LLM-Assisted Plasma Simulation Code Automation: Draft 2

Kentaro Seki
Kyoto University
seki.kentaro.66s@st.kyoto-u.ac.jp

January 6, 2026

Abstract

Contents

1 Introduction

1.1 Motivation

Modern nuclear-fusion research relies heavily on **gyrokinetic simulation**, which reduces the full 6D kinetic plasma problem into a more tractable form while preserving key microturbulence physics. This shift enabled "first-principles" turbulence studies and transport predictions that are difficult to obtain experimentally, and it is now routine to use large gyrokinetic codes for tokamak/stellarator turbulence and validation studies (e.g., GENE, GS2, GTC, GKV and related validation work) [28].

However, **gyrokinetic simulation software is hard to extend**:

- **Large, interdependent codebases:** physics modules (e.g., collisions, electromagnetic terms, geometry, diagnostics) are spread across many files and abstraction layers.
- **High "cost per new feature":** adding a seemingly small physics change (example: a new collision operator term, a new equilibrium interface, or a new diagnostic) often forces edits in multiple places: data structures, solvers, I/O, tests, and documentation.
- **High onboarding cost:** new contributors must learn both the physics and the code architecture before making safe changes.

Large Language Models (LLMs) are becoming practical tools for software development and scientific workflows, especially when paired with **external knowledge, tools, and structured agent pipelines**. This motivates building an LLM system that helps fusion researchers (1) understand unfamiliar parts of a codebase faster, and (2) implement changes more reliably—ultimately accelerating iteration cycles for simulation studies.

1.2 Current Development on Large Language Models

The last several years produced a "stack" of advances that (together) explain why modern LLM systems can help with non-trivial scientific coding.

1.2.1 (1) Transformers

Most modern LLMs are based on the **Transformer**, which replaces recurrence with **self-attention**. Intuitively:

- In an RNN, information must flow step-by-step through time.
- In a Transformer, each token can "look at" other tokens directly (through attention), which enables **parallel training** and better long-range dependency handling.

This architecture made it feasible to scale training to very large datasets and models while keeping optimization stable [1].

1.2.2 (2) Scaling laws (and compute-optimal training)

Scaling law work observed that model loss often follows **smooth power-law trends** as you increase model size, data size, and compute—meaning progress can be predicted and engineered [2]. Later work showed that, under a fixed compute budget, many LLMs were "undertrained" on too few tokens, motivating **compute-optimal** training strategies (often summarized via the "Chinchilla" perspective) [3].

Why this matters for scientific coding: scaling isn't just "bigger is better"—it also teaches how to spend compute efficiently, which becomes important when we later discuss RL-based post-training costs.

1.2.3 (3) InstructGPT / RLHF (often PPO-based)

Base LLMs are trained to predict the next token, not to follow instructions. InstructGPT-style pipelines added a practical recipe:

1. Collect **human-written demonstrations** (supervised fine-tuning).
2. Collect **human preference rankings** over model outputs.
3. Train a **reward model** to predict those preferences.
4. Optimize the policy using RL (often **PPO**) to improve reward while limiting drift from the base model.

This line of work improved instruction-following and user preference alignment—even showing cases where a much smaller aligned model is preferred to a much larger base model [5]. The broader "learning from preferences" setup is also well-established in RL [7].

1.2.4 (4) DPO (Direct Preference Optimization)

DPO reframes preference optimization as a **simple classification-like objective** that can match RLHF-style goals (reward maximization with KL constraint) without running full RL rollouts. Practically, that means:

- No explicit reward model is required at training time (it's "implicit").
- Training looks closer to supervised fine-tuning, often simplifying stability and reducing engineering overhead.

This matters for us because we want a training loop that is realistic for academic GPU budgets [10].

1.2.5 (5) DeepSeek-R1-style reasoning-focused post-training

Recent reasoning-focused model releases and reports (e.g., DeepSeek-R1) highlight renewed emphasis on **reinforcement learning / preference optimization** as a path to stronger reasoning behavior, often combined with careful data and evaluation design [35].

For our paper's narrative, the key point is not "one model is best," but that the field trend increasingly treats **post-training (preferences, RL, or RL-free variants like DPO)** as a major lever for reasoning and reliability.

1.2.6 (6) Test-time compute (inference-time search)

Even without changing model weights, you can often improve reasoning by spending **more computation at inference time**. Common patterns:

- **Chain-of-thought prompting**: ask the model to write intermediate steps [11].
- **Self-consistency**: sample multiple reasoning traces and pick the majority-consistent answer [12].

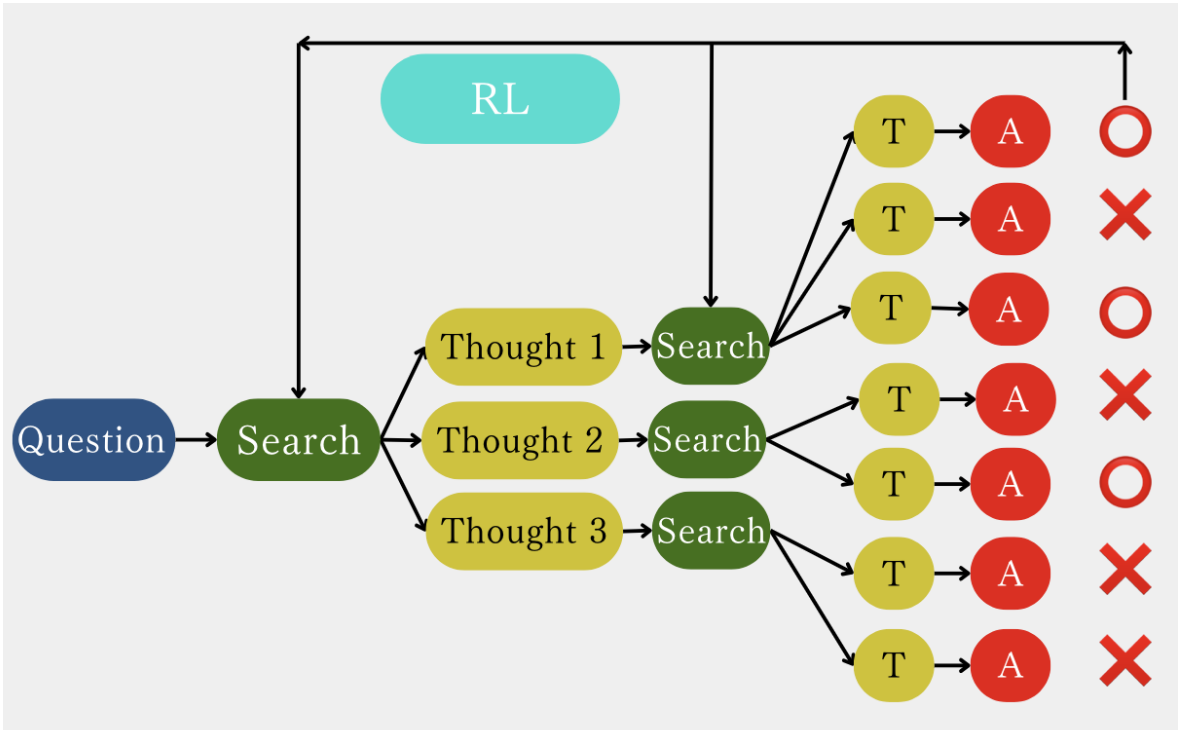

Figure 1: SEIMEI's inference and learning pipeline. Reinforcement-learning on the search model improves the model reasoning path through guiding its thought.

- **Tree of Thoughts (ToT)**: explicitly search a tree of partial solutions, backtrack, and evaluate branches [13].

A simple analogy: instead of trusting the first draft, you ask the model to generate multiple candidate "proof attempts," then select or refine the best. This "test-time search" theme is directly relevant to agentic coding pipelines, where we can try multiple patch candidates and score them.

Why RL can help, and why it can be expensive. Empirically, preference/RL-style post-training often improves **instruction following** and sometimes **reasoning benchmark performance**, but it can be resource-intensive because it requires generating many samples (rollouts) and performing multiple optimization steps (e.g., PPO) [5]. This motivates exploring **RL-free** (or "more supervised-like") alternatives such as DPO where appropriate [10].

1.3 SEIMEI

In this work we use **SEIMEI**, an open-source library that orchestrates LLM-based inference as a **search-integrated agent pipeline**. Conceptually, SEIMEI is designed for tasks requiring domain-specific knowledge and reasoning. SEIMEI realizes this by reinforcement-learning on a search model which integrates agents and knowledge [34].

Key idea (plainly): instead of training an LLM on next-token prediction for the domain task, SEIMEI **trains a search model to guide inference**. This has the following advantages over the previous method:

1. Training the search model doesn't break the core inference LLM, which prevents core inference collapsion (TODO: add continuous learning citation).

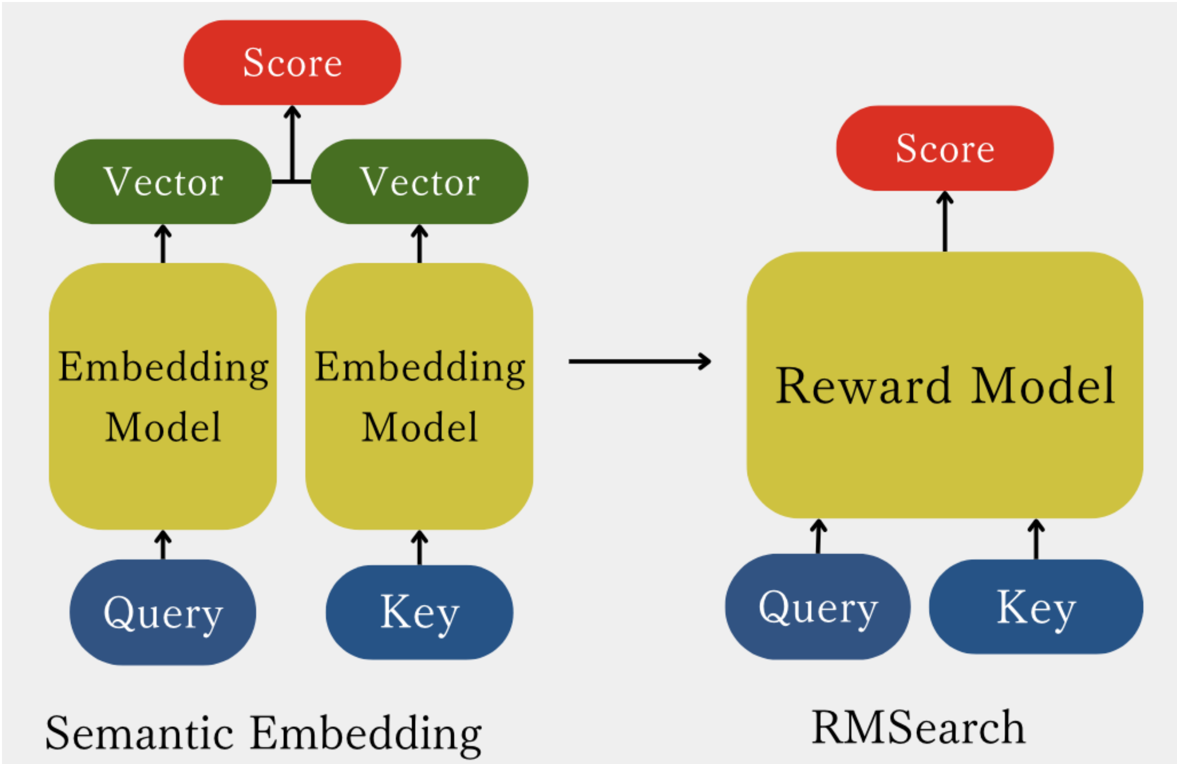

Figure 2: Model architecture difference between conventional vector search model and RMSearch.

2. Adapting the search model to one domain requires much less calculation cost than training a next-token-generation LLM.

Search has been a key technology for knowledge expansion. Search engines connect documents created by many contributors and enable humans to enhance the whole system by adding their own knowledge. This flexibility is key to expanding knowledge for an AI system. SEIMEI has the potential to become a new AI system: the search model integrates not only knowledge but also "how to think"—beyond current workflow agent paradigms.

1.4 RMSearch

SEIMEI's routing and knowledge selection use **RMSearch**, a reward-model-style search/reranking component. RMSearch retrieves and prioritizes helpful snippets (knowledge, prior solutions, docs, heuristics) that augment the LLM's next step [34].

Architecturally, RMSearch is closest to a **reranker**: given a query and candidate texts, it scores query–candidate relevance (often with a cross-encoder-style setup). This aligns with a long line of reranking research, including BERT-style rerankers and efficient interaction models [23, 24].

The practical difference is the retrieval pipeline. In many systems, a dense retriever first returns a top-k set (e.g., 100), and a reranker then reorders that shortlist to select the final few. RMSearch instead scores candidates directly with the reward model, so relevance is computed with the same signal used for selection rather than as a separate post-processing stage.

RMSearch (reranking) matters especially in domain-specific search, where generic embedding similarity can miss task-relevant signals. A dense retriever compresses text into vectors, while a reward model learns to score relevance directly from paired query-candidate text. This makes RMSearch easier to adapt to a specific domain with preference data [34].

1.5 Goal of our research

We aim to improve LLM-assisted **plasma simulation code writing/editing** using SEIMEI + RMSearch, with an emphasis on gyrokinetic code (GKV). Our plan:

1. Build an inference system using SEIMEI.
2. Generate a dataset from a real gyrokinetic codebase.
3. Generate a **knowledge pool** by solving/repairing tasks (and extracting reusable lessons).
4. Train RMSearch (reranker-style) over that knowledge pool using preference-style data (DPO).
5. Measure whether the knowledge pool + trained RMSearch improves code-repair accuracy in the same domain.

(Experiments/results are not included in this baseline draft.)

1.6 Related Research

Below are the main research threads we build on; this section is intentionally plain and cross-disciplinary.

1.6.1 Retrieval-Augmented Generation (RAG)

RAG methods attach an external memory (documents, snippets, code, papers) to an LLM so it can **retrieve** relevant context rather than relying only on parameters. The core motivation is modularity:

- You can update knowledge by updating the corpus/index, without retraining the whole model.
- You can attach provenance ("this answer used these sources").

Classic RAG combines a neural retriever with a generator for knowledge-intensive tasks [19]. Related lines include retrieval during pretraining (REALM) [20] and retrieval-enhanced generation at very large scale (RETRO) [21]. Dense retrieval also became a standard baseline for open-domain QA and retrieval pipelines [22].

Why it matters here: codebases are "documents." A fusion code repository contains the ground truth for function contracts, data layouts, and physics assumptions. RAG-style grounding reduces hallucinated edits.

1.6.2 AI agents (tool use, browsing, iterative editing)

Modern "LLM agents" combine a language model with tools and iterative control:

- Tool use learned or prompted (e.g., Toolformer) [15].
- Modular architectures mixing LMs with specialized components (MRKL) [16].
- Reason+act prompting patterns (ReAct), useful for multi-step tasks [14].

- Web/tool-assisted answering and reference collection (WebGPT), relevant for grounded reasoning workflows [17].
- Software-engineering agents with repository navigation and editing interfaces (SWE-agent) [18].

Agent frameworks often emphasize interfaces (file editing, running checks, browsing) rather than only better prompts—this aligns with our SEIMEI design goal of structured code repair [18].

1.6.3 DSPy (automatic pipeline improvement from evaluation)

DSPy proposes a programming model where you declare an LLM pipeline, define a metric, and let a compiler-like optimizer improve prompts/modules using data [25]. **Connection to our goal:** we also want an evaluation-driven improvement loop, but focused on (a) code repair tasks, and (b) improving retrieval/reranking (RMSearch) and knowledge pools that feed the pipeline.

1.6.4 Reranker models (relevance scoring)

Rerankers are a standard IR technique: a fast retriever gets candidates; a stronger model reorders them. BERT reranking (monoBERT) demonstrated large gains in passage ranking [23]. ColBERT provides an efficiency–quality tradeoff via late interaction [24]. **Connection to our work:** RMSearch plays the reranker role for "which knowledge or context should the agent use next," which becomes crucial in specialized domains like gyrokinetics.

2 Approach

2.1 Overview

Our approach is an end-to-end loop that turns a real fusion codebase into (1) code-repair tasks, (2) an agentic inference pipeline, and (3) training data for a domain-adapted reranker (RMSearch):

1. **Dataset creation:** intentionally break code, then create a question that asks for a repair.
2. **Inference pipeline:** given (question, broken code), generate a patch/repair candidate.
3. **Scoring:** evaluate candidate repairs (static checks, diff quality, possibly tests/compilation when available).
4. **Knowledge generation:** collect reusable "lessons" from successful (and failed) repairs.
5. **RMSearch dataset construction:** create preference pairs over candidate knowledge/context and/or candidate repairs.
6. **Train RMSearch** (DPO-style preference optimization).
7. **Evaluate:** run the inference pipeline again, now routed/augmented by trained RMSearch.

This is designed to be feasible without full RLHF infrastructure, while still leveraging preference-style learning.

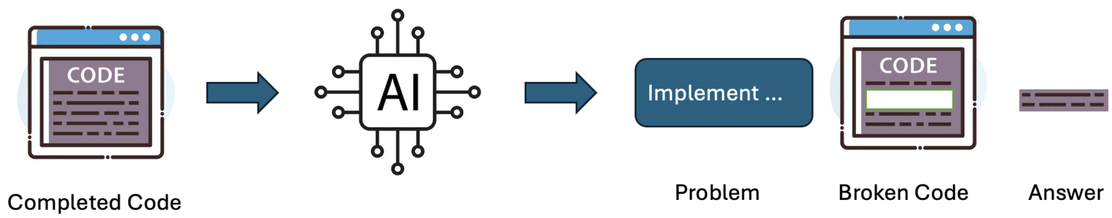

Figure 3: Dataset generation pipeline. Make an LLM intentionally break complete code and generate a problem related to repairing the code snippet.

2.2 Dataset Creation

Target repository: we use the open gyrokinetic Vlasov simulation code **GKV (gkvp)** as the source code to generate dataset from [27].

Task type (plain description): "repair the code so it matches the original intended physics-aware behavior."

Concretely, for a given file:

1. An LLM selects **which line(s) to break**, with the constraint that the deletion should affect **core physics logic** (not only comments or trivial formatting).
2. The pipeline generates:
   - a **patch** that removes important lines,
   - a **question** describing the observed failure or missing behavior ("restore the missing computation / boundary condition / term") and asking an LLM to fix the issue.
3. **Patch debugging:** a patch is often invalid when applied to the file. An LLM attempts to modify the patch based on the error message for several iterations. Patches that still error after this process are removed.

Why "break-and-repair" is useful:

- **Correctness:** it provides a clean answer patch because the correct fix is known; an LLM can check correctness by comparing against the original code.
- **Scalability:** this method only requires complete code and LLM calls; it can scale to other plasma-physics codebases or other domains.
- **Realistic:** it creates physics-centered code editing problems researchers often encounter in small code additions or fixes related to plasma equations.

2.3 Inference Pipeline

We implement the inference loop using SEIMEI [34]. There are basically two agents used in analyzing the GKV code:

- **code_act agent:** runs Unix commands (e.g., `cat`, `ls`, `rg`) or Python code to analyze local folders and files.

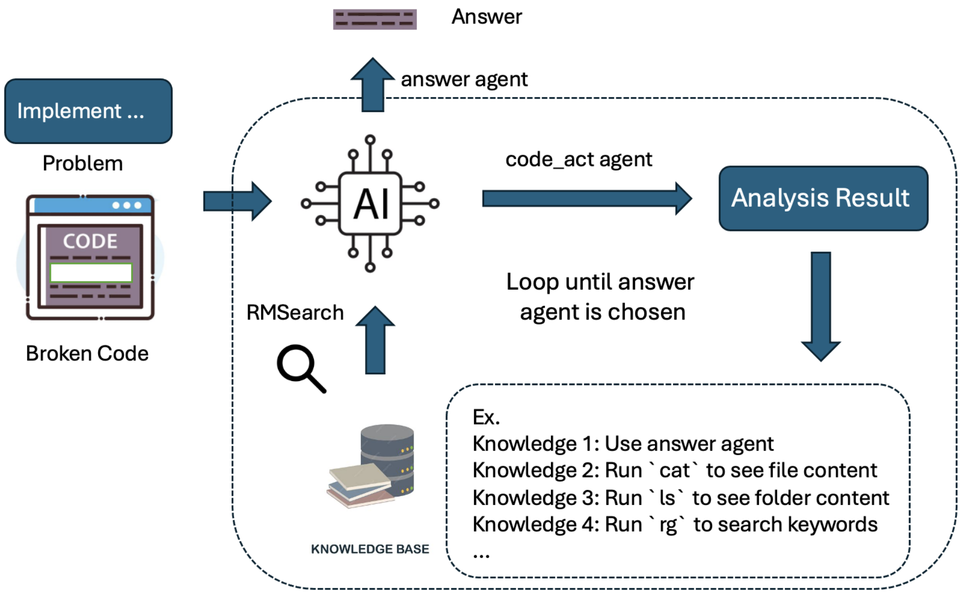

Figure 4: SEIMEI inference pipeline (conceptual).

- **answer agent:** gives the final answer summarizing all analysis results of the code_act agent.

This aligns with patterns demonstrated in prior agent work for tool use and repository-scale editing: ReAct-style loops [14], tool-augmented LMs (Toolformer) [15], modular systems (MRKL) [16], SWE-agent [18], and OpenHands [32].

Compared to these methods, SEIMEI has an additional feature: **routing and augmentation**. At each step, SEIMEI can call RMSearch to retrieve domain-relevant knowledge snippets and incorporate them into the next model call [34].

We use RMSearch for integrating a search model rather than a conventional semantic-embedding model because RMSearch (a reranker) calculates relevance scores more directly and therefore can be more adaptable to a specific domain than embedding similarity [33].

2.4 Knowledge Generation

We generate knowledge in two complementary ways:

1. **Automatic knowledge generation (from repairs):** compare the agent's output patch to the expected fix and extract reusable statements, e.g.,
   - "When modifying the gyroaveraged potential term, also update normalization in ____."
   - "This file assumes flux-tube geometry; boundary conditions are enforced in ____."

   The goal is to turn "one solved instance" into a hint that helps future instances.

2. **Manual knowledge:** domain experts (or careful readers) write concise task instructions, e.g.,
   - "Run answer agent because enough agent outputs are obtained."
   - "Run `ls` command to see what are in current folder."

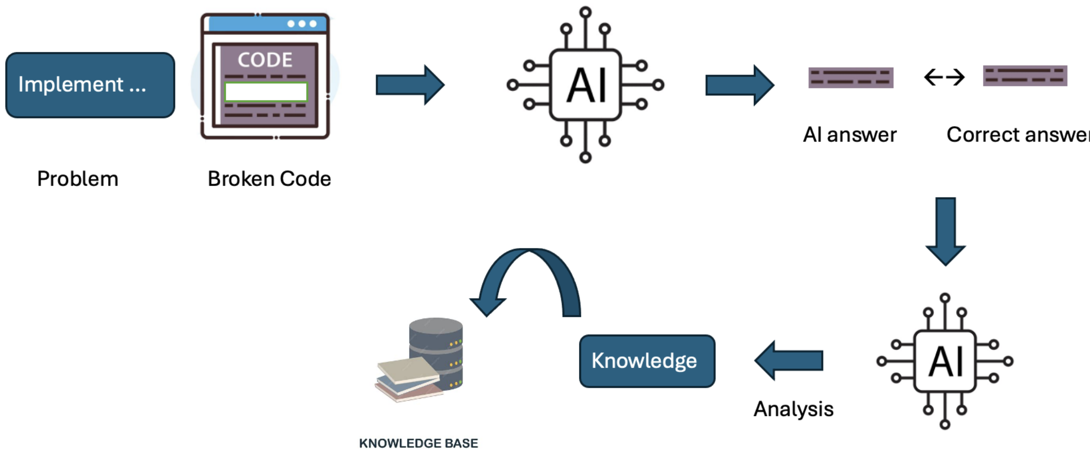

Figure 5: Knowledge generation pipeline from correct answers.

- "Run `cat` command to see what's inside a file."
- "Run `rg` command to search keywords over a folder."
- "Change the strategy to solve the question to file-comparison-centered analysis."

We keep both because automatic knowledge can scale, while manual knowledge can be higher precision.

2.5 Dataset for RMSearch

RMSearch training requires a preference-style dataset. We construct it from pipeline runs by creating comparisons like:

- **Knowledge preference:** given a query ("fix this missing physics term") and previous agent outputs, compare two candidate knowledge snippets A vs. B based on whether using them leads to a higher-scoring answer.

This matches the general "learning from preferences" paradigm used in RLHF, but we apply it to **retrieval/reranking and agent routing** rather than directly to the generator model [8].

In this method, knowledge sampling and preference assignment are key. The sampling procedure used in this experiment is:

- **Knowledge generation or selection:** from the full message history of a base inference trial, an LLM generates or selects knowledge for designated steps. A set of knowledge texts spanning several steps is called a **knowledge chunk**. We generate multiple chunks for each problem.
- **Rerun inferences with knowledge chunks:** using each generated or selected knowledge chunk, rerun the inference. Several runs are performed per chunk.

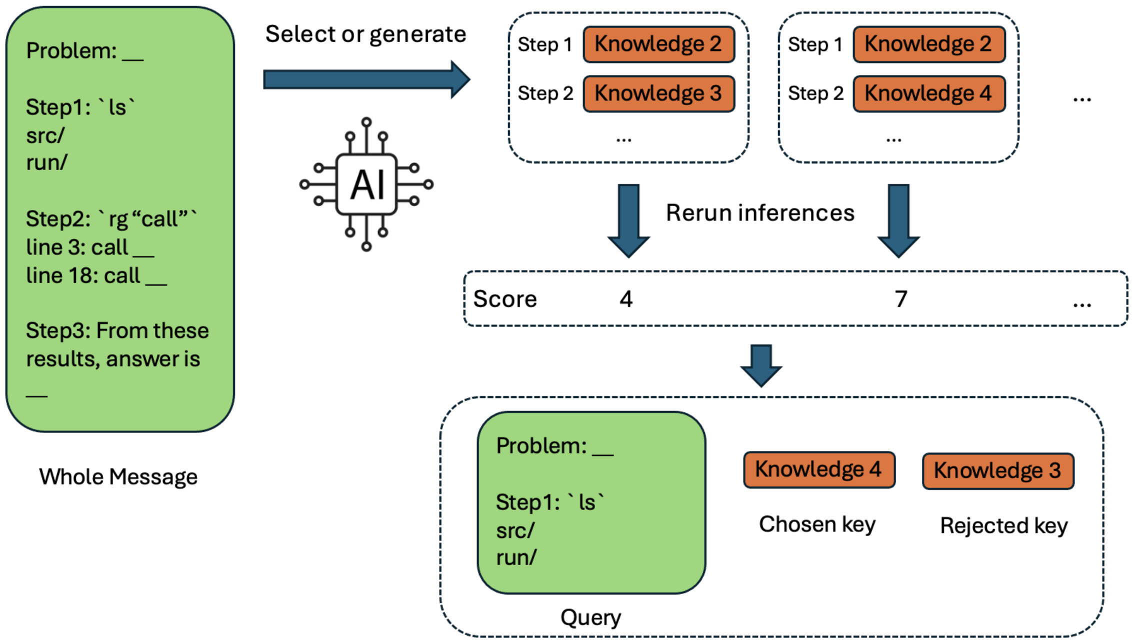

Figure 6: Preference-style dataset creation for DPO training (conceptual).

- **Scoring knowledge chunks:** score each inference run and average scores per knowledge chunk to obtain a chunk-level score.
- **Converting to a preference dataset:** from chunk scores, build (query, chosen knowledge, rejected knowledg pairs. The query is the message history up to the step that will be augmented by the chosen or rejected knowledge. A threshold removes trivial score differences.

2.6 RMSearch Training

We train RMSearch with **DPO-style preference optimization**, treating the reranker as a model that should assign higher score to preferred items (knowledge snippets or patches) [10].

Concretely, RMSearch implements a scoring function that maps a **query + key** pair to a scalar score, i.e., $s_\theta(q, k) \to r$. For a preference triple (q, k^+, k^-), we use the DPO loss:

$$\mathcal{L}_{\text{DPO}} = -\log \sigma \left(\beta \left[s_\theta(q, k^+) - s_\theta(q, k^-) \right] \right), \tag{1}$$

where q is the query, k^+ is the preferred key (knowledge or candidate patch), k^- is the less-preferred key, s_θ is the RMSearch score function, β controls the sharpness of the preference margin, and σ is the logistic function. This makes the connection to RMSearch explicit: training adjusts the **query + key** $\to$ **score** mapping so that preferred knowledge/patches receive higher scores, which directly improves reranking at inference time.

Batch construction (following InstructGPT's RM training recipe). A practical issue in preference training is that many pairwise comparisons derived from the *same* query are highly correlated. In the InstructGPT reward-modeling pipeline, labelers rank K candidate completions per prompt (with K between 4 and 9), yielding up to $\binom{K}{2}$ pairwise comparisons; the authors found that naively shuffling these comparisons and training on them as independent datapoints caused rapid overfitting. Instead, they treat **all $\binom{K}{2}$ comparisons from a single prompt as one batch element**, which is both (i) **more compute-efficient** (one forward pass per candidate rather than $\binom{K}{2}$ passes) and (ii) **more stable** (reduced overfitting, improved validation loss/accuracy) [5].

We adopt the same batching principle for RMSearch: each minibatch contains B distinct queries, and for each query we score K candidate keys and apply the pairwise preference loss over the within-query comparison set, rather than mixing comparisons across queries indiscriminately. This keeps updates aligned with the "within-query reranking" structure that RMSearch must solve at inference time.

2.7 RMSearch & Inference Evaluation

3 Experiment

3.1 `rmsearch-plasma-1.0` Evaluation

3.2 `seimei-plasma-1.0` Evaluation

3.3 Human Evaluation

4 Discussion

4.1 Possibility of Domain Specific LLM

4.2 Unsuccessful Attempts

- sampling methods
- making dataset with codex

5 Conclusion

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017. https://arxiv.org/abs/1706.03762.

[2] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. 2020. https://arxiv.org/abs/2001.08361.

[3] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Dieuwke Hupkes, Scott H. Hawkins, et al. Training compute-optimal large language models. In *NeurIPS*, 2022. https://arxiv.org/abs/2203.15556.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, et al. Language models are few-shot learners. In *NeurIPS*, 2020. https://arxiv.org/abs/2005.14165.

[5] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, et al. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022. https://arxiv.org/abs/2203.02155.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 2017. https://arxiv.org/abs/1707.06347.

[7] Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *NeurIPS*, 2017. https://arxiv.org/abs/1706.03741.

[8] Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. 2019. https://arxiv.org/abs/1909.08593.

[9] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, John Schulman, et al. Learning to summarize from human feedback. In *NeurIPS*, 2020. https://arxiv.org/abs/2009.01325.

[10] Rafailov, et al. Direct preference optimization. 2023. https://arxiv.org/abs/2305.18290.

[11] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. 2022. https://openreview.net/forum?id=_VjQlMeSB_J.

[12] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. 2022. https://ui.adsabs.harvard.edu/abs/2022arXiv220311171W/abstract.

[13] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: deliberate problem solving with large language models. In *NeurIPS*, 2023. https://openreview.net/forum?id=5Xc1ecxO1h.

[14] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. 2023. https://github.com/ysymyth/ReAct.

[15] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, et al. Toolformer: language models can teach themselves to use tools. 2023. https://arxiv.org/abs/2302.04761.

[16] Eliot Karpas, et al. MRKL systems. 2022. https://arxiv.org/abs/2205.00445.

[17] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, et al. WebGPT: browser-assisted question-answering with human feedback. 2021. https://arxiv.org/abs/2112.09332.

[18] John Yang, Carlos E. Jimenez, Alex Liska, et al. SWE-agent: agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024. https://arxiv.org/abs/2405.15793.

[19] Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*, 2020. https://arxiv.org/abs/2005.11401.

[20] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. REALM: retrieval-augmented language model pre-training. 2020. https://arxiv.org/abs/2002.08909.

[21] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, et al. Improving language models by retrieving from trillions of tokens (RETRO). In *ICML*, 2022. https://arxiv.org/abs/2112.04426.

[22] Vladimir Karpukhin, Barlas Oguz, Sewon Min, et al. Dense passage retrieval for open-domain question answering. In *EMNLP*, 2020. https://arxiv.org/abs/2004.04906.

[23] Rodrigo Nogueira and Kyunghyun Cho. Passage re-ranking with BERT. 2019. https://arxiv.org/abs/1901.04085.

[24] Omar Khattab and Matei Zaharia. ColBERT: efficient and effective passage search via contextualized late interaction over BERT. 2020. https://arxiv.org/abs/2004.12832.

[25] Omar Khattab, et al. DSPy: compiling declarative language model calls into self-improving pipelines. 2023. https://arxiv.org/abs/2310.03714.

[26] Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. 2021. https://arxiv.org/abs/2107.03374.

[27] GKV-developers. gkvp: GyroKinetic Vlasov simulation code (repository). https://github.com/GKV-developers/gkvp.

[28] T. Görler, X. Lapillonne, S. Brunner, et al. The global version of the gyrokinetic turbulence code GENE. 2011. https://pure.mpg.de/rest/items/item_2139664/component/file_2139663/content.

[29] W. Dorland, F. Jenko, M. Kotschenreuther, and B. N. Rogers. Gyrokinetic simulations of tokamak microturbulence. 2000. https://w3.pppl.gov/~hammett/gyrofluid/papers/dorland00-iaea.pdf.

[30] I. Holod, Z. Lin, and Y. Xiao. Gyrokinetic particle simulations of toroidal momentum transport. *Physics of Plasmas*, 2008. https://pubs.aip.org/aip/pop/article/15/9/092302/282005/Gyrokinetic-particle-simulations-of-toroidal.

[31] M. Nakata, T.-H. Watanabe, and H. Sugama. Validation studies of gyrokinetic ITG and TEM turbulence … *Nuclear Fusion*, 2016. https://nifs-repository.repo.nii.ac.jp/record/10912/files/Nucl.Fusion_56_pp086010.pdf.

[32] OpenHands. OpenHands (coding agent platform, repository/site). https://github.com/OpenHands/OpenHands.

[33] KyotoAI. RMSearch (repository). https://github.com/kyotoai/RMSearch.

[34] KyotoAI. SEIMEI (repository). https://github.com/kyotoai/SEIMEI.

[35] DeepSeek. DeepSeek-R1 (report/repository and coverage). https://arxiv.org/abs/1706.03762.