Building a Next-Frontier Small LLM for Agentic Coding: A Technical Blueprint

1. The Core Philosophy: Post-Training as the Key Differentiator

1.1. The Shift from Scale to Sophisticated Post-Training

The development of next-generation language models for agentic coding and software engineering is undergoing a fundamental paradigm shift. While the industry has long been driven by the "bigger is better" mantra, focusing on increasing model parameters and training data volume, the frontier of performance is now being defined by the sophistication of the post-training process. The success of models like **Kimi K2**, **MiniMax M2**, and **GLM 4.6** demonstrates that raw scale is no longer the sole determinant of agentic capability. Instead, the ability to transform a powerful but generalist base model into a specialized, reasoning, and tool-using agent through advanced post-training techniques has become the critical differentiator. This evolution is driven by the inherent scarcity of high-quality, naturally occurring agentic data. Capabilities such as **multi-step reasoning, long-term planning, and complex tool use** are rarely found in standard web-scraped corpora, making their acquisition and scaling prohibitively expensive. Consequently, top-tier AI labs are investing heavily in creating structured, high-quality agentic trajectories synthetically and employing general reinforcement learning (RL) frameworks that can instill these complex behaviors into a pre-trained model. This strategic pivot elevates the importance of **token efficiency**—maximizing the learning signal per token—and scalable data synthesis, positioning post-training not as a final polish but as the core engine for achieving agentic intelligence .

The challenges of building agentic models are twofold, spanning both the pre-training and post-training phases. During pre-training, the primary constraint is the limited availability of high-quality data that can provide the necessary general-purpose priors for agentic behavior. This elevates token efficiency to a critical scaling coefficient, pushing researchers to develop novel optimizers and data generation techniques to extract maximum intelligence from every available token . However, it is in post-training where these priors are forged into actionable, reliable agentic skills. The rarity of complex agentic behaviors in natural data necessitates a deliberate and systematic approach to data creation and model refinement. Top labs are moving beyond simple supervised fine-tuning (SFT) on curated datasets and are embracing multi-stage

pipelines that combine large-scale data synthesis with advanced RL algorithms. These pipelines are designed to teach the model not just *what* to do, but *how* to think, plan, and interact with its environment. By creating **verifiably correct agentic interactions at scale** and rewarding the model for successful task completion, these methods bridge the gap between a model's latent knowledge and its practical application in complex, real-world scenarios. This focus on post-training represents a more intelligent and efficient path to frontier performance, where the quality and structure of the training signal matter more than the sheer quantity of data .

1.2. The Multi-Stage Post-Training Pipeline

The architecture of a modern post-training pipeline for agentic models is a complex, multi-stage process designed to systematically build and refine sophisticated capabilities. This pipeline moves beyond traditional fine-tuning, which primarily adapts a model to a new domain or style, and instead focuses on teaching the model to perform as an autonomous agent. The process begins with the creation of a massive, high-quality dataset of agentic interactions, which is then used to prime the model through supervised learning. This is followed by a more advanced stage of reinforcement learning, where the model is trained to optimize for successful task completion in dynamic environments. This structured approach ensures that the model not only learns the correct actions but also develops the underlying reasoning and planning skills necessary for robust agentic performance. The **Kimi K2 model**, for instance, exemplifies this methodology by employing a pipeline that includes a large-scale agentic data synthesis system, supervised fine-tuning on the generated trajectories, and a general reinforcement learning framework that incorporates both verifiable rewards and self-critique mechanisms . This comprehensive pipeline is what enables models like Kimi K2 to achieve state-of-the-art performance on a wide range of agentic and frontier benchmarks, demonstrating the power of a well-designed post-training process.

The multi-stage pipeline is a carefully orchestrated sequence where each stage builds upon the last, progressively enhancing the model's capabilities. The initial stage, **Large-Scale Agentic Data Synthesis**, is foundational, as it addresses the critical lack of natural agentic data. This involves creating a system that can generate diverse and verifiably correct demonstrations of tool use, multi-turn interactions, and problem-solving in both simulated and real-world environments. The quality and diversity of this synthetic data are paramount, as it forms the basis for all subsequent learning. The second stage, **Supervised Fine-Tuning (SFT) on High-Quality Trajectories**, uses this

curated dataset to teach the model the basic patterns of agentic behavior. During SFT, the model learns to mimic the successful trajectories, absorbing the strategies and formats required for effective tool use and reasoning. The final and most sophisticated stage, **Reinforcement Learning with Verifiable Rewards (RLVR)** , takes the SFT-trained model and refines its decision-making abilities. By placing the model in interactive environments and rewarding it for successful outcomes (e.g., passing tests, completing tasks), RLVR teaches the model to generalize beyond the initial demonstrations and develop its own robust problem-solving strategies. This combination of data synthesis, imitation learning, and reinforcement learning creates a powerful feedback loop that drives the model toward true agentic intelligence .

1.2.1. Stage 1: Large-Scale Agentic Data Synthesis

The first and most critical stage in the post-training pipeline is the systematic generation of high-quality, diverse, and verifiably correct agentic data. This process is essential because the complex behaviors required for agentic coding—such as multi-step planning, tool invocation, and error recovery—are sparsely represented in naturally occurring text corpora. To overcome this scarcity, top AI labs have developed sophisticated data synthesis pipelines that can create a vast array of agentic interactions at scale. The **Kimi K2 project**, for example, introduced a large-scale system designed to generate tool-use demonstrations by constructing diverse tools, agents, tasks, and trajectories within both simulated and real-world environments . This approach allows for the creation of high-fidelity interactions that are not only diverse but also guaranteed to be correct, providing a solid foundation for training. The system is designed to be comprehensive, covering a wide range of scenarios and edge cases to ensure that the model is exposed to a rich and varied set of problem-solving experiences. This synthetic data generation is a key enabler of agentic intelligence, as it provides the raw material from which the model can learn the intricate patterns of effective agentic behavior.

The data synthesis pipeline is not merely about generating large volumes of data; it is about creating structured and meaningful learning experiences for the model. The process involves defining a set of tools and APIs that the agent can use, creating a variety of tasks that require the use of these tools, and then generating complete interaction trajectories that demonstrate successful task completion. These trajectories can include multiple turns of reasoning, tool calls, and responses from the environment, providing a rich, contextual learning signal. The use of simulated environments allows for the creation of complex and challenging scenarios that would be difficult or

impossible to replicate with real-world data alone. Furthermore, the ability to **verify the correctness of the generated trajectories** is crucial, as it ensures that the model is learning from high-quality examples. This systematic approach to data generation is a significant departure from traditional data collection methods and represents a key innovation in the development of agentic models. By creating a scalable and reliable source of high-quality agentic data, this stage of the pipeline lays the groundwork for all subsequent training and is a critical factor in achieving frontier performance .

1.2.2. Stage 2: Supervised Fine-Tuning (SFT) on High-Quality Trajectories

Once a substantial corpus of high-quality agentic trajectories has been synthesized, the next stage in the pipeline is **Supervised Fine-Tuning (SFT)** . This stage serves as the initial phase of behavioral cloning, where the model is trained to imitate the successful patterns and strategies demonstrated in the synthetic data. The goal of SFT is to provide the model with a strong foundational understanding of agentic behavior, including how to format tool calls, interpret environmental feedback, and structure its reasoning process. By training on a diverse set of verifiably correct trajectories, the model learns to associate specific problem contexts with effective sequences of actions. This process effectively primes the model for the more advanced reinforcement learning stage that follows, providing it with a solid baseline of agentic capabilities. The quality of the SFT data is paramount, as any errors or inconsistencies in the training set can be learned and amplified by the model. Therefore, the rigorous data curation and verification processes employed in the synthesis stage are directly reflected in the effectiveness of the SFT phase.

The SFT stage is more than just simple imitation; it is about instilling a structured and coherent approach to problem-solving. The training data, consisting of multi-turn interactions, teaches the model how to maintain context over a long conversation, how to break down a complex task into a series of smaller, manageable steps, and how to adapt its strategy in response to new information. This is particularly important for agentic coding, where a model might need to read a codebase, identify a bug, write a test case to reproduce the issue, and then implement a fix. The SFT process helps the model to internalize this entire workflow, from initial analysis to final validation. The resulting model, after SFT, is not just a language model that can generate code; it is an agent that has been trained to follow a structured process for solving software engineering problems. This foundational training is crucial for the success of the subsequent RL stage, as it provides a well-behaved and capable starting point for further optimization .

1.2.3. Stage 3: Reinforcement Learning with Verifiable Rewards (RLVR)

The final and most sophisticated stage of the post-training pipeline is **Reinforcement Learning with Verifiable Rewards (RLVR)** . This stage is designed to move the model beyond simple imitation and teach it to generalize and develop its own robust problem-solving strategies. In the RLVR framework, the model is placed in an interactive environment where it can take actions (e.g., write code, run tests) and receive feedback in the form of rewards. The key innovation of RLVR is the use of **verifiable rewards**, which are based on objective, automatically checkable criteria such as the successful execution of code or the passing of a test suite. This approach is a significant improvement over traditional RL methods that rely on learned reward models, which can be brittle and prone to reward hacking. By grounding the reward signal in verifiable outcomes, RLVR provides a stable and reliable learning signal that encourages the model to develop genuinely effective solutions.

The RLVR process is a powerful mechanism for refining the model's decision-making abilities. It allows the model to explore different approaches to a problem, learn from its mistakes, and discover novel solutions that may not have been present in the initial SFT data. This is particularly important for agentic coding, where there are often multiple valid ways to solve a problem, and the best solution may depend on the specific context. The **Kimi K2 model**, for example, uses a general RL framework that combines verifiable rewards with a **self-critique rubric reward mechanism** . This allows the model to learn not only from externally defined tasks but also from evaluating its own outputs, extending its alignment from static domains into more open-ended ones. This combination of external and internal feedback creates a powerful learning loop that drives the model toward a deeper understanding of the problem domain and enables it to produce more robust and reliable solutions. The RLVR stage is the culmination of the post-training pipeline, transforming a capable but passive model into a truly agentic system that can reason, plan, and act autonomously to achieve its goals.

2. Advanced Post-Training Techniques from Top-Tier Labs

2.1. Reinforcement Learning with Verifiable Rewards (RLVR)

Reinforcement Learning with Verifiable Rewards (RLVR) has emerged as a cornerstone of modern post-training for agentic coding models, representing a fundamental shift away from purely probabilistic training objectives towards a paradigm that directly optimizes for objective, verifiable correctness. This approach is particularly potent in domains like software engineering and mathematics, where the quality of a model's

output can be determined by an unambiguous, automated process, such as the execution of a test suite or the evaluation of a logical proof. By providing a clear, binary signal of success or failure, RLVR allows the model to learn complex, multi-step reasoning patterns that lead to correct solutions, effectively navigating the vast search space of potential code generations. This method has been instrumental in the success of models like **Kimi K2**, which leverages a "Verifiable Rewards Gym" to achieve state-of-the-art performance on benchmarks like **SWE-bench Verified** . The core innovation of RLVR lies in its ability to provide dense, reliable feedback, enabling stable and efficient policy optimization even for long-horizon tasks that would be intractable for traditional RL methods relying on sparse or subjective rewards.

2.1.1. Core Concept: Rewarding Correctness over Probabilities

The central tenet of Reinforcement Learning with Verifiable Rewards (RLVR) is the replacement of the traditional language modeling objective—maximizing the likelihood of the next token—with a reward signal derived from an external, deterministic verification process. Instead of learning to predict the most probable sequence of tokens based on a corpus of human-written text, the model is trained to generate sequences that produce a positive outcome when subjected to a verifiable test. In the context of agentic coding, this means the reward for a generated piece of code is not based on its stylistic similarity to existing code but on whether it **compiles, passes a predefined set of unit tests, and successfully resolves a specified issue** within a software repository. This approach, as implemented in Kimi K2's post-training pipeline, utilizes a simple, rule-based reward function that assigns a binary score: a reward of **1 for a correct output and 0 for an incorrect one** . This clear, unambiguous feedback mechanism allows the model to learn the causal relationship between its actions (the code it generates) and the desired outcome (a passing test suite), fostering the development of robust problem-solving capabilities rather than mere pattern matching.

The power of this paradigm is its ability to scale learning to complex, multi-step problems. For instance, when tackling a real-world bug from SWE-bench, the model must perform a series of actions: read the issue description, locate the relevant files in a codebase, understand the existing logic, generate a patch, and then execute the test suite to verify the fix. The final pass/fail signal from the test suite serves as the reward for the entire trajectory of actions. This dense, outcome-oriented feedback is far more informative for learning complex reasoning than the sparse, token-level supervision of traditional pre-training. Kimi K2's framework extends this concept into a **"Verifiable Rewards Gym,"** an extensible collection of task templates with well-defined evaluation

logic across domains like coding, mathematics, logic, and safety . This gym environment allows the model to be trained on a vast array of problems, each with its own verifiable ground truth, thereby building a generalized capability for systematic problem-solving that is directly applicable to real-world software engineering challenges.

2.1.2. Implementation: Using PPO (Proximal Policy Optimization)

The implementation of Reinforcement Learning with Verifiable Rewards (RLVR) requires a robust and stable policy optimization algorithm to effectively translate the binary reward signals into meaningful model updates. **Proximal Policy Optimization (PPO)** has become the algorithm of choice for this task in many top-tier AI labs, including Moonshot AI for the development of Kimi K2 . PPO is particularly well-suited for this environment because it addresses the key challenge of training stability in RL for large language models. It prevents the policy from changing too drastically in a single update, which is a common cause of training collapse and performance degradation. This is achieved through a **"clipped" objective function** that constrains the policy update to a small, trusted region, ensuring that each step leads to a monotonic improvement in performance. This stability is crucial when dealing with the sparse, binary rewards of RLVR, as it allows the model to explore the solution space for complex problems without the risk of catastrophic forgetting or divergence.

The specific formulation of the RL objective used in Kimi K2's post-training, as detailed in its technical report, is a variant of PPO designed for this context . For each problem, the model samples K responses from its current policy. The optimization objective then aims to maximize the expected reward of these responses, regularized by a term that penalizes large deviations from the previous policy. The objective function is given by:

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

Here, $r(x, y_i)$ is the reward for response y_i to problem x, $\bar{r}(x)$ is the mean reward across the K samples, and the final term is the KL-divergence penalty that enforces the trust region. This formulation allows the model to learn from the relative performance of its own generated samples, effectively creating a preference signal even from sparse binary rewards. The use of PPO, combined with this carefully designed objective, provides a stable and efficient mechanism for the model to

iteratively refine its policy, learning to generate code that is not only plausible but demonstrably correct according to the verifiable reward function.

2.1.3. Application: Code Execution, Test Suite Pass/Fail, and Math Verification

The practical application of Reinforcement Learning with Verifiable Rewards (RLVR) is most evident in its ability to train models on tasks where correctness can be automatically and objectively measured. In the domain of agentic coding, this primarily involves the **execution of generated code against a suite of tests**. The Kimi K2 model, for example, was trained using a "Verifiable Rewards Gym" that includes a wide range of coding problems, from simple function implementation to complex, multi-file bug fixes in real-world repositories . For each problem, the model generates a candidate solution (e.g., a code patch), which is then executed in a sandboxed environment. The reward is determined by the outcome of this execution: a reward of **1 is given if the code compiles without errors and passes all provided unit tests**, and a **0 otherwise**. This process allows the model to learn the intricate logic of programming, including syntax, semantics, and algorithmic correctness, by directly interacting with a code execution environment and receiving immediate feedback on its attempts.

This paradigm extends beyond simple coding tasks to more complex software engineering workflows. On benchmarks like **SWE-bench Verified**, the model must not only generate a correct patch but also navigate a real-world codebase, understand the context of a bug report, and use tools like a code editor and a bash shell to implement and test its solution . The final reward is based on whether the full test suite of the repository passes with the model's patch applied. This provides a powerful learning signal for long-horizon, multi-step agentic behavior. The same principle is applied to other verifiable domains. For mathematical problems, the reward can be based on whether the model's final numerical answer matches the ground truth. For logical puzzles, it can be based on the correctness of the derived conclusion. By constructing a diverse "gym" of such tasks, Kimi K2's training process instills a generalized ability to reason systematically and produce outputs that are not just probable, but **verifiably correct**, a critical capability for any reliable agentic system .

2.2. Agentic Data Synthesis and Curation

A critical prerequisite for the successful application of advanced post-training techniques like RLVR is the availability of a massive, high-quality dataset of agentic interactions. Top-tier AI labs have moved beyond relying solely on human-annotated data, which is expensive and difficult to scale, and have instead developed

sophisticated pipelines for synthesizing agentic trajectories at scale. This process involves creating a simulated environment where an agent model can interact with a variety of tools and tasks, generating a rich dataset of multi-turn interactions that demonstrate successful problem-solving. The **Kimi K2 model**, for instance, was trained on data generated by a large-scale agentic data synthesis pipeline that systematically creates tool-use demonstrations in both simulated and real-world environments . This pipeline is designed to produce a diverse and verifiably correct set of trajectories that serve as the foundation for both the initial supervised fine-tuning (SFT) and the subsequent reinforcement learning (RL) phases. The quality and diversity of this synthetic data are paramount, as they directly influence the model's ability to learn robust and generalizable agentic behaviors.

2.2.1. Generating Diverse Agentic Trajectories

The generation of diverse and high-fidelity agentic trajectories is a multi-stage process that forms the bedrock of the post-training pipeline for models like Kimi K2. The goal is to create a vast dataset of multi-turn interactions where an agent successfully uses tools to accomplish a wide range of tasks. The process begins with the construction of a comprehensive repository of tool specifications. This includes both real-world APIs and a large number of synthetic tools, each with a well-defined schema and purpose . This diverse toolset ensures that the model is exposed to a wide variety of interaction patterns and is not overfitted to a specific domain. The next stage involves the generation of a "zoo" of agents and tasks. Different agent personas are created, each with a unique set of available tools and a specific goal. A corresponding set of tasks is then generated, designed to be solvable by one or more of these agents. This step is crucial for ensuring diversity in the dataset, as it forces the model to learn how to select and use the appropriate tools for a given situation.

The final and most critical stage is the generation of successful multi-turn trajectories. In this phase, a powerful **"teacher" model** (which can be a larger, more capable LLM) is used to simulate the agent's behavior. The teacher model is given a task and the specifications for its available tools. It then interacts with a simulated environment, making tool calls and processing the results, until it successfully completes the task. The entire interaction history—the prompt, the sequence of tool calls, and the final outcome—is recorded as a single trajectory. This process is repeated millions of times with different agents, tools, and tasks to create a massive dataset of verifiably correct agentic behavior. This synthetic data provides the rich, structured examples needed to

teach a smaller "student" model the fundamentals of tool use and multi-step reasoning through supervised fine-tuning, before it is further refined with reinforcement learning .

2.2.2. Using LLM-based Verifiers to Filter High-Quality Data

Once a large volume of agentic trajectories has been generated, the next critical step is to filter this data to ensure that only the highest-quality examples are used for training. This is where **LLM-based verifiers** play a crucial role. These verifiers are typically powerful language models that have been trained to assess the quality, correctness, and coherence of an agentic trajectory. They act as a quality control mechanism, automatically identifying and discarding trajectories that are incorrect, inefficient, or otherwise suboptimal. This automated filtering process is essential for scaling up the data curation process, as it would be prohibitively expensive and time-consuming to manually review the millions of trajectories required for training a frontier model. The Kimi K2 model's development process explicitly mentions the use of **"LLMs or human-based judges to perform automated quality evaluation and filtering"** as part of its data synthesis pipeline . This highlights the importance of a robust verification system in ensuring the quality of the training data.

The use of LLM-based verifiers offers several advantages over traditional filtering methods. First, they can be trained to evaluate trajectories based on a wide range of criteria, including not just the final outcome of the task, but also the efficiency and logical coherence of the intermediate steps. This allows for a more nuanced and comprehensive assessment of quality than a simple pass/fail metric. Second, they can be used to identify and correct minor errors in otherwise high-quality trajectories, further improving the overall quality of the dataset. Third, they can be scaled up to process massive datasets in a relatively short amount of time, making the entire data curation process more efficient. The Kimi K2 paper's emphasis on **"verifiably correct agentic interactions"** suggests a verification process that is both rigorous and reliable . By leveraging the power of LLM-based verifiers, top-tier AI labs are able to create high-quality training datasets that are essential for building the next generation of capable and reliable AI agents. This automated and scalable approach to data curation is a key enabler of the rapid progress being made in the field of agentic AI.

2.2.3. Simulating Multi-Turn Interactions with Tools and Environments

A critical aspect of agentic data synthesis is the simulation of **multi-turn interactions** with a variety of tools and environments. This is what distinguishes agentic data from standard instruction-following data; it captures the dynamic, back-and-forth nature of

an agent's interaction with the world. A single agentic task, such as fixing a bug in a software repository, will typically involve multiple turns, where the agent uses a tool (e.g., a code search tool), observes the result, formulates a new plan, and then uses another tool (e.g., a text editor). This multi-turn interaction is the essence of agentic behavior, and it is essential that the training data captures this complexity. The Kimi K2 model's development process explicitly mentions the generation of **"tool-use demonstrations via simulated and real-world environments,"** which implies a focus on these multi-turn interactions . The goal is to create a dataset that not only teaches the model how to use individual tools, but also how to orchestrate a sequence of tool calls to achieve a complex goal.

The simulation of these multi-turn interactions requires a sophisticated infrastructure that can provide the agent with a realistic and responsive environment. This might involve a simulated operating system with a file system, a command-line interface, and a set of pre-installed tools. The agent is then given a task and allowed to interact with this environment in a free-form manner, with its actions and the corresponding observations being recorded to form a trajectory. The Kimi K2 paper's mention of a **"data synthesis pipeline to teach models tool-use capabilities through multi-step, interactive reasoning"** further underscores the importance of this approach . By training on these simulated multi-turn interactions, the model learns to develop a "mental model" of the environment and the tools at its disposal, allowing it to plan and execute complex strategies. This is a significant step up from single-turn instruction following, and it is a key reason why models like Kimi K2 are able to achieve such impressive results on challenging agentic benchmarks like **SWE-Bench and Tau2-Bench** . The ability to effectively simulate and learn from these multi-turn interactions is a core competency for any organization seeking to build state-of-the-art agentic models.

2.3. Advanced Policy Optimization Algorithms

While Reinforcement Learning with Verifiable Rewards (RLVR) provides the foundational framework for training agentic models, the choice of the underlying policy optimization algorithm is critical for achieving stable and efficient learning, especially for long-horizon, complex tasks. The standard Proximal Policy Optimization (PPO) algorithm, while effective, can face challenges in this domain, such as training instability and difficulty in assigning credit across long sequences of actions. In response, leading AI labs have developed a new generation of advanced policy optimization algorithms specifically tailored for the unique demands of training LLM

agents. These novel methods introduce innovations in importance sampling, credit assignment, and exploration strategies to overcome the limitations of traditional approaches. Models like **MiniMax M2** and research papers from 2025 have introduced algorithms such as **CISPO (Context-aware Importance Sampling for Policy Optimization)** and **AEPO (Agentic Entropy-Balanced Policy Optimization)** , which represent the state of the art in this field . These algorithms are designed to handle the complexities of agentic workflows, where the model must make a series of interdependent decisions, often involving tool use and interaction with external environments, to achieve a final goal.

表格 复制

Algorithm	Model	Core Innovation
PPO	Kimi K2	Clipped objective function to constrain policy updates, ensu stability.
CISPO	MiniMax M2	Adjusts the "importance weight" of entire sequences rather individual tokens.
AEPO	Research (2025)	Entropy-based adaptive rollout scheme and a novel credit assignment mechanism.
GRPO	Qwen3-Coder	Optimizes policy based on the relative performance of a gro samples.

Table 1: Comparison of Advanced Policy Optimization Algorithms for Agentic LLMs.

2.3.1. CISPO (Context-aware Importance Sampling for Policy Optimization) from MiniMax M2

MiniMax M2, a leading open-weight model for agentic coding, introduced a novel post-training technique called **CISPO (Context-aware Importance Sampling for Policy Optimization)** to address the instability issues often encountered with traditional RL methods when applied to long, structured outputs like code generation . The core innovation of CISPO lies in its reformulation of the policy update mechanism. Instead of applying the clipping constraint to the policy ratio at the individual token level, as is done in standard PPO, CISPO adjusts the "importance weight" of entire sequences. This context-aware approach makes the training process significantly more stable, as it prevents the accumulation of small, per-token clipping errors that can lead to large,

destabilizing policy updates over long sequences . By operating at the sequence level, CISPO preserves the fine-grained gradient information at the token level, allowing for more effective learning while maintaining the stability needed for large-scale training.

The technical implementation of CISPO, as described in a 2025 research paper, involves reformulating the ratio clipping by applying the constraint to the importance sampling (IS) weights rather than the policy ratios themselves . This method bounds the magnitude of the policy update in expectation, providing a more stable and reliable training dynamic. This is particularly beneficial for agentic coding tasks, where a single incorrect token can render an entire code block useless, and the model must learn to generate long, coherent, and correct sequences. The success of MiniMax M2, which achieves a high score of **69.4 on the SWE-bench Verified** benchmark, is a testament to the effectiveness of this approach . CISPO represents a significant advancement in policy optimization for LLMs, providing a robust and scalable solution for training models on complex, long-horizon tasks where stability and sample efficiency are paramount.

2.3.2. AEPO (Agentic Entropy-Balanced Policy Optimization)

Agentic Entropy-Balanced Policy Optimization (AEPO) is another advanced RL algorithm, detailed in a 2025 research paper, that is specifically designed to enhance the training of multi-turn LLM agents . AEPO introduces two key innovations: an **entropy-based adaptive rollout scheme** and a novel credit assignment mechanism. The adaptive rollout scheme addresses the challenge of exploration in large action spaces. It dynamically allocates more sampling resources to reasoning paths where the model's uncertainty, measured by the entropy of its output distribution, is high. This allows the model to focus its exploration on areas of the problem space that are most likely to yield informative outcomes, improving sample efficiency and learning speed . This is particularly useful for agentic tasks, where the model must make a series of decisions, and some steps may be more critical or ambiguous than others.

The second key innovation in AEPO is its approach to credit assignment. In multi-step agentic workflows, it can be difficult to determine which specific action in a long trajectory was responsible for the final success or failure. AEPO addresses this by incorporating an **advantage attribution mechanism** that assigns credit across branching tool-use interactions. This allows the model to learn more effectively from its experiences, as it can better understand the causal links between its actions and the outcomes. The paper demonstrates the effectiveness of AEPO by testing it on a **Llama3.1-8B-Instruct** backbone, where it outperforms other state-of-the-art RL

algorithms like GRPO and CISPO on a range of mathematical and knowledge-intensive reasoning benchmarks . This makes AEPO a promising candidate for training next-generation small LLMs for agentic coding, as it provides a robust framework for handling the complexities of multi-step reasoning and tool use.

2.3.3. GRPO (Group Relative Policy Optimization) from Qwen3-Coder

In the pursuit of enhancing agentic capabilities, particularly in the domain of coding, the development of advanced policy optimization algorithms has become a key area of research. One such algorithm that has shown significant promise is **Group Relative Policy Optimization (GRPO)** , which was utilized in the training of the **Qwen3-Coder model** . GRPO is a sophisticated reinforcement learning technique designed to improve the model's ability to reason through complex tasks and generate more accurate and reliable code. The core idea behind GRPO is to optimize the model's policy not just based on the absolute reward of a single output, but **relative to a group of other candidate outputs**. This relative optimization approach helps to mitigate the effects of noisy or sparse reward signals, which are common in complex tasks like code generation. By comparing the model's output to a set of alternatives, GRPO can provide a more stable and informative learning signal, leading to more robust and generalizable improvements in performance.

The application of GRPO in the Qwen3-Coder model was a key factor in its impressive performance on a range of coding benchmarks. The model was trained using a multi-stage RL process, with GRPO being applied during the final stage to further refine its capabilities . This allowed the model to learn from a more nuanced and informative reward signal, leading to significant improvements in its ability to generate correct and efficient code. The success of GRPO in this context highlights the importance of developing specialized optimization algorithms that are tailored to the specific challenges of agentic tasks. As the complexity of these tasks continues to grow, the need for more advanced and sophisticated policy optimization techniques will only become more acute. The development of algorithms like GRPO is a clear indication that the field is moving in this direction, and it is likely that we will see many more such innovations in the coming years. The ability to effectively train and optimize agentic models is a key differentiator for top-tier AI labs, and the development of advanced algorithms like GRPO is a critical part of this process.

3. Architectural and Model Optimization Strategies

3.1. The Mixture-of-Experts (MoE) Architecture

In the pursuit of building next-generation language models with agentic capabilities, the architectural choice plays a pivotal role in balancing performance, efficiency, and scalability. The **Mixture-of-Experts (MoE)** architecture has emerged as a leading paradigm for achieving this balance, particularly for models targeting frontier performance levels. Unlike traditional dense transformer models, where every parameter is activated for every input, MoE models employ a sparse activation pattern. This means that for any given input, only a small subset of the model's parameters, known as "experts," are engaged in the computation. This design allows for the creation of extremely large models with a vast number of total parameters, which can capture a rich and nuanced understanding of the world, while keeping the number of active parameters—and thus the computational cost of inference—manageable. This **"large-but-sparse"** approach is particularly well-suited for agentic coding, where the model needs to have a broad base of knowledge to handle a wide variety of tasks, but the cost of deploying and running the model must remain practical.

The MoE architecture is not just a trick for reducing compute; it is a fundamental design choice that enables new levels of model capability. By having a large pool of specialized experts, the model can dynamically route different parts of an input to the most relevant experts, allowing for a more efficient and effective processing of information. This is analogous to how a team of human experts would tackle a complex problem, with each member contributing their specialized knowledge to the overall solution. For agentic coding, this could mean that one expert is specialized in understanding natural language requirements, another in generating Python code, and a third in debugging and testing. The MoE architecture allows the model to seamlessly combine the outputs of these experts to produce a coherent and effective solution. This ability to dynamically allocate computational resources based on the specific needs of the input is a key advantage of the MoE architecture and a major reason why it has become the preferred choice for top-tier AI labs building frontier models .

3.1.1. Achieving Large-Scale Performance with Small Active Parameters

The core advantage of the Mixture-of-Experts (MoE) architecture lies in its ability to **decouple the total number of model parameters from the number of parameters that are actively used during inference**. This is a crucial innovation for building large-scale models that are both powerful and practical to deploy. In a traditional dense model, doubling the number of parameters doubles the computational cost of every forward pass. In contrast, an MoE model can have an order of magnitude more total parameters while maintaining a constant computational cost, as long as the number of active

experts remains the same. This is achieved through a sparse activation mechanism, where a routing algorithm selects a small subset of experts to process each input token. This allows for the creation of models with a massive capacity for knowledge and nuanced understanding, without incurring the prohibitive computational and memory costs associated with dense models of a similar size.

This "large-but-sparse" design is particularly beneficial for agentic coding applications. A model designed for agentic coding needs to have a broad and deep understanding of a wide range of topics, including programming languages, software engineering principles, API documentation, and natural language instructions. A dense model with sufficient capacity to capture all of this knowledge would be extremely large and expensive to run. An MoE model, on the other hand, can achieve the same level of knowledge capacity with a much smaller active parameter count. This makes it feasible to deploy the model in real-world applications where latency and cost are critical considerations. Furthermore, the specialized nature of the experts in an MoE model can lead to more efficient and effective learning. By allowing the model to specialize different experts for different types of information, the MoE architecture can improve the model's ability to handle the diverse and complex inputs that are characteristic of agentic coding tasks. This combination of large-scale performance and practical efficiency is a key reason why the MoE architecture has become a cornerstone of modern LLM design .

3.1.2. Case Study: Kimi K2's 1T Total / 32B Active Parameter Model

The **Kimi K2 model** serves as a prime example of the power and effectiveness of the Mixture-of-Experts (MoE) architecture in building a frontier-level agentic model. Kimi K2 is a massive **1.04 trillion-parameter model**, but thanks to its MoE design, it only activates **32 billion parameters** for any given input . This represents a sparsity ratio of over 97%, meaning that less than 3% of the model's total capacity is used during inference. This remarkable efficiency is a key enabler of Kimi K2's impressive performance across a wide range of benchmarks. The model's large total parameter count allows it to store a vast amount of knowledge and develop a deep understanding of the world, which is essential for the complex reasoning and planning tasks required for agentic behavior. At the same time, its relatively small active parameter count makes it practical to train and deploy, even at a massive scale.

The architectural details of Kimi K2 further highlight the sophistication of its design. The model employs an ultra-sparse MoE architecture with **multi-head latent attention (MLA)** , a design choice that was derived from empirical scaling law analysis . This

attention mechanism is a key component of the model's efficiency, as it allows for the processing of long sequences without a quadratic increase in computational cost. The combination of the MoE architecture and the MLA attention mechanism creates a model that is both powerful and efficient, capable of handling the demanding requirements of agentic coding. The success of Kimi K2 demonstrates that the MoE architecture is not just a theoretical concept, but a practical and effective approach for building the next generation of AI models. By carefully balancing model size, sparsity, and architectural innovations, the developers of Kimi K2 have created a model that pushes the boundaries of what is possible in agentic intelligence .

3.1.3. Case Study: MiniMax M2's 230B Total / 10B Active Parameter Model

MiniMax M2 is another leading open-weight model that effectively utilizes the Mixture-of-Experts (MoE) architecture to deliver high performance in coding and agentic tasks. The model has a total of **230 billion parameters**, but only activates **10 billion** during inference, resulting in an activation rate of approximately 4.3% . This efficient design allows MiniMax M2 to achieve a balance of intelligence, speed, and cost-effectiveness, making it a powerful tool for a wide range of applications. The model's inference speed is reported to be around **100 tokens per second**, which is nearly double that of some competing models, a significant advantage for high-throughput production workloads and interactive applications . This speed, combined with its low active parameter count, makes MiniMax M2 an ideal candidate for building complex, multi-step agentic systems that require real-time responsiveness.

The performance of MiniMax M2 on key benchmarks underscores the effectiveness of its MoE design. The model achieves a score of **69.4 on SWE-bench Verified**, placing it in the same league as top proprietary models like GPT-5 and Claude Sonnet 4.5, but at a fraction of the cost . It also excels on competitive coding benchmarks, with a score of around **83% on LiveCodeBench**, which is on par with other leading models like GLM-4.6 . This high level of performance is a direct result of the model's architecture, which allows it to leverage a vast knowledge base while maintaining the efficiency required for practical deployment. The success of MiniMax M2 demonstrates that the MoE architecture is a key enabler for building next-generation small LLMs that can compete with much larger, denser models in the domain of agentic coding and software engineering.

3.2. Optimizing for Long Context

3.2.1. The Importance of Long Context for Agentic Workflows

The ability to process and understand long contexts is a critical requirement for any LLM designed for agentic workflows, particularly in the domain of software engineering . Real-world coding tasks often involve reasoning over large and complex codebases, which can easily exceed the context window of traditional models. An agentic model must be able to ingest and analyze entire files, understand the relationships between different parts of the code, and maintain a coherent understanding of the project over long interactions. This is essential for tasks like debugging, refactoring, and implementing new features, which require a deep and holistic understanding of the codebase. Without a sufficiently large context window, a model will be forced to rely on incomplete information, which can lead to incorrect or suboptimal solutions.

The importance of long context is not limited to code understanding. In a multi-turn agentic workflow, the model must be able to maintain a memory of the entire conversation history, including the user's instructions, the model's previous actions, and the feedback it has received. This is crucial for maintaining coherence and ensuring that the model's actions are aligned with the user's goals. A model with a limited context window will struggle to keep track of the broader context of the task, leading to a disjointed and ineffective interaction. The ability to handle long contexts is therefore a key enabler of the sophisticated reasoning and planning capabilities that are required for true agentic intelligence. This is why top-tier models like **Kimi K2 and Claude 4** have invested heavily in extending their context windows, with Kimi K2 supporting up to **128,000 tokens** and Claude 4 supporting up to **100,000 tokens** .

3.2.2. Techniques: YaRN for Context Extension up to 1M Tokens

To achieve these extended context windows, researchers have developed a variety of techniques, with **YaRN (Yet Another Recursive Network)** being a prominent example. YaRN is a method for extending the context window of a pre-trained language model without requiring any additional training. It works by modifying the attention mechanism to allow the model to attend to a much larger number of tokens than it was originally trained for. The **Qwen team**, for instance, used YaRN to extend the context window of their Qwen3-Coder model from its native 256K tokens up to an impressive **1 million tokens** . This capability is a game-changer for agentic coding, as it allows the model to work with even the largest codebases and to maintain context over extremely long and complex interactions. The ability to process such a large amount of information in a single session is a key enabler for the development of truly autonomous and capable coding agents.

3.2.3. Architectural Tweaks: Reducing Attention Heads for Stability

While extending the context window of a language model is crucial for agentic workflows, it also introduces significant technical challenges, particularly in terms of training stability . The standard attention mechanism used in transformers has a quadratic time and memory complexity with respect to sequence length, which can lead to computational bottlenecks and training divergence for very long contexts. To address this, researchers have developed a number of architectural tweaks and optimizations. One such tweak, employed in the **Kimi K2 model**, is the **reduction of the number of attention heads** in the transformer layers . This is a practical adjustment that helps to improve the stability of the model during training, particularly for very deep networks.

The rationale behind this tweak is that a large number of attention heads can lead to a more complex and less stable optimization landscape. By reducing the number of heads, the model becomes simpler and easier to train, which can help to prevent the training divergence that is often seen in very large models. This is a trade-off between model capacity and training stability, but it is one that has proven to be effective in practice. The Kimi K2 model, for example, was able to be pre-trained on an astounding **15.5 trillion tokens without a single loss spike**, a feat that would have been nearly impossible without these types of architectural optimizations . This focus on training stability is a key aspect of modern LLM development and is a critical component of any successful long-context model.

3.3. General Efficiency Optimizations

3.3.1. Memory and Compute Optimization (Flash Attention, KV Caching)

Optimizing the memory and compute requirements of Large Language Models is a critical challenge, particularly as models continue to grow in size and complexity. A number of techniques have been developed to address this challenge, with **Flash Attention** and **KV Caching** being two of the most important . Flash Attention is a memory-efficient attention mechanism that reduces the memory complexity of the standard attention mechanism from quadratic to linear with respect to sequence length. This is achieved through a combination of tiling and recomputation techniques, which allow the model to process long sequences without running into memory bottlenecks . This is a crucial optimization for any model that needs to handle long contexts, and it is a key enabler of the impressive performance of models like Kimi K2.

KV Caching is another critical optimization for improving the efficiency of autoregressive generation. During the generation process, the model computes a set of

key and value tensors for each token in the sequence. With KV Caching, these tensors are stored in a cache and reused for subsequent tokens, which avoids the need to recompute them for each new token . This can lead to a dramatic reduction in both computation and memory usage, particularly for long sequences. The combination of Flash Attention and KV Caching is a powerful one-two punch for optimizing the memory and compute requirements of LLMs, and it is a key component of the modern AI toolkit. These techniques are essential for deploying large-scale models in production environments, where cost and latency are major concerns.

3.3.2. Training Optimizations (Data/Pipeline/Tensor Parallelism)

Training Large Language Models at the scale of trillions of parameters requires a sophisticated set of parallelism strategies to distribute the computational and memory load across multiple devices. The three most common approaches are **data parallelism, pipeline parallelism, and tensor parallelism** . Data parallelism is the simplest form of parallelism, where the training data is split across multiple GPUs, each of which holds a full replica of the model. The gradients from each GPU are then averaged to update the model weights. While simple to implement, data parallelism can be inefficient for very large models that do not fit on a single GPU.

Pipeline parallelism and tensor parallelism are more advanced techniques that are designed to address this limitation. Pipeline parallelism splits the model's layers across multiple GPUs, with each GPU processing a different set of layers. This allows for the training of models that are too large to fit on a single GPU, but it can introduce "bubbles" in the training process where some GPUs are idle while waiting for others to complete their computations. Tensor parallelism, on the other hand, splits the individual weight matrices of the model across multiple GPUs. This is a more fine-grained approach to parallelism that can be more efficient than pipeline parallelism, but it also introduces more communication overhead between the GPUs. The most effective training strategies typically involve a combination of these three approaches, with the specific configuration depending on the size of the model and the available hardware. The **ZeRO (Zero Redundancy Optimizer)** technique is another important optimization that can be used in conjunction with data parallelism to reduce the memory footprint of the model by partitioning the optimizer states, gradients, and parameters across the GPUs .

3.3.3. Inference Optimizations (Quantization, Pruning)

Optimizing the inference process of Large Language Models is crucial for deploying them in real-world applications, where cost and latency are major concerns. Two of the most effective techniques for inference optimization are **quantization and pruning** . Quantization involves reducing the precision of the model's weights and activations, for example, from 32-bit floating-point numbers to 8-bit or 4-bit integers. This can lead to a dramatic reduction in both memory usage and computational overhead, which can result in significant improvements in inference speed. There are two main approaches to quantization: post-training quantization (PTQ), which is applied to a pre-trained model, and quantization-aware training (QAT), which is applied during the training process itself. While PTQ is simpler to implement, QAT can often lead to better performance by allowing the model to adapt to the reduced precision.

Pruning is another powerful technique for reducing the size and complexity of a model. It involves removing unnecessary connections and neurons from the neural network, which can lead to a more sparse and efficient model. There are two main types of pruning: structured pruning, which removes entire layers or attention heads, and unstructured pruning, which removes individual weights based on their importance. Structured pruning is generally easier to implement and can lead to more significant speedups, but it can also have a larger impact on the model's performance. Unstructured pruning, on the other hand, can be more fine-grained and can lead to a higher level of compression, but it can also be more challenging to implement efficiently on modern hardware. The combination of quantization and pruning can lead to a dramatic reduction in the size and computational cost of a model, making it more suitable for deployment on resource-constrained devices or in high-throughput environments .

4. The Optimal Path: From Base Model to Frontier Performance

4.1. Selecting the Optimal Base Model

The journey to building a next-frontier small LLM for agentic coding begins with the critical decision of selecting the right base model. This choice lays the groundwork for all subsequent training and optimization efforts. The ideal base model should possess strong foundational capabilities in coding and reasoning, be available in a size that is manageable for further development, and have a well-documented and open-source architecture. Based on the analysis of current state-of-the-art models, the **Qwen 2.5-Coder series** emerges as a highly compelling starting point.

4.1.1. Recommended Starting Point: Qwen 2.5-Coder (7B or 14B)

The **Qwen 2.5-Coder series**, developed by Alibaba's Qwen team, offers a range of model sizes, from 0.5B to 32B parameters, making it an excellent choice for building a small but powerful agentic coding model . The **7B and 14B models**, in particular, strike a good balance between performance and resource requirements. The Qwen2.5-Coder-7B model, for example, has demonstrated strong performance on a variety of coding benchmarks and can be deployed on a single high-end consumer GPU . The 14B model offers even better performance, while still being more manageable than the 32B flagship model. These models have already been pre-trained on a massive corpus of code and general text data, giving them a strong foundation in both coding and general language understanding . Furthermore, the Qwen team has open-sourced both the base and instruct versions of these models, providing a solid foundation for further fine-tuning and customization .

4.1.2. Alternative: Smaller Variants of GLM or DeepSeek Models

While the Qwen 2.5-Coder series presents a compelling option, it is by no means the only path to building a high-performance SLM for agentic coding. Several other open-source models, particularly smaller variants of the **GLM and DeepSeek families**, offer viable alternatives that are worth considering. The **GLM 4.6 model**, for example, has been recognized as a frontier-scale open-source model with a permissive MIT license, making it an attractive choice for developers who prioritize self-hosting and customization . Although the full GLM 4.6 model is a large 355B Mixture-of-Experts (MoE) architecture, the underlying technology and training methodologies can be adapted to create smaller, more efficient versions that are better suited for local deployment. The GLM team's focus on bilingual capabilities and their expertise in handling long-context sequences are particularly relevant for agentic coding applications, where the ability to understand and process large codebases is a significant advantage. The model's strong performance on benchmarks like SWE-Bench, where it has achieved results comparable to top-tier proprietary models, further underscores its potential as a foundation for a powerful agentic coding assistant .

Similarly, the **DeepSeek family** of models, including the DeepSeek-r1:8b model, has demonstrated strong performance on a range of reasoning and technical writing tasks, making it a suitable candidate for agentic coding applications . The DeepSeek team's focus on optimizing their models for structured output and multi-turn interactions is particularly relevant for building agents that can engage in complex, collaborative workflows. The availability of these models through platforms like Ollama also simplifies the process of local deployment and integration into existing development

environments . While the DeepSeek models may not be as specialized for coding as the Qwen 2.5-Coder series, their strong general reasoning capabilities and their ability to handle a wide range of technical tasks make them a versatile and powerful alternative. By fine-tuning a smaller variant of a DeepSeek model on a high-quality dataset of coding problems and solutions, developers can create a highly capable agentic coding assistant that is tailored to their specific needs. The choice between Qwen, GLM, and DeepSeek will ultimately depend on the specific requirements of the application, the available computational resources, and the developer's personal preferences.

4.1.3. Rationale: Strong Coding Priors and Open-Source Availability

The primary rationale for choosing a model from the Qwen 2.5-Coder series is the **strong coding priors** that these models possess. They have been specifically designed and trained for code-related tasks, and their performance on various benchmarks reflects this specialization . This means that a significant amount of the work required to instill coding capabilities has already been done, allowing the developer to focus on the more challenging task of instilling agentic behaviors. The **open-source availability** of these models is another major advantage. It provides full access to the model's architecture and weights, allowing for a deep understanding of its inner workings and enabling the application of advanced fine-tuning and optimization techniques. This level of transparency and control is often not available with proprietary models, making the Qwen 2.5-Coder series a particularly attractive option for researchers and developers who want to build and customize their own agentic coding models.

4.2. Pre-Training Strategy for a Small Model

While the selected base model will already have undergone extensive pre-training, a further phase of focused pre-training on a high-quality, domain-specific corpus can be highly beneficial. This is particularly true for a small model, where every bit of performance gain is crucial. The goal of this phase is to further enhance the model's coding and reasoning capabilities, and to prepare it for the more complex demands of the post-training pipeline.

4.2.1. Dataset Composition: High-Quality, Code-Heavy Corpora

The composition of the pre-training dataset is of paramount importance. To build a model that excels at agentic coding, the dataset should be heavily weighted towards high-quality code. The Qwen team, for their Qwen3-Coder model, used a pre-training corpus of 7.5 trillion tokens, with **70% of it being code** . While a smaller model will not

be able to handle a dataset of this scale, the principle of using a code-heavy corpus remains the same. The dataset should be diverse, covering a wide range of programming languages, coding styles, and software engineering tasks. It should also include a significant amount of data that is relevant to agentic workflows, such as code with extensive documentation, examples of test-driven development, and data that illustrates the use of various development tools and APIs.

4.2.2. Dataset Quality: Leveraging Larger Models for Data Cleaning and Synthesis

The quality of the pre-training data is just as important as its quantity. Noisy or low-quality data can have a detrimental effect on the model's performance. To ensure a high-quality dataset, a common strategy is to use a larger, more capable model to clean and synthesize the data. The Qwen team, for example, used their **Qwen2.5-Coder model** to clean and rewrite noisy data, which was then used to train the larger Qwen3-Coder model . This approach can be adapted for a smaller model by using a state-of-the-art model like Qwen2.5-Coder-32B or GPT-4o to filter, clean, and even generate new training data. This can involve tasks such as removing code with bugs or security vulnerabilities, improving the quality of code comments and documentation, and generating new, high-quality code examples for specific tasks. This "teacher-student" approach to data curation can significantly improve the quality of the final training corpus, leading to a more robust and capable model.

4.2.3. Token Scale: Training on Several Trillion Tokens

The scale of the pre-training dataset is a critical factor in determining the final performance of a language model, even for a "small" model in the context of the current frontier. While the industry has seen a trend towards ever-larger datasets, the focus is shifting towards the quality and efficiency of the data used. The **Kimi K2 model**, for instance, was pre-trained on a massive **15.5 trillion high-quality tokens** . This large-scale training is essential for imbuing the model with the broad general-purpose priors that are necessary for agentic behavior. The sheer volume of data allows the model to learn the intricate patterns and relationships that govern language, code, and the world at large, providing a rich foundation for the more specialized post-training that follows. However, it is not just about the quantity of the data; the quality is equally, if not more, important.

The emphasis on "high-quality" tokens is a key aspect of the modern pre-training strategy. As the availability of high-quality human-generated data becomes increasingly limited, the need to maximize the learning signal per token becomes

paramount. This has led to the development of new techniques for data curation and synthesis, as well as novel optimizers like **Muon**, which is designed to be more token–efficient than traditional methods like AdamW . The goal is to create a pre–training dataset that is not only large but also rich in information and free from noise and biases. This focus on quality over quantity is a more sustainable and effective approach to scaling, as it allows for the creation of powerful models without requiring exponentially larger datasets. For a small model aiming for frontier performance, a pre–training dataset on the order of **several trillion tokens** is a reasonable target, provided that the data is of the highest possible quality. This will ensure that the model has a strong foundation of knowledge and reasoning abilities, which can then be honed through the advanced post–training techniques described earlier.

4.3. Implementing the Post–Training Pipeline

4.3.1. Building the Agentic Data Synthesis Infrastructure

The first step in implementing the post–training pipeline is to build the infrastructure for generating large–scale agentic data. This involves creating a system that can simulate a wide variety of coding tasks and environments, and then use the model to generate a massive number of agentic trajectories. The system should be designed to be highly parallelizable, so that it can generate data at a very high rate. It should also include a mechanism for automatically verifying the correctness of the generated trajectories, so that only high–quality data is used for training. This infrastructure is a critical component of the pipeline, and its design and implementation will have a major impact on the final performance of the model.

4.3.2. Setting up the RLVR Environment (e.g., 20k Parallel Environments)

The next step is to set up the environment for the reinforcement learning with verifiable rewards (RLVR) stage. This involves creating a system that can run a large number of parallel environments, where the model can interact with the world and receive feedback on its actions. For a coding task, this might involve a system that can spin up a large number of Docker containers, each with a different code repository and a set of unit tests. The model would then be tasked with fixing a bug in the repository, and it would receive a reward based on whether the tests pass. The system should be designed to be highly scalable, so that it can support a large number of parallel environments. This is crucial for the RL stage, as it allows the model to explore a wide range of different tasks and to learn from a large number of interactions.

4.3.3. Iterative Refinement with Self-Critique and Human Feedback

The final stage of the post-training pipeline involves an iterative refinement process that combines self-critique mechanisms with human feedback. This is a crucial step for pushing the model's performance to the frontier and ensuring that it is aligned with human values and expectations. The self-critique mechanism, as seen in the Kimi K2 model, allows the model to learn from its own evaluations of its outputs, extending its alignment from static, pre-defined tasks to more open-ended domains . This instills a sense of self-reflection and continuous improvement, which are hallmarks of advanced intelligence. By training the model to assess its own performance, researchers can create a powerful learning loop that enables the model to refine its strategies in real-time.

In addition to self-critique, the iterative refinement process should also incorporate human feedback. This can be done through a variety of methods, such as Reinforcement Learning from Human Feedback (RLHF), where human annotators are used to rank the model's outputs and provide a reward signal. This human-in-the-loop approach is essential for ensuring that the model's behavior is not only effective but also safe, ethical, and aligned with human preferences. The combination of self-critique and human feedback creates a robust and comprehensive refinement process that allows the model to continuously improve its performance and adapt to new challenges. This iterative approach is a key differentiator for top-tier AI labs and is a critical component of any successful post-training pipeline.

5. Benchmarking and Evaluation for Agentic Coding

5.1. Key Benchmarks for Software Engineering

To measure the progress and performance of a next-frontier agentic LLM, it is essential to have a robust set of benchmarks and evaluation metrics. These benchmarks should be designed to test the model's ability to perform a wide range of agentic tasks, from simple code generation to complex, multi-step problem-solving. The evaluation should be as objective and automated as possible, to ensure that the results are reliable and reproducible. The choice of benchmarks is a critical decision, as it will guide the development of the model and provide a clear measure of its success.

表格 复制

Benchmark	Focus Area	Evaluation Method
SWE-Bench Verified	Real-world bug fixing	Resolves a GitHub issue in an oper the project's test suite.
LiveCodeBench	General coding proficiency	Solves competitive programming p
Terminal-Bench	Command-line and tool usage	Executes a series of commands in complete a task.

Table 2: Key Benchmarks for Evaluating Agentic Coding Capabilities.

5.1.1. SWE-Bench Verified for Real-World Bug Fixing

SWE-Bench Verified is a challenging benchmark that is designed to evaluate a model's ability to fix real-world bugs in open-source Python repositories . The benchmark consists of a set of GitHub issues and their corresponding pull requests. The model is given the issue description and the codebase, and it is tasked with generating a patch that fixes the bug. The patch is then evaluated by running the project's test suite. This is a very realistic and challenging task, as it requires the model to understand the codebase, to identify the root cause of the bug, and to generate a correct and robust fix. The **Kimi-Dev-72B model**, for example, achieved a score of **65.8%** on this benchmark, demonstrating the power of the RLVR approach .

5.1.2. LiveCodeBench for General Coding Proficiency

LiveCodeBench is a benchmark that is designed to evaluate a model's general coding proficiency. It consists of a set of competitive programming problems, similar to those found on platforms like LeetCode and Codeforces. The model is given a problem description and a set of hidden test cases, and it is tasked with generating a solution that passes all of the tests. This benchmark is a good measure of the model's ability to understand complex problem statements, to design and implement efficient algorithms, and to write correct and robust code. The **MiniMax M2 model**, for example, achieved a score of around **83%** on this benchmark, which is on par with other leading models like GLM-4.6 .

5.1.3. Terminal-Bench for Command-Line and Tool Usage

Terminal-Bench is a benchmark that is designed to evaluate a model's ability to use command-line tools and interact with a terminal environment. It consists of a set of

tasks that require the model to execute a series of commands to complete a specific goal. For example, a task might require the model to use `grep` and `awk` to analyze a log file and generate a summary report. This benchmark is a good measure of the model's ability to understand and use a variety of command-line tools, and to reason about the output of these tools to inform its next action. This is a critical capability for any agentic coding assistant, as it allows the model to interact with the development environment in a flexible and powerful way.

5.2. Evaluating Agentic Capabilities

5.2.1. Assessing Multi-Turn Reasoning and Planning

A key aspect of agentic behavior is the ability to perform multi-turn reasoning and planning. This involves breaking down a complex task into a series of smaller, manageable steps, and then executing these steps in a logical and coherent sequence. To evaluate this capability, it is important to use benchmarks that require the model to engage in a long and complex interaction with its environment. This could involve a task that requires the model to use multiple tools in a specific order, or a task that requires the model to adapt its plan in response to new information. The ability to perform multi-turn reasoning and planning is a critical differentiator between a simple code generator and a true agentic coding assistant.

5.2.2. Measuring Tool Use and API Interaction

Another critical aspect of agentic behavior is the ability to use a variety of tools and APIs to accomplish a task. This includes not only the ability to call the tools with the correct parameters, but also the ability to understand the output of the tools and to use this information to inform the next step in the workflow. To evaluate this capability, it is important to use benchmarks that require the model to interact with a diverse set of tools and APIs. This could include tools for code search, code editing, testing, and deployment. The ability to effectively use a wide range of tools is a key enabler of the sophisticated problem-solving capabilities that are required for true agentic intelligence.

5.2.3. Analyzing Performance on Long-Horizon Tasks

Finally, it is important to evaluate the model's performance on long-horizon tasks. These are tasks that require the model to maintain a coherent understanding of a complex goal over a long period of time, and to take a series of actions that are all aligned with this goal. This is a particularly challenging aspect of agentic behavior, as it

requires the model to have a strong sense of long-term planning and to be able to avoid getting sidetracked by short-term distractions. To evaluate this capability, it is important to use benchmarks that require the model to complete a complex, multi-step project, such as building a small application from scratch. The ability to perform well on long-horizon tasks is a key indicator of a model's true agentic capabilities.