, while matching or outperforming top-tier open-source and proprietary models.

In core reasoning tasks, DeepSeek-V4.1-Flash achieves a Codeforces rating of 3471, surpassing both DeepSeek-V4-Flash (3289) and DeepSeek-V4-Pro (3348). On MathArena Apex, it obtains a 65.6% Pass@1 accuracy, fully matching the top-performing open-source baseline Kimi-K3 (65.6%) and DeepSeek-V4-Pro (65.3%). Furthermore, its GPQA Diamond score reaches 90.9%, showing steady improvements over DeepSeek-V4-Flash (89.9%).

The performance gains are even more pronounced across agentic tasks. Notably, on DeepSWE v1.1, DeepSeek-V4.1-Flash reaches 74.2% pass rate, marking a substantial jump from DeepSeek-V4-Flash (54.4%) and surpassing leading proprietary models including Opus-5 (74.0%) and GPT-5.6 Sol (73.0%). On Terminal-Bench 2.1, it achieves 90.6%, outperforming Opus-5 (89.1%) and GLM-5.3 (88.2%). Similarly, on Automation-Bench (54.8%) and Agents' Last Exam (31.8%), DeepSeek-V4.1-Flash establishes leading scores over both open-source counterparts and top closed-source systems. Despite its compact nature, DeepSeek-V4.1-Flash demonstrates state-of-the-art agentic capabilities, substantially closing the gap with frontier closed-source models while establishing clear advantages among open-source alternatives.

On cyber-security tasks, DeepSeek-V4.1-Flash establishes a new state of the art among open-source models. Given the dual-use nature of these capabilities, we encourage the community to

apply them responsibly, such as for defensive security research and vulnerability remediation. In the domain of visual agent tasks, DeepSeek-V4.1-Flash demonstrates robust capabilities, particularly in scenarios requiring visual reasoning and the analysis of complex professional charts. While it outperforms the leading open-source model Kimi-K3, we acknowledge that a measurable gap still remains when benchmarked against the leading closed-source alternatives.

Beyond peak performance, DeepSeek-V4.1-Flash exposes a reasoning-effort setting that allows users to trade inference cost for accuracy in a controllable manner. As illustrated in Figure 9, both accuracy and output length increase steadily with the effort level across reasoning and agentic benchmarks alike. Raising the effort from 25 to 100 improves the average Pass@1 on eight reasoning-intensive benchmarks from 67.1% to 76.3%, on DeepSWE v1.1 from 66.0% to 74.2%, and on Terminal-Bench 2.1 from 82.4% to 90.6%, at the cost of roughly 2.5× more output tokens. Notably, the effort control learned on single-response reasoning transfers faithfully to long-horizon agentic trajectories, where it governs the total amount of exploration and verification across turns. The gains are front-loaded: the 60–80 range already recovers most of the accuracy of the maximum setting at less than half of its token budget, whereas the final step to effort 100 lengthens agent trajectories by 1.6–1.8× for only marginal improvements. The maximum tier is thus best reserved for the most challenging tasks, while moderate effort levels offer a favorable cost–performance balance for everyday agentic use.

5.3.3. Performance across reasoning efforts

Figure 9 reports performance and average response length across a range of budget values. As the reasoning effort increases, the model produces progressively longer reasoning traces and accuracy improves monotonically on reasoning-intensive benchmarks and software engineering tasks, with the largest gains concentrated in the low-to-mid budget range and diminishing returns beyond. Although our RL involves only a limited number of effort levels, the use of scalar efforts achieves flexible, interpolated control of response length within a specific range. This allows practitioners to trade off quality against latency and token cost along a smooth continuum: latency-sensitive applications can operate at low effort with modest accuracy degradation, while difficult tasks can invoke high effort to recover the model's full reasoning capability. In our public API service, we expose three preset effort levels that map onto this scale: `low`, `high`, and `max` correspond to effort values of 50, 75, and 100, respectively.

5.3.4. Performance across agent scaffolds

In practice, a model is rarely deployed within a single fixed agent framework; different scaffolds vary in their system prompts, tool definitions, context management strategies, and interaction protocols, and a model that overfits to one particular harness may degrade substantially when placed in another. To assess the robustness of our model to such variation, our comparison covers eight configurations from six scaffold families: Claude Code (Anthropic, 2026), Codex (OpenAI, 2026), OpenCode (Anomaly, 2026), Pi (Zechner, 2026), mini-SWE (Yang et al., 2024), and DeepSeek Harness (DSH) (DeepSeek-AI, 2026a) in Minimal, Standard, and PTC modes. For each scaffold, we keep the model checkpoint, decoding configuration, and task set identical, and only the surrounding harness, including its native system prompt, tool schema, and turn-taking logic, is changed. Table 4 reports performance at Max reasoning effort (100) on DeepSWE v1.1 and Terminal-Bench v2.1.

The model's agentic capabilities transfer well across scaffold families with different prompts and tool interfaces, rather than depending on conventions specific to a particular harness. Its performance remains robust as the surrounding interaction protocol and tool abstractions

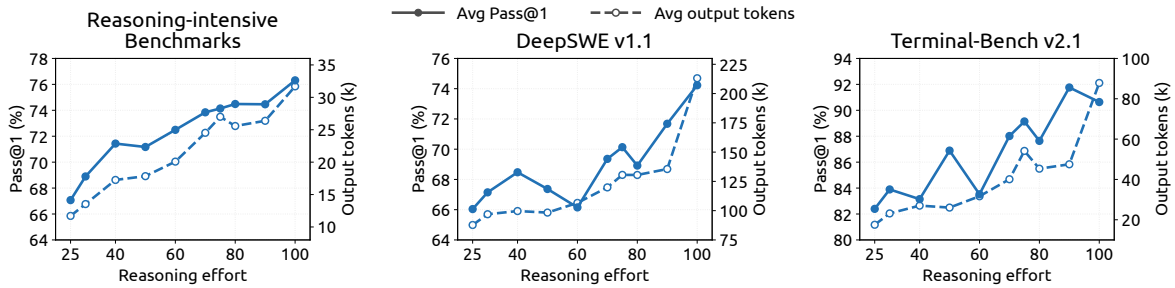

Figure 9 | Performance and output length as a function of reasoning effort. Each panel plots Pass@1 (solid, left axis) and mean output tokens per response (dashed, right axis) as the reasoning-effort value is varied from 25 to 100; The results of reasoning-intensive benchmarks are averaged over eight benchmarks (AIME 2026, Apex 2025 Shortlist, GPQA Diamond, HLE, IMO-AnswerBench, LiveCodeBench, MathArena-Apex, SimpleQA-Verified). DeepSWE v1.1 is evaluated based on mini-SWE and Terminal-Bench v2.1 is evaluated based on DeepSeek Harness (Minimal).

Table 4 | Performance across agent scaffolds at Max reasoning effort.

Benchmark (Metric)	**Claude Code**	**Codex**	**OpenCode**	**Pi**	**mini-SWE**	**DeepSeek Harness**		
						Minimal	**Standard**	**PTC**
DeepSWE v1.1 (Resolved)	69.8	65.6	65.5	66.2	74.2	72.6	70.5	67.6
Terminal-Bench v2.1 (Pass@1)	88.0	84.1	85.0	86.1	90.3	90.6	85.8	85.8

Note. All scaffolds use $N = 8$ samples per task on DeepSWE v1.1 and $N = 3$ on Terminal-Bench v2.1. Runs use Linux containers, temperature 1.0, top-p 0.95, a 1M-token context window, and `max_steps=500` model-generation rounds per agent on both benchmarks. Terminal-Bench v2.1 is evaluated without network access. We evaluate four Claude Code versions, with per-version results and their average reported in Appendix Table 5; this table reports v2.1.251. More scaffold-specific configurations are detailed in Appendix B.1.

change, indicating that its agentic behavior is not tightly coupled to a single scaffold design. This robustness is consistent with the diversity of environments, tool schemas, and interaction formats in our synthesized training data (Section 5), which is designed to encourage generalization across agent scaffolds.

5.3.5. Multi-Agent

To explore multi-agent collaboration on complex tasks, we conduct preliminary experiments with DeepSeek-V4.1-Flash using DeepSeek Harness's Agent Team mode.

Multi-Agent Harnesses. We use DeepSeek Harness in Agent Team mode, where a lead agent can asynchronously create named, persistent teammates by default through `spawn_teammate`. Each receives a delegated task and starts in either `fresh` mode without lead history or `fork` mode with a one-time snapshot of the lead's completed turns. All agents share one repository checkout, making edits immediately visible to one another.

Agents communicate through a durable peer mailbox. A message sent through `send_message` reaches a running teammate at its next step boundary, starts a new turn for an idle teammate, or resumes an inactive teammate. Across each teammate's task lifecycle, the lead monitors runtime status with `list_agents` and waits for status, mailbox, or shared-task changes with `wait_agent`. Task ownership, dependencies, and advisory write scopes are maintained on a shared task board (using the four `team_task_*` tools with revision checks on updates).

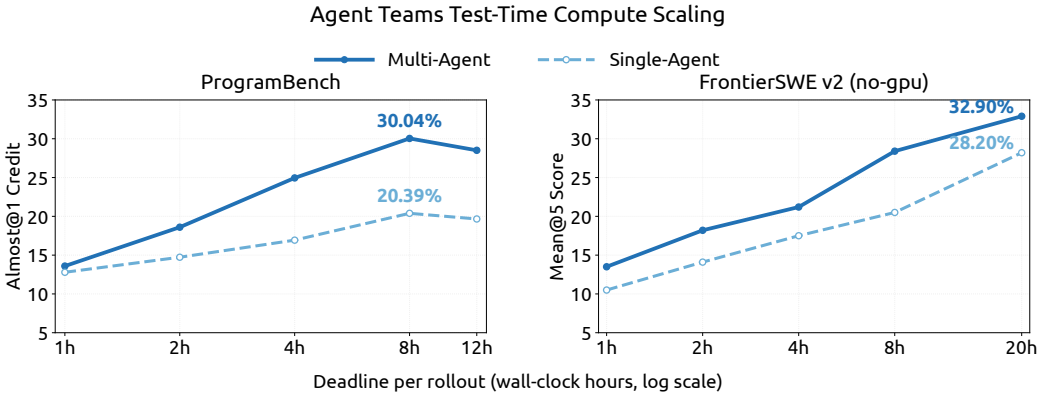

Figure 10 | Test-time compute scaling for single-agent and multi-agent configurations on ProgramBench (Almost@1) and FrontierSWE v2 (Mean@5) as functions of the per-rollout wall-clock deadline.

When intervention is needed, only the lead can interrupt a teammate's current turn through `interrupt_agent`. Once the required work is complete, the lead reviews and tests the combined changes and produces the final response.

Training. We train Agent Team mode with an RL reward combining task performance, a collaboration bonus that encourages delegation and inter-agent communication, and a derived-latency penalty that promotes efficient coordination. Derived latency is computed by representing execution events and their collaboration dependencies as a directed acyclic graph (DAG), assigning costs from token counts at fixed prefill/decode rates plus measured tool-execution time, and taking the length of the critical path. This encourages useful parallelism while penalizing unnecessary sequential work and synchronization, with reduced sensitivity to serving-side batching and queuing delays.

Performance. We construct a high-confidence subset of ProgramBench (Yang et al., 2026) by retaining only tasks for which the reference solution achieves a pass rate of at least 95% on the hidden test suite. This filtering procedure leaves 172 "golden" tasks. We also evaluate on FrontierSWE v2 (Kondra et al., 2026), a larger and more challenging successor to FrontierSWE that uses a substantially improved methodology. We construct a no-GPU subset from the currently public tasks by excluding tasks that require GPU access. We evaluate single-agent and multi-agent configurations on both benchmarks under explicit per-rollout wall-clock deadlines. The results reported here are preliminary: we compare the strongest observed multi-agent configurations with the strongest available single-agent baselines. On ProgramBench, we run up to three rollouts per task, corresponding to 516 planned rollouts for each configuration. We report Almost@1, which measures the fraction of individual rollouts achieving a score of at least 0.95. On FrontierSWE v2, we report Mean@5. ProgramBench deadlines range from 1 to 12 hours, while FrontierSWE v2 is evaluated at deadlines ranging from 1 to 20 hours. At each deadline, metrics are computed from the outputs available when the deadline is reached.

As shown in Figure 10, multi-agent configurations outperform their single-agent counterparts at every deadline on both benchmarks. On ProgramBench, Almost@1 increases from 13.59% at 1 hour to a peak of 30.04% at 8 hours for the multi-agent configuration, compared with 12.79% and 20.39% for the single-agent configuration. On FrontierSWE v2, Mean@5 increases from 13.50% at 1 hour to 32.90% at 20 hours for the multi-agent configuration, compared with an increase from 10.50% to 28.20% for the single-agent configuration.

6. Conclusion, Limitations, and Future Directions

In this work, we introduce DeepSeek-V4.1-Flash, a multimodal Mixture-of-Experts (MoE) model with support for contexts of up to one million tokens. Through joint optimization of model architecture, cache precision, and deployment strategy, DeepSeek-V4.1-Flash pushes the limits of KV cache compression. Its Causal Encoder-Decoder (CED) architecture enables the model to activate only 8B parameters per token during prefill, compared with 16B during decode, improving cost efficiency for input-heavy agentic workloads. At equal sequence lengths, cross-layer KV cache reuse in Compressed Sparse Attention 2 (CSA2) and FP4 KV caching reduce its global KV cache footprint (always in HBM) to 890 bytes per token, roughly 1/4 of the corresponding footprint of DeepSeek-V4-Flash. SWA Bounded Replay further reduces its persistent KV cache footprint (always on SSD or in host memory) to roughly 1/8 of that of DeepSeek-V4-Flash. These reductions alleviate HBM and SSD capacity pressure while the model delivers substantially better overall performance than DeepSeek-V4-Flash. Despite possessing a significantly smaller parameter footprint than contemporary open-source models such as GLM-5.3 and Kimi-K3, DeepSeek-V4.1 achieves comparable—and in several tasks, superior—performance across key benchmarks.

Although DeepSeek-V4.1-Flash substantially simplifies several architectural components relative to DeepSeek-V4-Flash, the newly introduced architectural changes also create robustness boundaries that have yet to be fully characterized. Our internal evaluations cover a diverse range of test cases and boundary conditions, and we have not observed any systematic degradation in model capabilities in the evaluated settings. Nevertheless, no finite test suite can cover every extreme input and deployment condition. Potential selection errors in CSA2 and approximate state reconstruction in SWA Bounded Replay may still cause capability degradation in untested boundary cases. Going forward, we will continue to expand our stress-testing and evaluation stack, with particular attention to sparse retrieval over long contexts and SWA state reconstruction at cache-resumption boundaries. We will also monitor real-world workloads, systematically characterize potential failure modes and robustness boundaries, and further improve model robustness under extreme conditions.

As AI models achieve remarkable performance capabilities, standard evaluation benchmarks have increasingly reached saturation. While DeepSeek-V4.1-Flash demonstrates performance that closely approaches top-tier models like Fable-5 and GPT-6 Astra—offering a highly comparable user experience in daily applications—a performance gap remains on the most challenging tasks. Although benchmark scores show a narrow margin, this parity does not imply that the model matches the frontier capabilities of leading closed-source systems on complex, high-difficulty reasoning and edge cases.

Consequently, we will continuously update our evaluation protocols to ensure rigorous assessment of state-of-the-art reasoning boundaries. Alongside continued efforts to reduce model costs, we believe that further advances in model intelligence will depend on the coordinated scaling of data, model capacity, and RL. With DeepSeek-V4.1-Flash as a new starting point, we will continue to explore the limits of model capabilities and systematically address key challenges in large-scale data synthesis and RL scaling. We will also actively integrate model–harness co-design, enabling the joint system to evolve and be optimized together. By advancing cost reduction and capability scaling in tandem, we hope to make highly capable agents more accessible and easier to deploy, further lowering the barriers to adopting AI technologies across a broader range of industries and scenarios.

References

J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. arXiv preprint arXiv:2305.13245, 2023.

E. Alvarez, O. Almog, E. Chung, S. Layton, D. Stosic, R. Krashinsky, and K. Aubrey. Introducing nvfp4 for efficient and accurate low-precision inference, 2025. URL https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/.

Anomaly. Opencode. https://github.com/anomalyco/opencode, 2026.

Anthropic. Claude code. https://code.claude.com/docs/en/overview, 2026.

Y. Bai, S. Tu, J. Zhang, H. Peng, X. Wang, X. Lv, S. Cao, J. Xu, L. Hou, Y. Dong, et al. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 3639–3664, 2025.

Y. Bai, Q. Dong, T. Jiang, X. Lv, Z. Du, A. Zeng, J. Tang, and J. Li. Indexcache: Accelerating sparse attention via cross-layer index reuse. arXiv preprint arXiv:2603.12201, 2026.

W. Brandon, M. Mishra, A. Nrusimha, R. Panda, and J. Ragan-Kelley. Reducing transformer key-value cache size with cross-layer attention. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, Advances in Neural Information Processing Systems, volume 37, pages 86927–86957. Curran Associates, Inc., 2024. doi: 10.52202/079017-2758. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/9e23d020c18e4c40d81c6a0fc7a46f68-Paper-Conference.pdf.

L. Chen, D. Xu, C. An, X. Wang, Y. Zhang, J. Chen, Z. Liang, F. Wei, J. Liang, Y. Xiao, et al. Powerattention: exponentially scaling of receptive fields for effective sparse attention. arXiv preprint arXiv:2503.03588, 2025.

L. Chen, W. Xie, Y. Liang, H. He, H. Zhao, Z. Yang, Z. Huang, H. Wu, H. Lu, Y. Bao, et al. Babyvision: Visual reasoning beyond language. arXiv preprint arXiv:2601.06521, 2026.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. CoRR, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

X. Cheng, X. Yu, C. Shao, J. Li, Y. Xiong, Y. Qian, J. Zhu, S. Ma, X. Zhang, J. Ye, et al. Dspark: Confidence-scheduled speculative decoding with semi-autoregressive generation. arXiv preprint arXiv:2607.05147, 2026a.

X. Cheng, W. Zeng, D. Dai, Q. Chen, B. Wang, Z. Xie, K. Huang, X. Yu, Z. Hao, Y. Li, H. Zhang, H. Zhang, D. Zhao, and W. Liang. Conditional memory via scalable lookup: A new axis of

sparsity for large language models. CoRR, abs/2601.07372, 2026b. doi: 10.48550/ARXIV.2601.07372. URL https://doi.org/10.48550/arXiv.2601.07372.

X. Cheng, W. Zeng, D. Dai, Q. Chen, B. Wang, Z. Xie, K. Huang, X. Yu, Z. Hao, H. Zhang, et al. Conditional memory via scalable lookup: A new axis of sparsity for large language models. In Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 4968–4990, 2026c.

K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al. Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168, 2021.

D. Dai, C. Deng, C. Zhao, R. X. Xu, H. Gao, D. Chen, J. Li, W. Zeng, X. Yu, Y. Wu, Z. Xie, Y. K. Li, P. Huang, F. Luo, C. Ruan, Z. Sui, and W. Liang. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. CoRR, abs/2401.06066, 2024. URL https://doi.org/10.48550/arXiv.2401.06066.

DataCurve. Deepswe v1.1, 2026. URL https://deepswe.datacurve.ai/.

DeepSeek-AI. Deepseek-v3 technical report. CoRR, abs/2412.19437, 2024. URL https://doi.org/10.48550/arXiv.2412.19437.

DeepSeek-AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. CoRR, abs/2405.04434, 2024. URL https://doi.org/10.48550/arXiv.2405.04434.

DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

DeepSeek-AI. Deepseek harness: Everything is a plugin. https://github.com/deepseek-ai/deepseek-harness, 2026a.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. CoRR, abs/2606.19348, 2026b. URL https://doi.org/10.48550/arXiv.2606.19348.

J. Dekoninck, N. Jovanović, I. Petrov, and M. Vechev. Matharena apex: Unconquered final-answer problems, 2025. URL https://matharena.ai/apex/.

S. Deng, Z. Ouyang, T. Pang, Z. Liu, R. Jin, S. Yu, and Y. Yang. Rmnp: Row-momentum normalized preconditioning for scalable matrix-based optimization. arXiv preprint arXiv:2603.20527, 2026.

J. Ding, S. Long, C. Pu, H. Zhou, H. Gao, X. Gao, C. He, Y. Hou, F. Hu, Z. Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. arXiv preprint arXiv:2512.12730, 2025.

A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In International Conference on Learning Representations, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

X. Du, Y. Yao, K. Ma, B. Wang, T. Zheng, K. Zhu, M. Liu, Y. Liang, X. Jin, Z. Wei, et al. Supergpqa: Scaling llm evaluation across 285 graduate disciplines. arXiv preprint arXiv:2502.14739, 2025.

D. Dua, Y. Wang, P. Dasigi, G. Stanovsky, S. Singh, and M. Gardner. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In J. Burstein, C. Doran, and T. Solorio, editors, Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), pages 2368–2378. Association for Computational Linguistics, 2019. doi: 10.18653/V1/N19-1246. URL https://doi.org/10.18653/v1/n19-1246.

Y. Gao, J. Wei, Q. Zhang, Y. Cheng, S. Chen, Z. Tang, Z. Jiang, Y. Song, H. Zhang, L. Zhao, B. Yang, G. Wang, S. Cao, and F. Luo. Hysparse: A hybrid sparse attention architecture with oracle token selection and kv cache sharing. arXiv preprint arXiv:2602.03560, 2026.

S. Garre, C. Mutty, S. Mehta, and E. Chen. Chartography: A benchmark for professional chart understanding. arXiv preprint arXiv:2608.10677, 2026.

A. Glentis, J. Li, A. Han, and M. Hong. Memory-efficient llm pretraining via minimalist optimizer design. arXiv preprint arXiv:2506.16659, 2025.

Y. Gu, L. Dong, F. Wei, and M. Huang. Minillm: Knowledge distillation of large language models. In The Twelfth International Conference on Learning Representations, 2024.

L. Haas, G. Yona, G. D'Antonio, S. Goldshtein, and D. Das. Simpleqa verified: A reliable factuality benchmark to measure parametric knowledge. arXiv preprint arXiv:2509.07968, 2025.

D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt. Measuring mathematical problem solving with the math dataset. arXiv preprint arXiv:2103.03874, 2021.

Y. Huang, Y. Bai, Z. Zhu, J. Zhang, J. Zhang, T. Su, J. Liu, C. Lv, Y. Zhang, J. Lei, et al. C-Eval: A multi-level multi-discipline chinese evaluation suite for foundation models. arXiv preprint arXiv:2305.08322, 2023.

D. Hupkes and N. Bogoychev. Multiloko: a multilingual local knowledge benchmark for llms spanning 31 languages. CoRR, abs/2504.10356, 2025. doi: 10.48550/ARXIV.2504.10356. URL https://doi.org/10.48550/arXiv.2504.10356.

B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2018.

K. Jordan, Y. Jin, V. Boza, J. You, F. Cesista, L. Newhouse, and J. Bernstein. Muon: An optimizer for hidden layers in neural networks. Cited on, page 10, 2024.

M. Kazemi, B. Fatemi, H. Bansal, J. Palowitch, C. Anastasiou, S. V. Mehta, L. K. Jain, V. Aglietti, D. Jindal, Y. P. Chen, et al. Big-bench extra hard. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 26473–26501, 2025.

S. Kazemzadeh, V. Ordonez, M. Matten, and T. Berg. Referitgame: Referring to objects in photographs of natural scenes. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pages 787–798, 2014.

R. Kondra, S. Mhatre, A. Kumar, E. Chu, B. B. Ahmad, A. Nangia, R. Agarwal, A. Dasgupta, A. Sinha, B. Sridharan, K. Dave, B. Graham, G. Song, A. Rahul, W. H. Lim, A. Thangamuthu, R. Singh, D. Liu, N. Pour, C. Chen, and J. Mattern. Frontierswe v2. Proximal Blog, 2026. https://frontierswe.com/blog/v2.

H. Lee, J. Liu, D. Kim, W. Xia, Z. Zhang, C. S. Xia, and L. Zhang. Sec-bench pro: Can language models solve long-horizon software security tasks? arXiv preprint arXiv:2605.26548, 2026.

J. Li and S. Liu. Flashmla: Efficient multi-head latent attention kernels. `https://github.com/deepseek-ai/FlashMLA`, 2025.

J. Liu, J. Su, X. Yao, Z. Jiang, G. Lai, Y. Du, Y. Qin, W. Xu, E. Lu, J. Yan, Y. Chen, H. Zheng, Y. Liu, S. Liu, B. Yin, W. He, H. Zhu, Y. Wang, J. Wang, M. Dong, Z. Zhang, Y. Kang, H. Zhang, X. Xu, Y. Zhang, Y. Wu, X. Zhou, and Z. Yang. Muon is scalable for LLM training. CoRR, abs/2502.16982, 2025. URL `https://doi.org/10.48550/arXiv.2502.16982`.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In International Conference on Learning Representations, 2019. URL `https://openreview.net/forum?id=Bkg6RiCqY7`.

K. Lu and T. M. Lab. On-policy distillation. Thinking Machines Lab: Connectionism, 2025. doi: 10.64434/tml.20251026. https://thinkingmachines.ai/blog/on-policy-distillation.

J. Mao, J. Huang, A. Toshev, O. Camburu, A. L. Yuille, and K. Murphy. Generation and comprehension of unambiguous object descriptions. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 11–20, 2016.

A. Marafioti, O. Zohar, M. Farré, M. Noyan, E. Bakouch, P. Cuenca, C. Zakka, L. B. Allal, A. Lozhkov, N. Tazi, V. Srivastav, J. Lochner, H. Larcher, M. Morlon, L. Tunstall, L. von Werra, and T. Wolf. Smolvlm: Redefining small and efficient multimodal models. arXiv preprint arXiv:2504.05299, 2025.

R. Marten, A. Shaw, I. Bercovich, B. Droste, T. Cerruti, S. Dillmann, R. Wang, D. Wahdany, A. Hart, K. Krauth, ScaleAI, Snorkel AI, Turing, gNucleus AI, Boolean AI, N. Carlini, S. Lyu, A. Wei, A. Khatua, B. Plüster, C. Dwivedi, C. Sutcliffe, Yuming, D. Tivris, D. Wang, H. W. Goh, H. Xing, H. Lin, I. Salia, J. Seol, J. Bao, J. Ouyang, J. Park, L. Walsh, L. Kong, M. Ivanov, M. Ubl, M. Liamets, O. Menis, P. Migdal, Q. Bao, R. Movva, R. Ben Chaim, N. Srinath, S. Bogdanik, S. Yadav, S. Benjamin, T. Kung, W. Hughes, X. Lan, H. Gupta, S. Mishra, C. Wang, H. He, J. Tu, K. Montgomery, Z. Tu, A. Naik, D. Mortensen, I. Zhang, Y. Mathur, E. Liu, K. Singh, M. Yu, S. Feng, V. Gangal, Z. Tao, S. Ruan, J. Mueller, J. Cabezas, J. Bauer, K. X. Li, R. Zhang, A. Feller, A. Madayan, L. Chen, B. Feuer, X. Li, B. Li, H. Raj, S. Galler, L. Shi, I. Segal, K. Buchanan, S. P., R. Desai, A. Schneider, C. Settles, X. Lin, M. Nezhurina, A. Wang, M. Kowalczyk, J.-X. Zhao, S. Satia, J. Hu, S. Atef, K. Chen, S. Vance, G. Segato, J. Jitsev, A. Dimakis, M. Merrill, A. Konwinski, and L. Schmidt. Terminal-Bench, Aug. 2026a. URL `https://github.com/harbor-framework/terminal-bench`.

R. Marten, A. Shaw, A. Konwinski, and Terminal-Bench Contributors. Terminal-bench 3.0: Harder tasks for better agents. `https://www.tbench.ai/news/terminal-bench-3-0`, 2026b. An Open Benchmark for Agent Work in Terminal Environments.

M. Mathew, D. Karatzas, and C. Jawahar. Docvqa: A dataset for vqa on document images. In 2021 IEEE Winter Conference on Applications of Computer Vision (WACV), pages 2199–2208. IEEE, 2021.

M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. arXiv preprint arXiv:2601.11868, 2026.

V. K. Nagaraja, V. I. Morariu, and L. S. Davis. Modeling context between objects for referring expression understanding. In European conference on computer vision, pages 792–807. Springer, 2016.

Y. Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. Soviet Mathematics Doklady, 27:372–376, 1983.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. CoRR, abs/2508.10925, 2025. doi: 10.48550/ARXIV.2508.10925. URL https://doi.org/10.48550/arXiv.2508.10925.

OpenAI. Codex. https://github.com/openai/codex, 2026.

L. Phan, A. Gatti, Z. Han, N. Li, J. Hu, H. Zhang, C. B. C. Zhang, M. Shaaban, J. Ling, S. Shi, et al. Humanity's last exam. arXiv preprint arXiv:2501.14249, 2025.

Y. Qian, S. Liu, and Y. Li. Deepselect: High-performance topk kernels for deepseek sparse attention and sampling. https://github.com/deepseek-ai/DeepSelect, 2026.

D. Rein, B. L. Hou, A. C. Stickland, J. Petty, R. Y. Pang, J. Dirani, J. Michael, and S. R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. arXiv preprint arXiv:2311.12022, 2023.

J. Roberts, M. R. Taesiri, A. Sharma, A. Gupta, S. Roberts, I. Croitoru, S.-V. Bogolin, J. Tang, F. Langer, V. Raina, et al. Zerobench: An impossible visual benchmark for contemporary large multimodal models. arXiv preprint arXiv:2502.09696, 2025.

B. D. Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf, S. Dusan, V. Elango, M. Golub, A. Heinecke, P. James-Roxby, D. Jani, G. Kolhe, M. Langhammer, A. Li, L. Melnick, M. Mesmakhosroshahi, A. Rodriguez, M. Schulte, R. Shafipour, L. Shao, M. Siu, P. Dubey, P. Micikevicius, M. Naumov, C. Verrilli, R. Wittig, D. Burger, and E. Chung. Microscaling data formats for deep learning, 2023.

M. Scetbon, C. Ma, W. Gong, and E. Meeds. Gradient multi-normalization for efficient LLM training. In The Thirty-ninth Annual Conference on Neural Information Processing Systems, 2025. URL https://openreview.net/forum?id=oanhUGY6un.

N. Shazeer. Glu variants improve transformer. arXiv preprint arXiv:2002.05202, 2020.

N. Shazeer and M. Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In International conference on machine learning, pages 4596–4604. PMLR, 2018.

D. Shepard and R. Salimans. Automationbench. arXiv preprint arXiv:2604.18934, 2026.

F. Shi, M. Suzgun, M. Freitag, X. Wang, S. Srivats, S. Vosoughi, H. W. Chung, Y. Tay, S. Ruder, D. Zhou, D. Das, and J. Wei. Language models are multilingual chain-of-thought reasoners. In The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net, 2023. URL https://openreview.net/forum?id=fR3wGCk-IXp.

Y. Sun, L. Dong, Y. Zhu, S. Huang, W. Wang, S. Ma, Q. Zhang, J. Wang, and F. Wei. You only cache once: Decoder-decoder architectures for language models. Advances in Neural Information Processing Systems, 37:7339–7361, 2024.

Y. Sun, X. Han, W. Zhang, Y. Pang, T. Wang, Y. Cao, Y. Huang, C. Duroiu, H. Zhang, J. Lin, et al. Agents' last exam. arXiv preprint arXiv:2606.05405, 2026a.

Y. Sun, Y. Zhang, L. Dong, J. Wang, and F. Wei. You only index once: Cross-layer sparse attention with shared routing. arXiv preprint arXiv:2606.06467, 2026b.

M. Suzgun, N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. V. Le, E. H. Chi, D. Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. arXiv preprint arXiv:2210.09261, 2022.

K. Team, T. Bai, Y. Bai, Y. Bao, J. Cai, X. Cai, P. Cao, Y. Cao, Z. Chai, Y. Charles, et al. Kimi k3: Open frontier intelligence. arXiv preprint arXiv:2607.24653, 2026a.

K. Team, T. Bai, Y. Bai, Y. Bao, S. Cai, Y. Cao, Z. Chai, Y. Charles, H. Che, C. Chen, et al. Kimi k2.5: Visual agentic intelligence. arXiv preprint arXiv:2602.02276, 2026b.

M. L. Team, B. Wang, B. Xiao, B. Zhang, B. Rong, B. Chen, C. Wan, C. Zhang, C. Huang, C. Chen, et al. Longcat-flash-omni technical report. arXiv preprint arXiv:2511.00279, 2025.

S. Tong, E. Brown, P. Wu, S. Woo, M. Middepogu, S. C. Akula, J. Yang, S. Yang, A. Iyer, X. Pan, A. Wang, R. Fergus, Y. LeCun, and S. Xie. Cambrian-1: A fully open, vision-centric exploration of multimodal llms, 2024.

L. Wang, H. Gao, C. Zhao, X. Sun, and D. Dai. Auxiliary-loss-free load balancing strategy for mixture-of-experts. CoRR, abs/2408.15664, 2024a. URL `https://doi.org/10.48550/arXiv.2408.15664`.

X. Wang, C. Xu, H. Cao, R. Tian, W. Zhao, K. Yu, and C. Zhao. Tilekernels. `https://github.com/deepseek-ai/TileKernels`, 2026a.

Y. Wang, X. Ma, G. Zhang, Y. Ni, A. Chandra, S. Guo, W. Ren, A. Arulraj, X. He, Z. Jiang, T. Li, M. Ku, K. Wang, A. Zhuang, R. Fan, X. Yue, and W. Chen. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. CoRR, abs/2406.01574, 2024b. URL `https://doi.org/10.48550/arXiv.2406.01574`.

Z. Wang, N. Schiller, H. Li, S. S. Narayana, M. Nasr, N. Carlini, X. Qi, E. Wallace, E. Bursztein, L. Invernizzi, et al. Exploitgym: Can ai agents turn security vulnerabilities into real attacks? arXiv preprint arXiv:2605.11086, 2026b.

Z. Wang, T. Shi, J. He, M. Cai, J. Zhang, and D. Song. Cybergym: Evaluating ai agents' real-world cybersecurity capabilities at scale. In International Conference on Learning Representations, volume 2026, pages 123341–123386, 2026c.

Z. Wen, Y. Shi, J. Wang, P. Luo, L. Qiao, D. Li, and T. Sun. Sron: State-free llm training via row-wise gradient normalization. 2025.

Z. Xie, Y. Wei, H. Cao, C. Zhao, C. Deng, J. Li, D. Dai, H. Gao, J. Chang, K. Yu, L. Zhao, S. Zhou, Z. Xu, Z. Zhang, W. Zeng, S. Hu, Y. Wang, J. Yuan, L. Wang, and W. Liang. mhc: Manifold-constrained hyper-connections, 2026. URL `https://arxiv.org/abs/2512.24880`.

R. Xu, J. Li, and Y. Lu. On the width scaling of neural optimizers under matrix operator norms i: Row/column normalization and hyperparameter transfer. arXiv preprint arXiv:2603.09952, 2026a.

Y. Xu, F. Meng, F. Jiang, Y. Wang, R. Zhou, Z. Wang, J. Wu, Z. Pan, X. Tang, W. Pei, et al. Hisa: Efficient hierarchical indexing for fine-grained sparse attention. arXiv preprint arXiv:2603.28458, 2026b.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In The Thirty-eighth Annual Conference on Neural Information Processing Systems, 2024. URL https://arxiv.org/abs/2405.15793.

J. Yang, K. Lieret, J. Ma, P. Thakkar, D. Pedchenko, S. Sootla, E. McMilin, P. Yin, R. Hou, G. Synnaeve, D. Yang, and O. Press. Programbench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

L. Yu, P. Poirson, S. Yang, A. C. Berg, and T. L. Berg. Modeling context in referring expressions. In European conference on computer vision, pages 69–85. Springer, 2016.

J. Yuan, J. Zou, S. Wang, Y. Liu, and F. Nie. Nora: Normalized orthogonal row alignment for scalable matrix optimizer. arXiv preprint arXiv:2605.03769, 2026.

X. Yue, T. Zheng, Y. Ni, Y. Wang, K. Zhang, S. Tong, Y. Sun, B. Yu, G. Zhang, H. Sun, et al. Mmmu-pro: A more robust multi-discipline multimodal understanding benchmark. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15134–15186, 2025.

M. Zechner. Pi: The coding-agent harness you can make your own. https://github.com/earendil-works/pi, 2026.

R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. HellaSwag: Can a machine really finish your sentence? In A. Korhonen, D. R. Traum, and L. Màrquez, editors, Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers, pages 4791–4800. Association for Computational Linguistics, 2019. doi: 10.18653/v1/p19-1472. URL https://doi.org/10.18653/v1/p19-1472.

A. Zeng, X. Lv, Z. Hou, Z. Du, Q. Zheng, B. Chen, D. Yin, C. Ge, C. Huang, C. Xie, et al. Glm-5: from vibe coding to agentic engineering. arXiv preprint arXiv:2602.15763, 2026.

X. Zhai, B. Mustafa, A. Kolesnikov, and L. Beyer. Sigmoid loss for language image pre-training. In 2023 IEEE/CVF International Conference on Computer Vision (ICCV), pages 11941–11952. IEEE, 2023.

B. Zhang and R. Sennrich. Root mean square layer normalization. Advances in neural information processing systems, 32, 2019.

Y. Zhang. On the principles behind neural network optimizers. arXiv preprint arXiv:2608.16760, 2026.

Y. Zhang, C. Chen, T. Ding, Z. Li, R. Sun, and Z.-Q. Luo. Why transformers need adam: A hessian perspective. Advances in neural information processing systems, 37:131786–131823, 2024.

Y. Zhang, C. Chen, Z. Li, T. Ding, C. Wu, D. D. Kingma, Y. Ye, Z.-Q. Luo, and R. Sun. Adam-mini: Use fewer learning rates to gain more. In International Conference on Learning Representations, volume 2025, pages 28033–28063, 2025a.

Z. Zhang, Y. Zhong, Y. Jiang, H. Hu, J. Sun, Z. Ge, Y. Zhu, D. Jiang, and X. Jin. Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models. In Proceedings of the ACM SIGCOMM 2025 Conference, pages 24–38, 2025b.

C. Zhao, L. Zhao, J. Li, Z. Xu, and C. Xu. Deepgemm: clean and efficient fp8 gemm kernels with fine-grained scaling. https://github.com/deepseek-ai/DeepGEMM, 2025.

W. Zhong, R. Cui, Y. Guo, Y. Liang, S. Lu, Y. Wang, A. Saied, W. Chen, and N. Duan. AGIEval: A human-centric benchmark for evaluating foundation models. CoRR, abs/2304.06364, 2023. doi: 10.48550/arXiv.2304.06364. URL https://doi.org/10.48550/arXiv.2304.06364.

T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, J. Hoang, A. R. Zebaze, X. Hong, W. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, and et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025. OpenReview.net, 2025. URL https://openreview.net/forum?id=YrycTjllL0.

Appendix

A. Author List

Authors are listed alphabetically by their first name. Names marked with * denote individuals who have departed from our team.

Research & Engineering: Anyi Xu, B. Li, Bangcai Lin, Bing Xue, BingCheng Xian, Bingzheng Xu, Bochao Wu, Bowei Zhang, Boyi Deng, C.C. Yu, Chao Jin, Chaofan Lin, Chen Dong, Chenbing Wang, Chenfan Feng, Chengda Lu*, Chenggang Zhao, Chengqi Deng, Chengyuan Zhang, Chenhao Xu, Chenqi Zhao, Chenze Shao, Chuhao Wang, Chuqi Zhang, Damai Dai, Dejian Yang, Deli Chen, Di Huang, Di Wu, Donghao Li, Erhang Li, Eric Fu, F. Zhou, Fangwei Zhou, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanglin Li, Guanting Chen*, Guoai Cao, Guofan Fan, Guolai Meng, Guowei Li, Haichuan Zhang, Haiyang Ma, Haiyang Shen, Han Li, Han Yu, Han Zhang, Hangyuan Deng, Hanwei Xu, Hanxiang Xu, Hanxun Zhong, Hao Guo, Hao Jiang, Hao Li, Hao Qin, Haodong Wen, Haofen Liang, Haofeng Huang, Haohua Liu, Haoling Zhang, Haoming Luo, Haoran Yang, Haotian Xu*, Haotian Yuan, Haoting Huang, Haowen Luo, Haoyang Cai, Haoyu Chen, Haozhe Ji, Hengran Zhang, Hengrui Wang, Hengxu Wu, Honghui Ding, Hongxuan Tang, Huadong Wang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J. Yang, J.H. Jin, J.H. Zhang, J.X. Zou, Jia Yu, Jiahui Zhou, Jiajun Chen, Jialiang Huang, Jialin Zhao, Jiamin Tang, Jian Zhou, Jianan Tong, Jianwen Li, Jiaqi Zhu, Jiarui Wang, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiaying Ding, Jibai Lu, Jiewen Hu, Jin Yan, Jincheng Zhai, Jingchang Chen, Jingcheng Hu, Jingli Zhou, Jingsheng Xu, Jingting Xiang, Jingyan Yun, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jinpeng Wang, Jinyi Chen, Jinyi Hu, Jiping Yu, Jueliang Guo, Junbo Pei, Junbo Sun, Junguang Jiang, Junjie Qiu, Junkang Zhou, Junqi Liu, Junren Li, Junxian Li, Junxiao Song, Junyi Guo, Kai Dong, Kaifeng Chen, Kaige Gao, Kang Guan, Kangdong Yuan, Ke Hong, Ke Xu, Kefan Zhao, Kexin Ji, Kexin Zhang, Kexing Zhou, Kuai Yu, Lan Zhang, Lean Wang, Lecong Zhang, Lei Wang, Letian Gao, Liang Zhao, Liansheng Xu, Lihua Guo, Lingxiao Luo, Lingyue Fu, Litao Deng, Litong Wang, Liyue Zhang, Longhao Chen, Lu Chen, Luotian Huang, Luyao Ma, Luyao Wang, M.S. Di, Max Mei, Menghao Ye, Miao Cui, Mingchuan Zhang, Minghua Zhang*, Minghui Tang, Mingjing Zhang, Mingqi Wei, Mingshu Chen, Mingxing Liu, Mingxu Zhou, Mingyu Xu, Mingyu Yang, Mingze Wang, Muyang Chen, Ni Shentu, Ning Wang, Niufang Ning, Panpan Huang, Peixin Cong, Peiyi Wang, Peiyuan Xin, Pengfei Ren, Pengfei Yan, Pengle Zhang, Qi Kang, Qi Tang, Qiancheng Wang, Qiang Li, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qizhou Guo, Rongxian Xu, Rui Ding, Rui Hu, Rui Tian, Rui Yu, Ruidong Zhu, Ruifan Xu, Ruihan Yang, Ruihang Xia, Ruijie Lu, Ruilin Geng, Ruipeng Hong, Ruiqi Ge, Ruisong Zhang, Ruize Sun, Ruizhe Pan, Runji Wang, Runqian Chen, Runxin Xu, Ruohong Tian, Ruomeng Shen, Ruoyu Zhang, Ryan X., S.H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoyuan Chen, Shengding Hu, Shengkai Lin, Shengwen Ran, Shengyu Liu, Shengyuan Jia, Shi Bai, Shi Feng, Shicheng Xu, Shichun Liu, Shiqiang Hu*, Shirong Ma, Shiyu Wang, Shiyuan Feng, Shufan Gong, Shuhan Lin, Shuiping Yu, Shunfeng Zhou, Shuo Yang, Shuomeng Wang, Shuting Guo, Shuting Pan, Shuying Yu, Sinuo Cao, Siyi Lin, Sizhe Chen, Songyang Chen, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji*, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tongrui Xiong, Wangding Zeng, Wei Liu, Wei Zhang, Weibin Xu, Weihao Zeng, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao*, Wenjun Gao, Wenkai Shao, Wenkai Yang, Wenli Zhang, Wenlu Wang, Wenlve Huang, Wenqian Yan, Wentao Zhang, Xi Gao, Xiang He, Xiang Li, Xiangli Li, Xiangwen Wang, Xiangying Zhang, Xiankui Wei, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojian Qu, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaoyao Zou, Xiaoyuan Li, Xicheng Guo, Xieting Chu,

Xin Cheng, Xin Liu, Xin Xie, Xinbo Xu, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xintong Yao, Xinyang Chen, Xinyong Jiang, Xinyu Yang, Xinyu Yang, Xu Chen, Xuanyu Wang, Xubei Zhong, Xuecheng Su, Xuejie Liu, Xuheng Lin, Xujie Fan, Xuncheng Zhao, Xuwei Fu, Y.C. Yan, Y.H. Jiang, Y.T. Wu*, Y.W. M., Y.Z. Wang, Yafei Gao, Yang Yang, Yang Zhang, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Meng, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yaoyang Ye, Yehang Yin, Yexinrui Wu, Yi Qian, Yi Tao, Yi Yu, Yichao Zhang, Yichen Jiang, Yicheng Wang, Yifan Ding, Yifan Shi, Yifeng Peng, Yifeng Zhai, Yijia Wu, Yiliang Xiong, Yilun Wang, Ying He, Ying Zhou*, Yingjia Luo, Yinmin Zhong, Yiping Wang, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyao Yang, Yiyuan Liu, Yizai Cai, Yizhen Wei, Yizhi Wang, Yonglun Yang, Yongqi Zhuo, Yongqiang Guo, Yongtong Wu, Yu Wu, Yu Zhang, Yuan Bian, Yuan Cheng, Yuan Ou, Yuan Sun, Yuanfan Xu, Yuanhang Sun, Yuanhao Li, Yuchen Liu, Yuchen Yao, Yudong Han, Yuduan Wang, Yuhan Wu, Yuhao Meng, Yuheng Zou, YuKun Li, Yunchuan Wang, Yunfan Xiao, Yunfan Xiong, Yupeng Chen, Yuqian Cao, Yuqian Wang, Yuqing Chen, Yushun Zhang, Yutong Lin, Yuwei Xiao, Yuxian Gu, Yuxiang Chen, Yuxiang Huang, Yuxiang Luo, Yuxiang You, Yuxin Chen, Yuxin Xiang, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhe Guo, Yuzhen Huang, Yuzhuo Bai, Z.Y. Z., Zanlin Ni, Zehao Wang, Zehua Zhao, Zehui Ren, Zejun Zhao, Zhangli Sha, Zhanying Wang, Zhaochen Zhang, Zhaoshuai Du, Zhe Fu, Zhean Xu, Zhenda Xie, Zheng Liu, Zhengyan Zhang, Zhenhua Dong, Zhewen Hao, Zhibang Wang, Zhibin Gou, Zhicheng Ma, Zhihao Li, Zhihong Shao, Zhihuan Huang, Zhijie Li, Zhirui Lu, Zhixian Huang, Zhixuan Chen, Zhixuan Chen, Zhixuan Pan, Zhiyu Wu, Zhizhou Ren, Zhu He, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zili Zhang, Zilin Li, Zilong Hou, Zilong Lyu, Ziqiao Wang, Ziwei Xie, Ziya Zhang, Ziyi Gao, Zizheng Pan, Zonglin Li, Zongqing Yao, Zui Chen, Zuofan Wu

Business & Compliance: Chenchen Ling, Chengyu Hou, Chong Chen, D. Li, Di Qi, Dongjie Ji, Fang Wei, Fanyi Xia, Fei Xie, Feiyi Tan, Hailong Guo, Haiyan Zhai, Hui Zhou, Huihui Tan, Huijie Li, Jia Luo, Jia Song, Jialu Cai, Jian Liang, Jiangting Zhou, Jiaqi Gao, Jiayi Shao, Jie Chen, Jieyu Yang, Jin Chen, Jingde Zhang, Jingzi Zhou, Jinqian Wang, Jinyang Liu, JinZhao Sun, Junhua Ling, Junmin Zheng, Kaicheng Yang, Ke Xu, Le Su, Leyi Xia, Liangfeng Ding, Lin Zhuo, Linwang Ma, Linyan Zhu, Liyu Cai, Luqi Yao, M.K. Zhang, Meng Li, Miao Lin, Miaojun Wang, Min Zhang, Mingming Li, Mingming Wang, Mingze Yin, Minmin Han, Nan Cao, Ning Wang, Ningxin Ma, Panpan Wang, Peihan Lin, Peng Sun, Peng Zhang, Qian Ying, Qiang Xiang, Qiao Wang, Qingmiao Mao, Qiwei Jiang, Rongli Jin, Ruyi Chen, Sha Tao, Shangmian Sun, Shaoqing Wu, Shichao Zou, Si Lei, Tianyang Zhang, Tianyu Sun, Tingting Yin, W.L. Xiao, Wei An, Wei Li, Wei Wang, Weiwei Lin, Wenqing Hou, X. Lin, Xiangfei Meng, Xianzhu Huang, Xiao Peng, Xiaoqian Li, Xiaoting Zhang, Xiaowen Sun, Xiaoxiang Wang, Xiaoyu Ye, Xinrou Zhang, Xinyu Zhang, Xue Cao, Xueyin Chen, Yanan Zhou, Yanhong Xu, Yao Xia, Yao Xu, Yi Shao, Yihong Zhang, Yiling Ma, Ying Tang, Yining Lou, Yiru Chen, Yishi Piao, Yixuan Chen, Yong Xiong, Yuchen Xuan, Yuehan Yang, Yuer Xu, Yukun Zha, Yunxian Ma, Yuping Lin, Yuting Yan, Yutong Xie, Yuwen Sheng, Yuxuan Zhu, Zekai Zhang, Zhe Ju, Zhenzhen Lin, Zheren Gao, Zheyang Sun, Zhigang Yan, Zhongyu Wu, Zi Wang, Zihua Qu, Ziling Yan, Ziyi Wan

B. Evaluation Details

B.1. Scaffold Configurations

All scaffolds run in Linux task containers using the shared evaluation settings in Table 4. Each run starts from the benchmark task description and uses the prompts, task templates, and tool definitions supplied by the scaffold's runtime or evaluation integration. We add no experimental system prompt.

Table 5 | Performance across Claude Code versions at Max reasoning effort.

Benchmark (Metric)	**v2.1.105**	**v2.1.238**	**v2.1.251**	**v2.1.259**	**Average**
DeepSWE v1.1 (Resolved)	68.4	68.7	69.8	68.6	68.9
Terminal-Bench v2.1 (Pass@1)	87.3	88.4	88.0	87.6	87.8

Note. Average is computed from the unrounded Pass@1 scores of the four versions.

- **Claude Code (v2.1.105, v2.1.238, v2.1.251, v2.1.259).** We use the Claude Agent SDK with each version's native tool interface. Table 4 reports v2.1.251; Table 5 compares all four versions.
- **Codex (v0.147.0).** We use standard app-server mode with adapted tool schemas.
- **OpenCode (v1.18.15).** We use the `build` agent with shell and file tools, and native task delegation.
- **Pi (v0.84.2).** We use RPC mode with file and shell tools plus a search extension exposing `search`, `open_page`, and `find_in_page`.
- **mini-SWE.** We use the `mini_swe_v2` port[1] with a single `bash` tool, requiring a tool call each turn and a submission marker to finish.
- **DeepSeek Harness (DSH).** We use Minimal with a single `bash` tool; Standard with the full `sdk` profile and 26 initial function tools, including web search and fetch; and PTC with `run_code` for TypeScript programs using 24 underlying tools. Standard and PTC use v0.1.1+ custom.202609011522.

B.2. Reasoning Efforts across Scaffolds

Across all six panels in Figure 11, raising the reasoning-effort setting lengthens the trajectories: mean output tokens per trajectory grow monotonically with effort in every scaffold–benchmark pair. Pass@1 tracks this growth only loosely. It improves overall, but the response is not monotone, with plateaus and dips at intermediate settings in most panels. The three scaffolds are also calibrated differently: on DeepSWE v1.1, Claude Code is the flattest curve and spends comparatively few extra tokens, whereas DeepSeek Harness (Minimal) starts lowest and gains the most at the largest token cost, with mini-SWE in between. On Terminal-Bench v2.1 the three scaffolds are compressed into a narrow band, and the ordering at maximum effort favours DeepSeek Harness (Minimal), followed by mini-SWE and Claude Code: scaffold choice matters at least as much as the effort tier once the task is nearly saturated.

B.3. Detailed Results of Reasoning Benchmarks across Reasoning Efforts

Figure 12 shows that the reasoning-effort setting gives smooth, well-behaved control over both output length and accuracy, consistently across all eight benchmarks, which span competition mathematics, science QA, open-domain knowledge and code. On the length side, raising the effort from 25 to 100 scales the average response predictably on every benchmark – a uniform 2.0–3.1× increase, from 4.6k to 11.4k tokens per response on AIME 2026 and from 29.1k to 86.1k on MathArena Apex 2025 – with no runaway growth or anomalies, so the compute cost of any tier can be estimated in advance. Accuracy follows the same smooth trajectory and responds in the right direction on every benchmark, with no benchmark ever degrading as the effort increases: MathArena Apex 2025 gains +40.3 points (25.3%→65.6%) and Apex 2025 Shortlist +11.5, while even the already-saturated benchmarks remain stable (GPQA Diamond

[1] mini-swe-agent, commit `04d809ceab9d`.

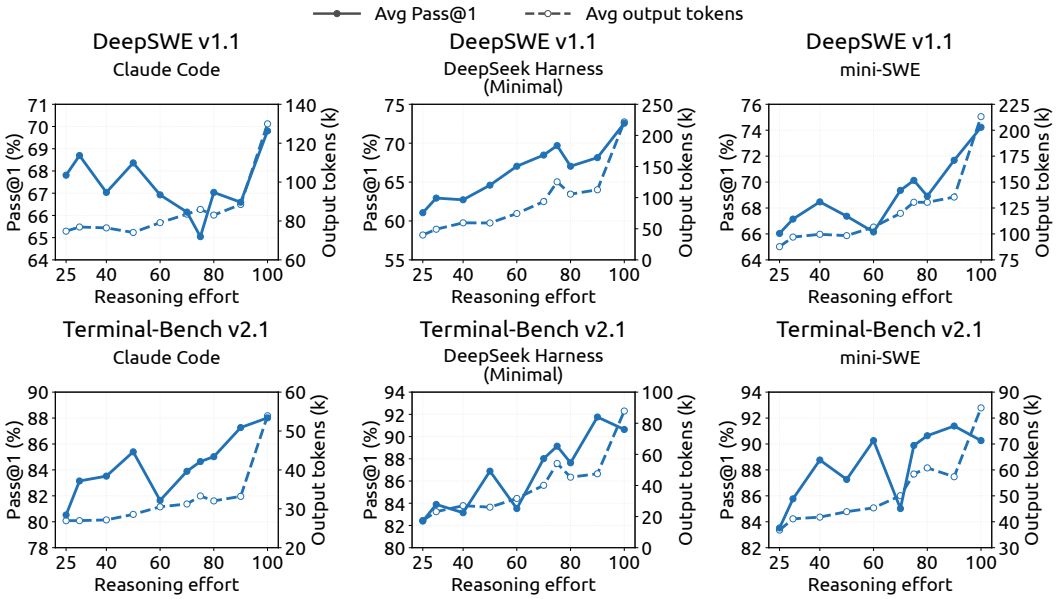

Figure 11 | Reasoning effort drives trajectory length consistently but correlates only weakly with accuracy across coding scaffolds. Each panel plots Pass@1 (%, solid, left axis) and mean output tokens per trajectory (k, dashed, right axis) against the reasoning-effort setting, for DeepSWE v1.1 (top row) and Terminal-Bench v2.1 (bottom row) under three agent scaffolds: Claude Code, DeepSeek Harness (Minimal) and mini-SWE. All panels come from the same checkpoint.

+1.3, LiveCodeBench +2.6), and AIME 2026 reaches a full 100%. The model thus exposes a single, reliable knob that moves the cost–accuracy operating point in a controlled and predictable way, allowing each deployment to select the effort tier that matches its latency and compute budget without sacrificing accuracy.

C. Exponential Token Penalty in Reasoning Effort Control

The scalar effort variable provides a deployment-time control over the cost–quality trade-off without imposing a hard token budget. During reinforcement learning, lower effort levels apply a stronger token penalty, whereas higher effort levels permit more computation. This section gives a simplified motivation for the exponential penalty schedule.

For a trajectory with ℓ reasoning tokens generated at effort level b, the length deduction is

$$r^{\text{len}}(\ell, b) = -\min\left\{C_{\max}, k(b)\frac{\ell}{L_{\text{norm}}}\right\}, \tag{11}$$

where L_{norm} is a reference length and $C_{\max}$ caps the deduction. The effort-dependent token-penalty coefficient is

$$k(b) = k_0 \exp\left(-\frac{b - b_{\min}}{\tau}\right), \tag{12}$$

where k_0 is the penalty coefficient at the lowest effort level $b_{\min}$ and τ controls the rate of penalty decay.

To motivate this choice, consider a fixed problem x. Let $p_x(\ell)$ denote its probability of being solved after ℓ reasoning tokens. In the uncapped region, define the preferred reasoning length

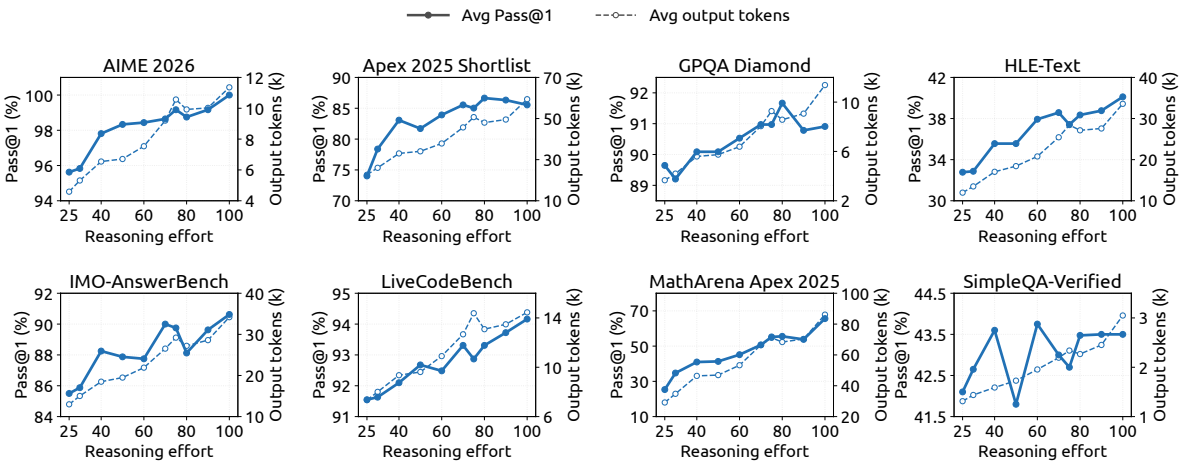

Figure 12 | Performance and output length as a function of reasoning effort on eight reasoning-intensive benchmarks. Each panel plots Pass@1 (solid, left axis) and mean output tokens per response (dashed, right axis) as the reasoning-effort value is varied from 25 to 100.

$\ell_x^*(b)$ by

$$\ell_x^*(b) \in \arg\max_{\ell \geqslant 0} \left[p_x(\ell) - k(b) \frac{\ell}{L_{\text{norm}}} \right]. \tag{13}$$

For an interior optimum, the first-order condition is

$$p_x'\left(\ell_x^*(b)\right) = \frac{k(b)}{L_{\text{norm}}}, \tag{14}$$

where $p_x'(\ell) = dp_x(\ell)/d\ell$ is the marginal improvement in solve probability from additional reasoning.

We assume that this marginal benefit decays approximately exponentially over the relevant operating range:

$$p_x'(\ell) \approx a_x \exp\left(-\frac{\ell}{s_x}\right), \tag{15}$$

where $a_x > 0$ is an instance-dependent scale and $s_x > 0$ determines the decay rate. Substituting Eqs. (12) and (15) into Eq. (14) gives

$$\ell_x^*(b) \approx C_x - s_x \log k_0 + \frac{s_x}{\tau}(b - b_{\min}), \tag{16}$$

where $C_x = s_x \log(a_x L_{\text{norm}})$ is independent of b. Thus, the exponential penalty schedule produces a simple first-order affine trend between the requested effort and the preferred reasoning length under this local model.

For two effort levels $b_2 > b_1$, the corresponding predicted length difference is

$$\ell_x^*(b_2) - \ell_x^*(b_1) \approx \frac{s_x}{\tau}(b_2 - b_1). \tag{17}$$

Consequently, k_0 mainly controls the overall pressure toward shorter reasoning, while τ controls the predicted sensitivity to effort.

This derivation is a local reward-level approximation, not a claim that measured average lengths must be linear or pointwise monotonic. Realized behavior may deviate because the

effort instruction can directly change the reasoning strategy, generation is stochastic, agent trajectories contain different numbers of turns, and subgroup reward normalization changes optimization strength. The analysis also assumes an interior solution for which the penalty cap is inactive. Once the cap is reached, the marginal token penalty becomes zero and the capped region must be considered separately.