Activation CPU offload All remaining activations are offloaded to CPU RAM. A copy engine is responsible for streaming the offload and onload, overlapping with both computation and communication kernels. During the 1F1B phase, we offload the forward activations of the previous micro-batch while prefetching the backward activations of the next. The warm-up and cool-down phases are handled similarly and the overall pattern is shown in Figure 7. Although offloading may slightly affect EP traffic due to PCIe traffic congestion, our tests show that EP communication remains fully overlapped.

2.5 Training recipe

We pre-trained the model with a 4,096-token context window using the MuonClip optimizer (Algorithm 1) and the WSD learning rate schedule [25], processing a total of 15.5T tokens. The first 10T tokens were trained with a constant learning rate of 2e-4 after a 500-step warm-up, followed by 5.5T tokens with a cosine decay from 2e-4 to 2e-5. Weight decay was set to 0.1 throughout, and the global batch size was held at 67M tokens. The overall training curve is shown in Figure 3.

Towards the end of pre-training, we conducted an annealing phase followed by a long-context activation stage. The batch size was kept constant at 67M tokens, while the learning rate was decayed from 2e-5 to 7e-6. In this phase, the model was trained on 400 billion tokens with a 4k sequence length, followed by an additional 60 billion tokens with a 32k sequence length. To extend the context window to 128k, we employed the YaRN method [55].

3 Post-Training

3.1 Supervised Fine-Tuning

We employ the Muon optimizer [33] in our post-training and recommend its use for fine-tuning with K2. This follows from the conclusion of our previous work [46] that a Muon-pre-trained checkpoint produces the best performance with Muon fine-tuning.

We construct a large-scale instruction-tuning dataset spanning diverse domains, guided by two core principles: maximizing prompt diversity and ensuring high response quality. To this end, we develop a suite of data generation pipelines tailored to different task domains, each utilizing a combination of human annotation, prompt engineering, and verification processes. We adopt K1.5 [35] and other in-house domain-specialized expert models to generate candidate responses for various tasks, followed by LLMs or human-based judges to perform automated quality evaluation and filtering. For agentic data, we create a data synthesis pipeline to teach models tool-use capabilities through multi-step, interactive reasoning.

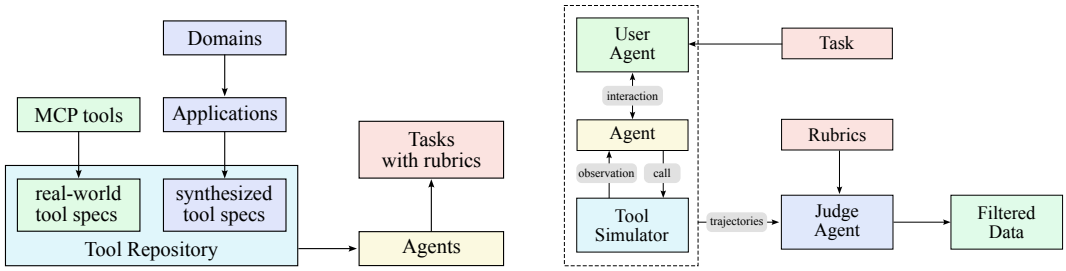
3.1.1 Large-Scale Agentic Data Synthesis for Tool Use Learning

A critical capability of modern LLM agents is their ability to autonomously use unfamiliar tools, interact with external environments, and iteratively refine their actions through reasoning, execution, and error correction. Agentic tool use capability is essential for solving complex, multi-step tasks that require dynamic interaction with real-world systems. Recent benchmarks such as ACEBench [6] and τ-bench [85] have highlighted the importance of comprehensive tool-use evaluation, while frameworks like ToolLLM [58] and ACEBench [6] have demonstrated the potential of teaching models to use thousands of tools effectively.

However, training such capabilities at scale presents a significant challenge: while real-world environments provide rich and authentic interaction signals, they are often difficult to construct at scale due to cost, complexity, privacy and accessibility constraints. Recent work on synthetic data generation (AgentInstruct [51]; Self-Instruct [75]; StableToolBench [20]; ZeroSearch [66]) has shown promising results in creating large-scale data without relying on real-world interactions. Building on these advances and inspired by ACEBench [6]'s comprehensive data synthesis framework, we developed a pipeline that simulates real-world tool-use scenarios at scale, enabling the generation of tens of thousands of diverse and high-quality training examples.

There are three stages in our data synthesis pipeline, depicted in Fig. 8.

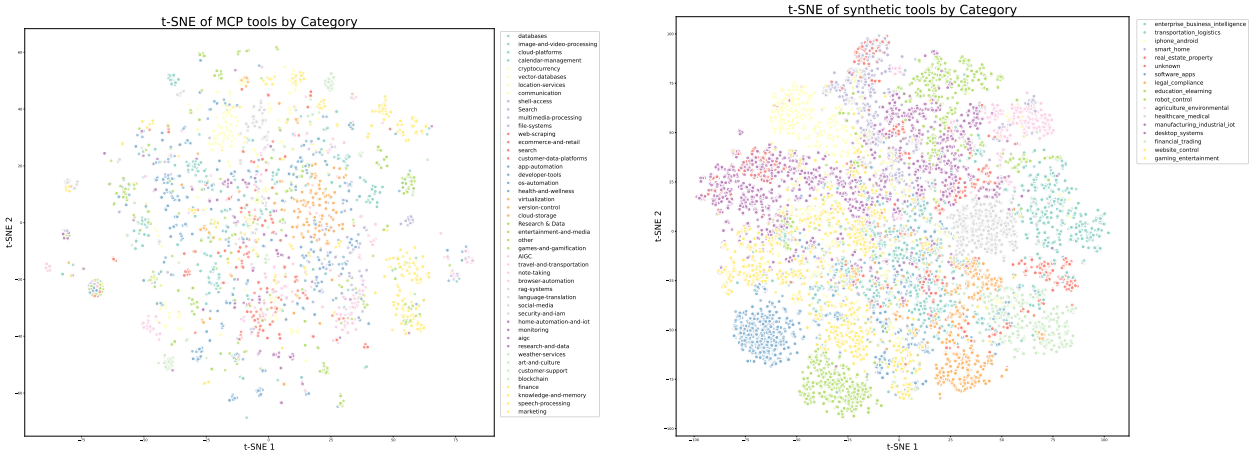
- *Tool spec generation*: we first construct a large repository of tool specs from both real-world tools and LLM-synthetic tools;
- *Agent and task generation*: for each tool-set sampled from the tool repository, we generate an agent to use the toolset and some corresponding tasks;
- *Trajectory generation*: for each agent and task, we generate trajectories where the agent finishes the task by invoking tools.



(a) Synthesizing tool specs, agents and tasks

(b) Generating agent trajectories

Figure 8: Data synthesis pipeline for tool use. (a) Tool specs are from both real-world tools and LLMs; agents and tasks are the generated from the tool repo. (b) Multi-agent pipeline to generate and filter trajectories with tool calling.



(a) t-SNE visualization of real MCP tools, colored by their original source categories

(b) t-SNE visualization of synthetic tools, colored by pre-defined domain categories

Figure 9: t-SNE visualizations of tool embeddings. (a) Real-world MCP tools exhibit natural clustering based on their original source categories. (b) Synthetic tools are organized into pre-defined domain categories, providing systematic coverage of the tool space. Together, they ensure comprehensive representation across different tool functionalities.

Domain Evolution and Tool Generation. We construct a comprehensive tool repository through two complementary approaches. First, we directly fetch 3000+ real MCP (Model Context Protocol) tools from GitHub repositories, leveraging existing high-quality tool specs. Second, we systematically evolve [82] synthetic tools through a hierarchical domain generation process: we begin with key categories (e.g., financial trading, software applications, robot control), then evolve multiple specific application domains within each category. Specialized tools are then synthesized for each domain, with clear interfaces, descriptions, and operational semantics. This evolution process produces over 20,000 synthetic tools. Figure 9 visualizes the diversity of our tool collection through t-SNE embeddings, demonstrating that both MCP and synthetic tools cover complementary regions of the tool space.

Agent Diversification. We generate thousands of distinct agents by synthesizing various system prompts and equipping them with different combinations of tools from our repository. This creates a diverse population of agents with varied capabilities, areas of expertise, and behavioral patterns, ensuring a broad coverage of potential use cases.

Rubric-Based Task Generation. For each agent configuration, we generate tasks that range from simple to complex operations. Each task is paired with an explicit rubric that specifies success criteria, expected tool-use patterns, and evaluation checkpoints. This rubric-based approach ensures a consistent and objective evaluation of agent performance.

Multi-turn Trajectory Generation. We simulate realistic tool-use scenarios through several components:

- *User Simulation*: LLM-generated user personas with distinct communication styles and preferences engage in multi-turn dialogues with agents, creating naturalistic interaction patterns.

- *Tool Execution Environment*: A sophisticated tool simulator (functionally equivalent to a world model) executes tool calls and provides realistic feedback. The simulator maintains and updates state after each tool execution, enabling complex multi-step interactions with persistent effects. It introduces controlled stochasticity to produce varied outcomes including successes, partial failures, and edge cases.

Quality Evaluation and Filtering. An LLM-based judge evaluates each trajectory against the task rubrics. Only trajectories that meet the success criteria are retained for training, ensuring high-quality data while allowing natural variation in task-completion strategies.

Hybrid Approach with Real Execution Environments. While simulation provides scalability, we acknowledge the inherent limitation of simulation fidelity. To address this, we complement our simulated environments with real execution sandboxes for scenarios where authenticity is crucial, particularly in coding and software engineering tasks. These real sandboxes execute actual code, interact with genuine development environments, and provide ground-truth feedback through objective metrics such as test suite pass rates. This combination ensures that our models learn from both the diversity of simulated scenarios and the authenticity of real executions, significantly strengthening practical agent capabilities.

By leveraging this hybrid pipeline that combines scalable simulation with targeted real-world execution, we generate diverse, high-quality tool-use demonstrations that balance coverage and authenticity. The scale and automation of our synthetic data generation, coupled with the grounding provided by real execution environments, effectively implements large-scale rejection sampling [26, 87] through our quality filtering process. This high-quality synthetic data, when used for supervised fine-tuning, has demonstrated significant improvements in the model's tool-use capabilities across a wide range of real-world applications.

3.2 Reinforcement Learning

Reinforcement learning (RL) is believed to have better token efficiency and generalization than SFT. Based on the work of K1.5 [35], we continue to scale RL in both task diversity and training FLOPs in K2. To support this, we develop a Gym-like extensible framework that facilitates RL across a wide range of scenarios. We extend the framework with a large number of tasks with verifiable rewards. For tasks that rely on subjective preferences, such as creative writing and open-ended question answering, we introduce a self-critic reward in which the model performs pairwise comparisons to judge its own outputs. This approach allows tasks from various domains to all benefit from the RL paradigm.

3.2.1 Verifiable Rewards Gym

Math, STEM and Logical Tasks For math, stem and logical reasoning domains, our RL data preparation follows two key principles, *diverse coverage* and *moderate difficulty*.

Diverse Coverage. For math and stem tasks, we collect high-quality QA pairs using a combination of expert annotations, internal QA extraction pipelines, and open datasets [41, 52]. During the collection process, we leverage a tagging system to deliberately increase coverage of under-covered domains. For logical tasks, our dataset comprises a variety of formats, including structured data tasks (e.g., multi-hop tabular reasoning, cross-table aggregation) and logic puzzles (e.g., the 24-game, Sudoku, riddles, cryptarithms, and Morse-code decoding).

Moderate Difficulty. The RL prompt-set should be neither too easy nor too hard, both of which may produce little signal and reduce learning efficiency. We assess the difficulty of each problem using the SFT model's pass@k accuracy and select only problems with moderate difficulty.

Complex Instruction Following Effective instruction following requires not only understanding explicit constraints but also navigating implicit requirements, handling edge cases, and maintaining consistency over extended dialogues. We address these challenges through a hybrid verification framework that combines automated verification with adversarial detection, coupled with a scalable curriculum generation pipeline. Our approach employs a dual-path system to ensure both precision and robustness:

Hybrid Rule Verification. We implement two verification mechanisms: (1) deterministic evaluation via code interpreters for instructions with verifiable outputs (e.g., length, style constraints), and (2) LLM-as-judge evaluation for instructions requiring nuanced understanding of constraints. To address potential adversarial behaviors where models might claim instruction fulfillment without actual compliance, we incorporate an additional hack-check layer that specifically detects such deceptive claims.

Multi-Source Instruction Generation. To construct our training data, we employ three distinct generation strategies to ensure comprehensive coverage: (1) expert-crafted complex conditional prompts and rubrics developed by our data

team (2) agentic instruction augmentation inspired by AutoIF [12], and (3) a fine-tuned model specialized for generating additional instructions that probe specific failure modes or edge cases. This multipronged approach ensures both breadth and depth in instruction coverage.

Faithfulness Faithfulness is essential for an agentic model operating in scenarios such as multi-turn tool use, self-generated reasoning chains, and open-environment interactions. Inspired by the evaluation framework from FACTS Grounding [30], we train a sentence-level faithfulness judge model to perform automated verification. The judge is effective in detecting sentences that make a factual claim without supporting evidence in context. It serves as a reward model to enhance overall faithfulness performance.

Coding & Software Engineering To enhance our capability in tackling competition-level programming problems, we gather problems and their judges from both open-source datasets [27, 83] and synthetic sources. To ensure the diversity of the synthetic data and the correctness of reward signals, we incorporate high-quality human-written unit tests retrieved from pre-training data.

For software engineering tasks, we collect a vast amount of pull requests and issues from GitHub to build software development environment that consists of user prompts/issues and executable unit tests. This environment was built on a robust sandbox infrastructure, powered by Kubernetes for scalability and security. It supports over 10,000 concurrent sandbox instances with stable performance, making it ideal for both competitive coding and software engineering tasks.

Safety Our work to enhance the safety begins with a human-curated set of seed prompts, manually crafted to encompass prevalent risk categories such as violence, fraud, and discrimination.

To simulate sophisticated jailbreak attempts (e.g., role-playing, literary narratives, and academic discourse), we employ an automated prompt evolution pipeline with three key components:

- **Attack Model**: Iteratively generates adversarial prompts designed to elicit unsafe responses from the target LLM.
- **Target Model**: Produces responses to these prompts, simulating potential vulnerabilities.
- **Judge Model**: Evaluates the interaction to determine if the adversarial prompt successfully bypasses safety mechanisms.

Each interaction is assessed using a task-specific rubric, enabling the judge model to provide a binary success/failure label.

3.2.2 Beyond Verification: Self-Critique Rubric Reward

To extend model alignment beyond tasks with verifiable reward, we introduce a framework for general reinforcement learning from self-critic feedbacks. This approach is designed to align LLMs with nuanced human preferences, including helpfulness, creativity, depth of reasoning, factuality, and safety, by extending the capabilities learned from verifiable scenarios to a broader range of subjective tasks. The framework operates using a *Self-Critique Rubric Reward* mechanism, where the model evaluates its own outputs to generate preference signals. To bootstrap K2 as a competent judge, we curated a mixture of open-source and in-house preference datasets and initialize its critic capability in the SFT stage.

Self-Critiqued Policy Optimization In the first core process of the learning loop, the K2 actor generates responses for general prompts that cover a wide range of use cases. The K2 critic then ranks all results by performing pairwise evaluations against a combination of rubrics, which incorporates both *core rubrics* (Appendix. F.1), which represent the fundamental values of our AI assistant that Kimi cherish, prescriptive rubrics (Appendix. F.2) that aim to eliminate reward hacking, and *human-annotated rubrics* crafted by our data team for specific instructional contexts. Although certain rubrics can be designated as mandatory, K2 retains the flexibility to weigh them against its internal priors. This capacity enables a dynamic and continuous alignment with its evolving on-policy behavior, ensuring that the model's responses remain coherent with its core identity while adapting to specific instructions.

Closed-Loop Critic Refinement and Alignment During RL training, the critic model is refined using verifiable signals. On-policy rollouts generated from verifiable-reward prompts are used to continuously update the critic, a crucial step that distills objective performance signals from RLVR directly into its evaluation model. This transfer learning process grounds its more subjective judgments in verifiable data, allowing the performance gains from verifiable tasks to enhance the critic's judgment on complex tasks that lack explicit reward signals. This closed-loop process ensures that the critic continuously recalibrates its evaluation standards in lockstep with the policy's evolution. By

grounding subjective evaluation in verifiable data, the framework enables robust and scalable alignment with complex, non-verifiable human objectives.

Consequently, this holistic alignment yields comprehensive performance improvements across a wide spectrum of domains, including user intent understanding, creative writing, complex reasoning, and nuanced language comprehension.

3.2.3 RL Algorithm

We adopt the policy optimization algorithm introduced in K1.5 [35] as the foundation for K2. For each problem x, we sample K responses $\{y_1, \ldots, y_k\}$ from the previous policy π_{old}, and optimize the model π_θ with respect to the following objective:

$$L_{\text{RL}}(\theta) = \mathbb{E}_{x \sim \mathcal{D}} \left[\frac{1}{K} \sum_{i=1}^{K} \left[\left(r(x, y_i) - \bar{r}(x) - \tau \log \frac{\pi_\theta(y_i|x)}{\pi_{\text{old}}(y_i|x)} \right)^2 \right] \right],$$

where $\bar{r}(x) = \frac{1}{k} \sum_{i=1}^{k} r(x, y_i)$ is the mean rewards of the sampled responses, $\tau > 0$ is a regularization parameter that promotes stable learning. As in SFT, we employ the Muon optimizer [33] to minimize this objective. As we scale RL training to encompass a broader range of tasks in K2, a primary challenge is achieving consistent performance improvements across all domains. To address this, we introduce several additions to the RL algorithm.

Budget Control It has been widely observed that RL often results in a substantial increase in the length of model-generated responses [35, 19]. While longer responses can enable the model to utilize additional test-time compute for improved performance on complex reasoning tasks, the benefits often do not justify its inference cost in non-reasoning domains. To encourage the model to properly distribute inference budget, we enforce a per-sample *maximum token budget* throughout RL training, where the budget is determined based on the type of task. Responses that exceed this token budget are truncated and assigned a penalty, which incentivizes the model to generate solutions within the specified limit. Empirically, this approach significantly enhances the model's token efficiency, encouraging concise yet effective solutions across all domains.

PTX Loss To prevent the potential forgetting of valuable, high-quality data during joint RL training, we curate a dataset comprising hand-selected, high-quality samples and integrate it into the RL objective through an auxiliary PTX loss [54]. This strategy not only leverages the advantages of high-quality data, but also mitigates the risk of overfitting to the limited set of tasks explicitly present in the training regime. This augmentation substantially improves the model's generalization across a broader range of domains.

Temperature Decay For tasks such as creative writing and complex reasoning, we find that promoting exploration via a high sampling temperature during the initial stages of training is crucial. A high temperature allow the model to generate diverse and innovative responses, thereby facilitating the discovery of effective strategies and reducing the risk of premature convergence to suboptimal solutions. However, retaining a high temperature in the later stages of training or during evaluation can be detrimental, as it introduces excessive randomness and compromises the reliability and consistency of the model's outputs. To address this, we employ a temperature decay schedule, to shift from exploration to exploitation throughout the training. This strategy ensures that the model leverages exploration when it is most beneficial, while ultimately converge on stable and high-quality outputs.

3.3 RL Infrastructure

3.3.1 Colocated Architecture

Similar to K1.5 [35], we adopt a hybrid colocated architecture for our synchronized RL training, where the training and inference engines live on the same workers. When one engine is actively working, the other engine releases or offloads its GPU resources to accommodate. In each iteration of RL training, a centralized controller first calls the inference engine to generate new data for training. It then notifies the training engine to train on the new data, and send updated parameters to the inference engine for the next iteration.

Each engine is heavily optimized for throughput. In addition, as the model scales to the size of K2, the latency of engine switching and failure recovery becomes significant. We present our system design considerations in these aspects.

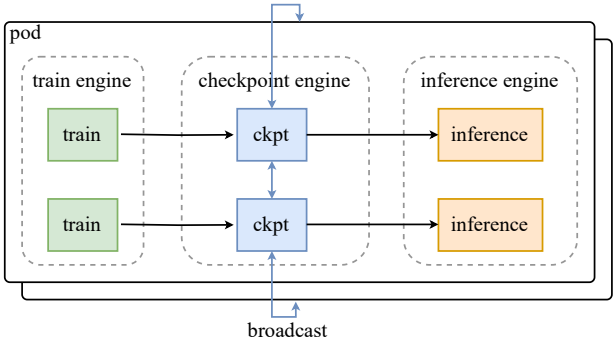


Figure 10: Parameter update utilizing a checkpoint engine

3.3.2 Efficient Engine Switching

During rollout, the parameters of the training engine are offloaded to DRAM. Bringing up the training engine is therefore a simple step of H2D transmission. However, bringing up the inference engine is a bigger challenge, as it must obtain updated parameters from the training engine with a different sharding paradigm.

Given the scale of K2 and the vast number of devices involved, using a network file system for resharding and broadcasting parameters is impractical. The aggregate bandwidth required to keep overhead low reaches several petabytes per second. To address this challenge, we developed a distributed checkpoint engine co-located on training nodes to manage parameter states. To perform a parameter update, each checkpoint engine worker obtains a local copy of parameters from the training engine, then broadcasts the full parameter set across all checkpoint engine workers. Subsequently, the inference engine retrieves only the parameter shard it requires from the checkpoint engine. This process is illustrated in Figure 10. To enable this for a 1T model, updates are performed parameter-by-parameter in a pipelined manner, minimizing memory footprint (see Appendix G).

We opt to broadcast the full parameter set across the entire cluster, regardless of the specific sharding schemes on each inference worker. While this transfers several times more data than a theoretically optimal approach, it offers a simpler system design that is less intrusive to the training and inference engines. We chose to trade off this minor overhead to fully decouple the training engine and the inference engine, significantly simplifying maintenance and testing.

Notably, this approach outperforms the transfer-what-you-need method due to reduced synchronization overhead and higher network bandwidth utilization. Our system can complete a full parameter update for Kimi K2 with less than 30 seconds, a negligible duration for a typical RL training iteration.

3.3.3 Efficient System Startup

As large-scale training is prone to system failure, optimizing the startup time is crucial for models as large as Kimi K2.

To start the training engine, we let each training worker selectively read part or none of the parameters from disk, and broadcast necessary parameters to its peers. The design goal is to ensure all workers collectively read the checkpoint only once, minimizing expensive disk IO.

As the inference engines are independent replicas, we would like to avoid introducing extra synchronization barriers between them. Therefore, we opt to reuse checkpoint engine for startup: we let checkpoint engine collectively read the checkpoint from disk, similar to how the training engine starts. Then it updates the state of the uninitialized inference engine, using the approach introduced in the previous section. By leveraging the dedicated checkpoint engine, the system also becomes robust to single-point failures, because an inference replica can restart without communicating with other replicas.

3.3.4 Agentic Rollout

Our RL infrastructure supports the training of long-horizon, multi-turn agentic tasks. During rollout, these tasks present distinct challenges, such as complex environmental interactions and prolonged rollout durations. Here we introduce a few optimizations to alleviate these issues.

Due to the diversity of environments, certain interactions may be blocked on waiting for environment feedback (e.g., a virtual machine or a code interpreter), leaving the GPUs idle. We employ two strategies to maximize GPU utilization:

(i) we deploy heavy environments as dedicated services that can scale up more easily; (ii) we employ a large number of concurrent rollouts to amortize the latency induced by certain expensive interactions.

Another challenge in agentic rollout is that individual rollout trajectories can be extremely long. To prevent long-tail trajectories from blocking the entire rollout process, we employ the partial rollout [35] technique. This strategy allows long-tail unfinished tasks to be paused, and resumed in the next RL iteration.

To improve research efficiency, we also design a unified interface inspired by the OpenAI Gym framework [49] to streamline the integration of new environments. We hope to scale our RL infrastructure to more diverse interactive environments in the future.

4 Evaluations

This section begins with the post-training evaluation of Kimi-K2-Instruct, followed by a brief overview of the capabilities of Kimi-K2-Base. We conclude with a comprehensive safety evaluation.

4.1 Post-training Evaluations

4.1.1 Evaluation Settings

Benchmarks We assess Kimi-K2-Instruct across different areas. For coding, we adopt LiveCodeBench v6 [31](questions from August 2024 to May 2025), OJBench [77], MultiPL-E [5], SWE-bench Verified [32, 84], TerminalBench [71], Multi-SWE-bench [86], SWE-Lancer [50], PaperBench [65], and Aider-Polyglot [16]. For tool use tasks, we evaluate performance on τ^2-Bench [3] and AceBench [6], which emphasize multi-turn tool-calling capabilities. In reasoning, we include a wide range of mathematical, science and logical tasks: AIME 2024/2025, MATH-500, HMMT 2025, CNMO 2024, PolyMath-en, ZebraLogic [43], AutoLogi [91], GPQA-Diamond [61], SuperGPQA [13], and Humanity's Last Exam (Text-Only) [56]. We benchmark the long-context capabilities on: MRCR[4] for long-context retrieval, and DROP [14], FRAMES [37] and LongBench v2 [2] for long-context reasoning. For factuality, we evaluate FACTS Grounding [30], the Vectara Hallucination Leaderboard [73], and FaithJudge [68]. Finally, general capabilities are assessed using MMLU [23], MMLU-Redux [17], MMLU-Pro [76], IFEval [90], Multi-Challenge [64], SimpleQA [78], and LiveBench [80] (as of 2024-11-25).

Baselines We benchmark against both open-source and proprietary frontier models, ensuring every candidate is evaluated under its non-thinking configuration to eliminate additional gains from test-time compute. Open-source baselines: DeepSeek-V3-0324 and Qwen3-235B-A22B, with the latter run in the vendor-recommended no-thinking regime. Proprietary baselines: Claude Sonnet 4, Claude Opus 4, GPT-4.1, and Gemini 2.5 Flash Preview (2025-05-20). Each invoked in its respective non-thinking mode via official APIs under unified temperature and top-p settings.

Evaluation Configurations All runs query models in their non-thinking mode. Output token length is capped at 8192 tokens everywhere except SWE-bench Verified (Agentless), which is raised to 16384. For benchmarks with high per-question variance, we adopt repeated sampling k times and average the results to obtain stable scores, denoted as Avg@k. For long-context tasks, we set the context window size to 128K tokens during evaluation, truncating any input that exceeds this limit to fit within the window. SWE-bench Verified is evaluated in two modes: Agentless Coding via Single Patch without Test (Acc) and Agentic Coding via bash/editor tools under both Single Attempt (Acc) and Multiple Attempts (Acc) using best-of-N selection with an internal verifier; SWE-bench Multilingual is tested only in the single-attempt agentic setting. Some data points have been omitted due to prohibitively expensive evaluation costs.

4.1.2 Evaluation Results

A comprehensive evaluation results of Kimi-K2-Instruct is shown in Table 3, with detailed explanation provided in the Appendix C. Below, we highlight key results across four core domains:

Agentic and Competitive Coding Kimi-K2-Instruct demonstrates state-of-the-art open-source performance on real-world SWE tasks. It outperforms most baselines on SWE-bench Verified (65.8%, 71.6% with multiple attemps), SWE-bench Multilingual (47.3%), and SWE-lancer (39.1%), significantly closing the gap with Claude 4 Opus and Sonnet. On competitive coding benchmarks (e.g., LiveCodeBench v6 53.7%, OJBench 27.1%), it also leads among all models, highlighting its practical coding proficiency across difficulty levels.

[4] https://huggingface.co/datasets/openai/mrcr

Table 3: Performance comparison of Kimi-K2-Instruct against leading open-source and proprietary models across diverse tasks. **Bold** denotes the global SOTA; **underlined bold** indicates the best open-source result. Data points marked with * are taken directly from the model's technical report or blog.

Benchmark	Open Source			Proprietary			
	Kimi-K2-Instruct	DeepSeek-V3-0324	Qwen3-235B-A22B	Claude Sonnet 4	Claude Opus 4	GPT-4.1	Gemini 2.5 Flash
Coding Tasks							
LiveCodeBench v6 (Pass@1)	**53.7**	46.9	37.0	48.5	47.4	44.7	44.7
OJBench (Pass@1)	**27.1**	24.0	11.3	15.3	19.6	19.5	19.5
MultiPL-E (Pass@1)	**85.7**	83.1	78.2	88.6	**89.6**	86.7	85.6
SWE-bench Verified *Agentless-Single-Patch* (Pass@1)	**51.8**	36.6	39.4	50.2	**53.0**	40.8	32.6
SWE-bench Verified *Agentic-Single-Attempt* (Pass@1)	**65.8**	38.8	34.4	**72.7***	72.5*	54.6	—
SWE-bench Verified *Agentic-Multi-Attempt* (Pass@1)	**71.6**	—	—	**80.2***	79.4*	—	—
SWE-bench Multilingual (Pass@1)	**47.3**	25.8	20.9	**51.0**	—	31.5	—
Multi-SWE-bench (Pass@1)	**18.3**	8.0	9.0	**29.2**	—	11.7	14.0
SWE-Lancer (Pass@1)	**39.1**	30.5	24.1	**40.8**	—	23.0	38.5
Paper Bench *Code-Dev* (Acc.)	**27.8**	12.2	13.2	**43.3**	—	29.9	5.7
Terminal Bench *In-House* (Acc.)	**30.0**	—	—	35.5	**43.2**	8.3	—
Terminal Bench *Terminus* (Acc.)	**25.0**	16.3	6.6	—	—	**30.3**	16.8
Aider-Polyglot (Acc.)	60.0	55.1	**61.8**	56.4	**70.7**	52.4	44.0
Tool Use Tasks							
Tau2 retail (Avg@4)	**70.6**	69.1	57.0	75.0	**81.8**	74.8	64.3
Tau2 airline (Avg@4)	**56.5**	39.0	26.5	55.5	**60.0**	54.5	42.5
Tau2 telecom (Avg@4)	**65.8**	32.5	22.1	45.2	57.0	38.6	16.9
AceBench (Acc.)	**76.5**	72.7	70.5	76.2	75.6	**80.1**	74.5
Math & STEM Tasks							
AIME 2024 (Avg@64)	**69.6**	59.4*	40.1*	43.4	48.2	46.5	61.3
AIME 2025 (Avg@64)	**49.5**	46.7	24.7*	33.1*	33.9*	37.0	46.6
MATH-500 (Acc.)	**97.4**	94.0*	91.2*	94.0	94.4	92.4	95.4
HMMT 2025 (Avg@32)	**38.8**	27.5	11.9	15.9	15.9	19.4	34.7
CNMO 2024 (Avg@16)	74.3	**74.7**	48.6	60.4	57.6	56.6	**75.0**
PolyMath-en (Avg@4)	**65.1**	59.5	51.9	52.8	49.8	54.0	49.9
ZebraLogic (Acc.)	**89.0**	84.0	37.7*	79.7	59.3	58.5	57.9
AutoLogi (Acc.)	**89.5**	88.9	83.3*	89.8	86.1	88.2	84.1
GPQA-Diamond (Avg@8)	**75.1**	68.4*	62.9*	70.0*	74.9*	66.3	68.2
SuperGPQA (Acc.)	**57.2**	53.7	50.2	55.7	56.5	50.8	49.6
Humanity's Last Exam (Acc.)	4.7	5.2	**5.7**	5.8	**7.1**	3.7	5.6
General Tasks							
MMLU (EM)	**89.5**	89.4	87.0	91.5	**92.9**	90.4	90.1
MMLU-Redux (EM)	**92.7**	90.5	89.2*	93.6	**94.2**	92.4	90.6
MMLU-Pro (EM)	81.1	**81.2***	77.3	83.7	**86.6**	81.8	79.4
IFEval (Prompt Strict)	**89.8**	81.1	83.2*	87.6	87.4	88.0	84.3
Multi-Challenge (Acc.)	**54.1**	31.4	34.0	46.8	49.0	36.4	39.5
SimpleQA (Correct)	**31.0**	27.7	13.2	15.9	22.8	**42.3**	23.3
Livebench (Pass@1)	**76.4**	72.4	67.6	74.8	74.6	69.8	67.8
Arena Hard v2.0 *Hard Prompt* (Win rate)	**54.5**	39.9	39.9	51.6	**59.7**	51.7	48.7
Arena Hard v2.0 *Creative Writing* (Win rate)	**85.0**	59.3	59.8	54.6	68.5	61.5	72.8
FACTS Grounding (Adjusted)	**88.5**	68.3	68.5	83.6	—	79.2	86.6
HHEM v2.1 (1-Hallu.)	**98.9**	88.9	94.5	94.5	—	96.7	97.8
FaithJudge (1-Hallu.)	**92.6**	83.4	75.7	83.0	—	91.0	**93.2**
LongBench v2 (Acc.)	49.1	**51.1**	—	52.5	—	54.3	**55.5**
FRAMES (Acc.)	77.1	**79.2**	—	76.3	—	**87.4**	72.9
MRCR (Acc.)	**55.0**	50.8	—	74.4	—	66.9	**81.7**
DROP (Acc.)	**93.5**	91.2	84.3	92.0	—	79.1	81.7