ChatGPT

Building a Next-Frontier Small LLM for Agentic Coding (Late 2025)

Developing a cutting-edge *small* Large Language Model (LLM) for coding assistance requires careful planning from the ground up – starting with the right base model and data, through specialized training for code and agent-like behavior, to alignment and efficient deployment. Below we outline **the optimal path**, based on the latest techniques circa November 2025 and practices used by top AI labs, to build an agentic coding assistant (comparable to models like MiniMax M2 or GLM-4.6).

Choosing the Base Model: Fine-Tune vs. From-Scratch Training

Leverage an open high-quality base model rather than training from scratch. In 2025, the open-source model ecosystem is rich – Meta's LLaMA 3.x series, Mistral and Qwen models, Zhipu's GLM-4.6, etc., offer powerful pretrained foundations. Fine-tuning one of these bases on coding tasks is generally *far more efficient* and effective than attempting to pretrain a brand new model from scratch on code data. Top-tier labs have shown that starting from a strong base and specializing it yields excellent results without the enormous cost of full pretraining [1] [2]. For example, Code Llama was created by **further training Llama-2 on code-specific data**, rather than training a new model from nothing [1]. Likewise, the DeepSeek project distilled a large "reasoning" model into smaller *Llama 3* and *Qwen* models, demonstrating that *small models can inherit the capabilities of larger ones* when fine-tuned appropriately [2]. In short, **fine-tuning a proven open model is the optimal path** given the availability of high-quality bases and datasets in late 2025.

Selecting a base model: For an *agentic coding* assistant (able to write code, debug, and use tools autonomously), a model in the range of ~7B to 34B dense parameters (or an equivalent sparse/MoE configuration) is a good starting point. Models in this size range can capture complex coding knowledge while remaining feasible to train and deploy on cloud GPUs. Notably, many current open models have versions specialized for coding: e.g. *LLaMA 3-Code* (hypothetical successor to Code Llama), **GLM-4.6** by Zhipu (357B total but open weights) which has strong coding capabilities [3], or *Qwen-14B* by Alibaba. If available, a model that has been pretrained on both text and code (for general reasoning plus coding) gives a head-start.

Dense vs. Mixture-of-Experts (MoE): Top labs in 2025 have pioneered MoE architectures to get "small LLM" efficiency with large-model performance. For instance, **MiniMax-M2** uses a Transformer MoE with 230B total parameters but only ~10B active at once [4]. This means it "behaves like a giant model when needed" but keeps inference cost closer to a 10B model [4] [5]. Similarly, **Kimi K2** from Moonshot AI is a 1 trillion-parameter MoE with 32B activated per token [6]. These architectures can dramatically boost capability at fixed compute cost by routing tasks to specialized expert subnetworks. If you have the engineering expertise and moderate distributed compute, adopting an MoE base can yield a *frontier-quality* coding model that remains lightweight to run. MiniMax-M2, for example, achieves performance near OpenAI's latest GPT models on coding benchmarks while "activating one-twentieth of the parameters" at inference [7]. That said, MoE training is more complex. For most teams focused on **open-source and compute-**

efficient development, starting with a strong *dense* model (e.g. 13B or 34B) and optimizing it is a more straightforward route. You can always incorporate MoE ideas later or use distilled versions of those MoE models (some projects release dense distilled checkpoints of MoE teachers [8]).

Pretraining Data: Leveraging Code Corpora and Software Knowledge

A coding model lives or dies by the quality of its training data. State-of-the-art code LLMs are trained on massive corpora of source code and related text:

- **Open code repositories:** Projects like *The Stack* (by Hugging Face's BigCode initiative) provide hundreds of gigabytes of permissively licensed code. For example, The Stack v2 contains *over 3 billion code files across 600+ programming languages* [9] – a trove of open-source code from GitHub. This serves as a primary pretraining corpus for many code-specialized models. Diverse language coverage is useful even if your assistant will primarily code in, say, Python – multi-language training teaches the model different paradigms and can strengthen core coding abilities (and you never know when a bit of Bash or SQL might be needed in a software engineering context).

- **Programming Q&A and documentation:** In addition to raw code, it's valuable to include data that teaches the model *how humans discuss and use code*. Datasets from Stack Overflow (Q&A pairs), GitHub Issues and commit messages, and API documentation can be incorporated. For instance, StarCoder's training data included ~54GB of GitHub issues discussions and ~13GB of Jupyter notebooks [10]. Such data helps the model learn to explain code, describe errors, and follow instructions – critical for an interactive coding assistant.

- **Project-level context:** To build an agent capable of larger software engineering tasks, the model should see not only single-file snippets but also *multi-file projects and build/test outputs*. Including entire repositories or segments of them, along with their README documentation and test cases, can teach the model about project structure. Some advanced benchmarks (like "SWE-Bench" or "TerminalBench" introduced in 2025) specifically evaluate a model's ability to handle multi-file refactoring or use a build system [11]. Training on such data (where available) will prepare the LLM for realistic developer workflows.

If you were training a model *from scratch*, you'd mix a large volume of code (potentially trillions of tokens) with general text. (In Kimi K2's case, they reportedly pre-trained on **15.5 trillion tokens** combined data with a specialized optimizer to avoid instabilities [12].) However, following our recommendation to fine-tune an existing model, you likely don't need to replicate a full pretrain. Instead, **perform a focused *continued-pretraining* on code**: start from the chosen base model and train further on a curated code corpus. This is exactly how Meta created **Code Llama** – by taking a LLaMA-2 model and continuing its training on code-specific datasets [1]. Continued pretraining infuses the model with coding fluency (syntax, libraries, common patterns) beyond what the base may have. When doing this, emphasize quality: filter out low-quality code (e.g. trivial snippets, boilerplate) and potentially deduplicate to avoid overfitting common files. Many top labs use strict data filtering and might *up-sample* high-value subsets (like well-documented code or problem-solving code from competitive programming) to teach best practices.

Model Size and Architecture Considerations

How big should the model be? "Small LLM" is a relative term – here we mean something that can be deployed on accessible hardware (a handful of GPUs) while still pushing the frontier of coding capability. In practice, **top-performing coding assistants in 2025 are often in the 10B–30B parameter range for dense models**, or use *sparse* techniques to reach effectively larger sizes. A 13B dense model that's been expertly fine-tuned can outperform a poorly trained 30B model, so quality beats raw size. For instance, Mistral AI's 7B and 16B models have proven surprisingly strong after fine-tuning, in some cases rivaling older 30B models [13]. That said, coding tasks (which involve precise logic and knowledge) do benefit from more parameters up to a point – Meta found that Code Llama 34B and 70B "return the best results" on coding benchmarks, with 7B/13B trailing in quality (though faster) [14]. Thus, if resources allow, leaning toward the higher end of our range (e.g. a ~30B model) for the final product is wise to maximize performance as an autonomous coding agent [14].

If using or designing a Mixture-of-Experts architecture, focus on the *activated parameter count* as the measure of "effective" model size. MiniMax-M2's design philosophy was to cap active size at ~10B to keep latency low, and it reaped benefits in speed and multi-agent throughput by doing so [15]. An MoE with 10B active and, say, 200B total (with many experts) can be a sweet spot if you have moderate scale: it's *cheap to query* like a 10B model, but during training it has the capacity of 200B worth of parameters to learn diverse skills [4] [5]. The "10B rule" MiniMax follows is not arbitrary – it was chosen to ensure fast feedback loops during code execution cycles and to allow running many agent instances concurrently without blowing up server costs [15]. In summary, choose the largest model you can *efficiently* train and serve, erring on the side of a **well-trained mid-size model** over an under-trained giant. Modern training techniques (discussed next) can then be applied to squeeze maximum capability out of that model.

Specialized Fine-Tuning for Coding Tasks

After (or in parallel with) general code pretraining, you will perform *supervised fine-tuning (SFT)* on instruction-following and problem-solving tasks to shape the model into a helpful coding assistant. This stage teaches the LLM *how to use its knowledge in an interactive setting* – writing functions given a description, answering questions about code, reviewing or refactoring code, etc., in plain English dialogue.

Key fine-tuning data sources and techniques:

- **Instruction datasets for coding:** Assemble high-quality examples of the types of interactions you expect. This can include: "user asks for a code snippet accomplishing X, assistant provides code and explanation," bug-fix dialogues (user presents code with a bug, assistant finds and fixes it), code review comments, explaining a piece of code, etc. Some open datasets exist (e.g. the Stack Exchange programming QA corpus, or synthetic instruct data generated from code solutions). In 2025, many organizations also leverage *GPT-4/GPT-5* to generate instruct data: for example, take a coding problem and have GPT produce an ideal step-by-step solution with explanations, then use that as a training sample (a form of **AI-assisted dataset generation**). Be cautious to review or filter such synthetic data for correctness.

- **Multi-turn conversations and context:** Ensure the fine-tune data includes multi-turn exchanges, not just single prompt-response pairs. A coding agent should handle clarifications or iterative

improvements. For instance, a conversation might start with *"Draft a function to do X."* Model outputs code. User then says *"Actually, make it more efficient."* Model refines the code. Training on such dialogues teaches adaptation to feedback. Real chat logs with coding assistants (if available) are great, otherwise you can simulate them.

- **Diversity of tasks:** Include not only straightforward coding but also higher-level software engineering tasks if possible. E.g., *architectural reasoning* (user asks how to design a system or class structure), *reading and understanding existing code* (user provides code and asks what it does or to document it), and *code parsing* (like interpreting error messages or logs). If you want the model to handle documentation or design questions (as you indicated "whichever is best" between strictly coding vs. some non-coding tasks), include data for those too. Top labs often fine-tune on a mix of domains even for a coding model, to keep it well-rounded. Given our focus is strictly English, all instruction data and discussions should be in English (even if code elements are present).

- **Quality over quantity:** It's better to have a smaller set of *expert-written or high-fidelity* examples than a huge noisy dump. A few thousand excellent coding Q&A pairs can significantly improve performance. For example, OpenAI's early Codex work distilled knowledge by fine-tuning on a curated set of problem-solution pairs (like from competitive programming) and saw big jumps in correctness. Similarly, the *WizardCoder* series (open fine-tunes in 2023) took a base model and fine-tuned on high-quality coding instructions, yielding state-of-the-art results for relatively little data because the data was targeted and clean.

Incorporating *Agentic* Behavior and Tool Use

What sets the "next frontier" coding assistant apart is **agentic capability** – the model can act like an autonomous developer: not just writing code when asked, but *proactively executing code, observing results, searching for information, and iterating* to achieve a goal. Building such behavior requires going beyond standard next-token prediction on static data; it involves training the model in a way that includes *tool interactions and self-reflection*.

Techniques to instill agentic behavior:

- **Interleaving reasoning and actions:** One effective approach, showcased by models like Kimi K2 and MiniMax M2, is to train the model to produce outputs in a special format that separates "thinking" from "acting." For instance, MiniMax-M2 wraps its internal chain-of-thought in special `<think>...</think>` tags during a reasoning process [16]. The model is fine-tuned to generate these tags and keep a running log of its thought process. Crucially, the *content inside* `<think>` *tags is fed back into the model in the next turn*, so it has a persistent memory of its reasoning [17]. This allows the LLM to plan and reflect step-by-step, which is vital for complex, multi-step tasks. **Kimi K2** uses a similar idea called *Interleaved Reasoning*, where the model's cycle is: **Plan → Act → Verify → Reflect → Refine**, looping as needed [18]. During training, K2 was likely given sequences that include these phases explicitly. By teaching the model a template for these steps, it learns to catch and correct mistakes in an autonomous loop rather than requiring human prompts at each step [19].

- **Tool-use APIs in training:** To make the model truly agentic, include examples where the model calls external tools. In a coding context, the primary "tool" is a **code execution environment** (a Python

REPL, a compiler, etc.). You can represent tool use in the training data by special tokens or markup. For example, a training instance might show:
User: *"What does this snippet print?"*
Assistant: `<tool> python\nprint(2+3)\n</tool>` (meaning the assistant "executes" the code) then gets back the result, and continues: *"The snippet prints 5."*
By including formatted dialogues like this, the model learns that it can solve problems by *executing code* when appropriate. This is how a framework like ReAct (Reason+Act) works, and it's been very influential in 2024–2025 for LLM training. In fact, **Moonshot's Kimi K2 was trained with a large-scale *agentic data synthesis* pipeline and a joint *reinforcement learning stage* where the model practiced in** real or simulated tool-use environments** [20]. That means they generated many example scenarios (possibly using the model itself or other models) requiring tool use and had K2 learn through trial and error (RL) to solve them. You can emulate this by generating synthetic training data: e.g. have a script that creates coding tasks and their solution transcripts including the model taking actions (like running code, web searches for documentation, etc.), then fine-tune on those transcripts.

- **Error handling and self-correction:** A hallmark of agentic coding models is the ability to recover from mistakes – just as a human developer debugs errors. Encourage this by training on **"compile-run-fix" loops**. For instance, show a scenario where the assistant writes some code, then receives an error message (as if from a compiler/test). The next model response should analyze that error and correct the code. MiniMax-M2 was explicitly tuned for *compile–run–fix cycles*, enabling it to iterate on code until it passes tests [11]. It even excels at benchmarks that require such iterative repairs, suggesting its training data contained many debug scenarios. Similarly, have Q&A where the model's initial answer is wrong, and then it "notices" a failing test or a contradiction and fixes its answer. Training the model with these feedback signals (even if we have to simulate them) will make it much more robust and "agent-like." Research has shown that models can learn to use *self-reflection and tool feedback* effectively: for example, a recent approach let small models use tools to check their outputs, significantly improving reasoning accuracy [21] [22]. In coding, this translates to using a compiler or runtime to verify outputs and then adjusting accordingly.

- **Multi-modal inputs (optional):** In some setups, an agent might need to read PDFs or images (for design diagrams) or interact with a GUI. If your use case foresees that (perhaps less likely strictly for coding tasks), you'd include data or adapters for those modalities. However, for now, focusing on text (code and docs) and possibly web content is usually sufficient.

During fine-tuning, you might construct **prompt-response trajectories** that mimic an agent's full workflow: from reading a task, planning the approach, writing code, running it, encountering an error, fixing it, to final solution. By treating the whole sequence as one training example (or a series of examples), the model learns not just to produce correct code in one shot, but to *engage in the process* a developer would. As Kimi K2's success shows, this yields a model that doesn't easily get stuck – it can make 200+ tool calls in one session, each time *verifying and refining* its approach without losing track [23]. The extra effort in crafting these agentic training scenarios pays off with a model that is far more reliable on complex tasks.

Comparison of Kimi K2 (blue) against OpenAI's GPT-5.1 (light blue) and an Anthropic Claude 4.5 model (gray) on various agentic reasoning and coding benchmarks in 2025. Notably, K2 performs competitively on coding tasks (e.g. SWE-Bench and LiveCodeBench) despite a fraction of the inference cost, thanks to its interleaved reasoning and tool-use training. [24] [25]

Alignment and Post-Training Refinement (RLHF, DPO, etc.)

After the model has been supervised-fine-tuned on code and agentic behaviors, top labs apply an **alignment tuning** phase. The goal here is to ensure the model's outputs are not only *correct*, but also *helpful, safe, and in line with user preferences*. In coding assistants, this translates to the model following user instructions closely (e.g. coding in the style requested, or not revealing confidential info), avoiding inappropriate content, and perhaps adhering to company-specific guidelines (if any).

Two main strategies are used for alignment in 2025:

- **Reinforcement Learning from Human Feedback (RLHF):** RLHF has become a standard recipe for aligning LLMs [26]. The process involves gathering a bunch of model responses, having humans (or proxies) rank them, training a reward model, and then fine-tuning the policy (the LLM) to prefer responses with higher reward (often via proximal policy optimization or similar RL algorithms). In a coding context, human evaluators might rank answers by correctness, clarity, and usefulness. For example, given a prompt "Explain this code bug," a response that accurately pinpoints the bug and fix would be rated above one that is vague or incorrect. OpenAI's ChatGPT and Anthropic's Claude were aligned with human feedback to be more helpful and truthful; similar techniques apply to coding assistants (though the "human" feedback could also include checking for code correctness). One noteworthy point: some research (like DeepSeek-R1) found that *even without an initial supervised phase*, a large model can be directly trained with RL on reasoning tasks to discover useful behaviors [27]. DeepSeek-R1-Zero emerged with abilities like *self-verification and long chain-of-thought generation* purely by optimizing a reward (likely correctness) in an RL loop [27]. However, they also noted this came with downsides (repetitions, etc.), so in practice a combination of SFT and RLHF was used [28]. For your project, RLHF could be used to specifically reward outcomes like **code that passes all tests** or **accurate explanations**. You could set up an automated proxy: run the model's code answer on a suite of hidden test cases and use that as a reward signal to refine the model (a form of *execution-based reward*). This was a strategy reportedly used to improve Codex and other code models – rewarding solutions that actually work and penalizing those that error or hallucinate. Over many examples, the model learns to produce more correct code. There is also the **Constitutional AI** approach (Anthropic's method) where AI feedback is used to refine the model along given principles (e.g. "don't produce insecure code unless explicitly asked"). By 2025, it's common to use a mix of human and AI-generated feedback for scalability.

- **Direct Preference Optimization (DPO):** A newer alignment method gaining popularity is DPO, which forgoes the traditional RL loop in favor of a simpler objective on preference data. In DPO, you take pairs of model outputs (one preferred, one not) and directly fine-tune the model to score the preferred one higher – effectively doing a form of logistic regression to push the model toward preferred outputs [29]. The advantage is that **DPO doesn't require training a separate reward model and is more stable and lightweight than RLHF, yet achieves comparable results** [30]. This can be very appealing if you have logged user preference data or some comparisons (perhaps from beta testers choosing which code answer is better). Microsoft's Azure team notes that DPO is *computationally cheaper and faster* than RLHF for aligning models to human preferences [30]. In an agentic coding scenario, you might collect comparisons like "Solution A vs Solution B – which is more correct/efficient?" and use DPO to tune the model. Many open-source projects have started adopting DPO because it sidesteps some complexities of RL (no reward model overfitting, no delicate PPO

hyperparameters). If your budget or expertise in RL is limited, DPO could be the **best alignment technique** to use, provided you have or can generate a decent set of preference data.

In practice at top labs, the alignment phase can involve **multiple stages**. For example, Kimi K2's creators ran a *joint RL stage* after their agentic fine-tuning, likely to further improve the model's performance on interactive tasks and align it with desired outcomes [20]. DeepSeek-R1's pipeline included *two RL stages (one to boost reasoning, one to align with human preferences)* and intermediate supervised calibrations [28]. This multi-step approach indicates that a model may be first taught *what* to do (e.g. solve problems stepwise), and then taught *how* to present the solution in a helpful manner.

Aside from RLHF/DPO, also consider **model self-refinement** techniques. One interesting 2025 development is letting the model refine its own outputs offline. For instance, you generate an answer with your fine-tuned model, then have the model (or a larger one) critique that answer, and fine-tune on the improved version. This "self-correction" loop can be repeated. Tools like the **CRITIC framework** allow an LLM to use external tools to evaluate its output and then improve it, effectively automating feedback incorporation [31] [32]. In code, a simplified version could be: have the model generate code for a problem, run it against tests, then append the test results and the correct answer to the training data if it was wrong – encouraging the model to get it right next time. A survey of self-correcting LLMs noted that training-time feedback (like providing the model with the results of executing its code and letting it update) significantly reduces hallucinations and errors [33]. All these refinements contribute to a more **robust and aligned model** that behaves closer to how a skilled human engineer would: double-checking their work and responding to feedback.

Model Compression and Efficient Deployment

Finally, after training and alignment, you want to deploy this model cost-effectively on cloud infrastructure. Even a "small" 13B or 30B model can be expensive to serve at scale, so top labs employ a few techniques to optimize inference:

- **Quantization:** Converting model weights to lower precision (8-bit, 4-bit, or even hybrid 8-bit/16-bit) can dramatically reduce memory and increase throughput with minimal loss in accuracy. Many of 2025's open models support 8-bit inference out-of-the-box. MiniMax-M2, for example, runs "comfortably" at FP8 or BF16 precision [34]. Using an efficient inference engine (like FasterTransformer, vLLM, or DeepSpeed-Inference) with quantization allows you to serve the model with lower latency and cost. Since you are targeting cloud GPUs, you can utilize tensor-core friendly formats (FP16/BF16/INT8). It's common to fine-tune the model in 16-bit then quantize the final weights for serving. Some frameworks also support quantization-aware training if you want to squeeze even more performance out.

- **Distillation:** We discussed earlier how large models can be distilled into smaller ones. If ultimate efficiency is needed, you might take your fine-tuned model and use it as a "teacher" to train an even smaller student model (e.g. distill a 13B down to 7B) while trying to preserve its behavior. DeepSeek open-sourced several **distilled models (1.5B, 7B, etc.) that retained surprising reasoning and coding ability learned from the 37B DeepSeek-R1 teacher** [8]. By distilling the agentic behaviors and knowledge, they achieved *better performance in small models than if those small models were trained on their own*. This is a powerful result: it means your final deploy model could be much smaller (for speed/cost reasons) without giving up too much capability – as long as you have a larger

model to learn from. If you don't have resources to train a larger model yourself, consider leveraging an existing one as the teacher. For instance, use outputs from a model like Kimi K2 (which is open-access) on a variety of prompts to create a training set for your 13B model to mimic. This kind of *knowledge distillation from a frontier model* can inject some of that frontier prowess into your smaller model [35].

- **Optimized architecture for serving:** Since you plan cloud GPU deployment, you can take advantage of optimized architectures like transformer inference with continuous batching (as in the *vLLM* library, which MiniMax-M2 supports [36]). These systems keep the GPU fed with work and can handle many parallel requests efficiently. Also, if using MoE, ensure your serving stack supports the gating and expert parallelism (there are MoE-serving frameworks given the popularity of models like GLaM, Switch, etc., and by 2025 this is more mature).

- **Long context and memory:** Depending on use case, you might want long context (to feed in entire code files or multiple files). If so, pick architectures that support it (GLM-4.6, for example, expanded context window to 200K tokens [37]). Handling long codebases agentically could require the model to read tens of thousands of tokens (e.g., entire project files) – in such cases, using an RMT (Retrieval-augmented Transformer) or a *Recurrent GPT* that can process streams might be necessary. Alternatively, you can equip the agent with retrieval: have it ingest documents as needed rather than via massive context. This falls under agent design – not model training – but keep in mind how you'll manage large code inputs when deploying the system.

- **Evaluation & iteration:** Finally, deploy with monitoring. Evaluate the model on standard coding benchmarks and your own test scenarios regularly. Benchmarks like **HumanEval**, **MBPP** (Python problems), and the newer **SWE-Bench** and **ArtifactsBench** (which measure handling of real-world coding tasks) should be used to verify performance. For example, MiniMax-M2 was shown to score **65.8 on SWE-Bench (Verified)**, close to GPT-5's 74.9, and even outperformed some closed models on certain agentic tasks [38] [7]. Knowing where your model stands will help prioritize further fine-tuning or data augmentation. Top labs often have a continuous evaluation pipeline; you can do a lighter-weight version with a set of unit tests or sample tasks that the agent should manage (like creating a small app, fixing a known bug, etc.). If the model fails in specific ways (e.g. always forgets to close files or struggles with a particular algorithm), you can address those by adding examples or applying another round of fine-tuning focused on failure cases.

Conclusion: The Optimal Path Forward

To summarize, **the best approach** to build a state-of-the-art small LLM for agentic coding in late 2025 is:

1. **Choose a robust open base model** (e.g. LLaMA 3 or similar 13B–34B model, or an MoE like MiniMax M2 if available) instead of training from scratch – this leverages existing knowledge and saves enormous compute [1] [2]. Fine-tune this base rather than reinventing the wheel.

2. **Specialize the model on code** using massive, high-quality code datasets (The Stack v2, etc.) and further *continue pretraining* on code. This gives the model strong coding fundamentals and familiarity with libraries/frameworks [1].

3. **Fine-tune on instruction and agentic data** specific to software engineering: include coding Q&A, debugging sessions, multi-step planning with tool use, and self-correction examples. Teach the model to *think aloud* (chain-of-thought) and to use tools (like a compiler or web search) within its responses [18] [16]. This will imbue agent-like behavior seen in top models like K2 and M2.

4. **Apply alignment techniques** to ensure the model's outputs are helpful and correct. Use RLHF if you have human ratings, or DPO for a simpler yet effective route leveraging preference pairs [30]. Emphasize feedback based on code correctness and user satisfaction. The model should not only solve problems but do so in a way that a human user would find useful (clear explanations, following instructions on style, etc.).

5. **Incorporate self-play and refinement**: let the model practice on generated tasks (and perhaps critique or refine its own solutions) to further improve its reasoning. For cutting-edge performance, consider a final reinforcement phase where the model *interacts with an environment* (running code, accessing a documentation tool) to organically improve its capability, much like Kimi K2's joint RL stage [20].

6. **Optimize and compress for deployment**: use 8-bit quantization or distill the model down to a smaller size to meet your latency/cost targets. Leverage the findings that a well-trained larger model can be compressed into a smaller one with minimal loss [8]. This way, you benefit from high-end training and still deploy a "small" model that is fast on cloud GPUs.

By following this path – **starting with a strong foundation, rigorously training on coding and agentic skills, aligning with human feedback, and deploying efficiently** – you will build a coding AI that operates at the frontier of what "small" LLMs can do. This approach is grounded in first principles (maximize data and compute utilization where it matters, inject feedback loops for learning) and is validated by the practices of leading AI research labs in 2025. The end result should be a capable, autonomous coding assistant that can plan, code, debug, and reason about software much like a human engineer, all within the compute budget of a relatively small model.

Sources: The above recommendations are supported by extensive research and industry reports, including open model releases and papers from 2025. For example, Moonshot AI's **Kimi K2** employed interleaved reasoning and tool use to achieve state-of-the-art agentic performance [18] [6], while MiniMax's **M2** demonstrated the efficiency of a 10B-parameter MoE for coding tasks [4] [11]. The **DeepSeek-R1** project showed how multi-stage RL training and distillation can produce a small model matching larger proprietary models on reasoning and code benchmarks [39] [2]. These and other references have been cited throughout to provide concrete evidence and guide the construction of your next-frontier coding LLM. Good luck with building it – with the right techniques, even a "small" model can reach great heights! [2] [20]

---

1 14 Introducing Code Llama, a state-of-the-art large language model for ...
https://ai.meta.com/blog/code-llama-large-language-model-coding/

2 8 27 28 35 39 deepseek-ai/DeepSeek-R1 · Hugging Face
https://huggingface.co/deepseek-ai/DeepSeek-R1

3 GLM-4.6 Review - Medium
https://medium.com/@leucopsis/glm-4-6-review-0600e9425c73

4 5 7 11 15 16 17 34 36 MiniMax-M2 : Best model for Coding and Agentic | by Mehul Gupta | Data Science in Your Pocket | Oct, 2025 | Medium
https://medium.com/data-science-in-your-pocket/minimax-m2-best-model-for-coding-and-agentic-2f0a80d4ef7f

6 18 19 23 24 25 The biggest model update this week wasn't GPT-5.1, it was Kimi K2: AI Update #3
https://www.news.aakashg.com/p/the-biggest-model-update-this-week

9 bigcode/the-stack-v2 · Datasets at Hugging Face
https://huggingface.co/datasets/bigcode/the-stack-v2

10 bigcode/starcoderdata · Datasets at Hugging Face
https://huggingface.co/datasets/bigcode/starcoderdata

12 20 38 [2507.20534] Kimi K2: Open Agentic Intelligence
https://arxiv.org/abs/2507.20534

13 The Top 10 Open Source LLMs: 2025 Edition - Scribble Data
https://www.scribbledata.io/blog/the-top-10-open-source-llms-2024-edition/

21 22 Distilling LLM Agent into Small Models with Retrieval and Code Tools
https://arxiv.org/html/2505.17612v2

26 Training Overview | RLHF Book by Nathan Lambert
https://rlhfbook.com/c/04-optimization

29 Direct Preference Optimization (DPO) for LLM Alignment coded from ...
https://www.reddit.com/r/ArtificialInteligence/comments/1ekphvl/direct_preference_optimization_dpo_for_llm/

30 Direct preference optimization - Azure OpenAI | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/fine-tuning-direct-preference-optimization?view=foundry-classic

31 [PDF] CRITIC: Large Language Models Can Self-Correct with Tool ...
https://www.semanticscholar.org/paper/CRITIC%3A-Large-Language-Models-Can-Self-Correct-with-Gou-Shao/bcdaf6c98ddbd6809cf6241aa77200d7394db163

32 (PDF) CRITIC: Large Language Models Can Self-Correct with Tool ...
https://www.researchgate.net/publication/370938047_CRITIC_Large_Language_Models_Can_Self-Correct_with_Tool-Interactive_Critiquing

33 Performance and interpretability analysis of code generation large ...
https://www.sciencedirect.com/science/article/pii/S0925231225021332

37 glm-4.6 - Ollama
https://ollama.com/library/glm-4.6