LLM-Assisted Plasma Simulation Code Automation: Draft 3

Kentaro Seki (seki.kentaro.66s@st.kyoto-u.ac.jp)
Kyoto University
Date: today

Abstract

This thesis studies LLM-assisted code repair for gyrokinetic plasma simulation software. Using the GKV (gkvp) codebase, we build a repair dataset by intentionally removing physics-critical lines and generating repair questions with known ground-truth patches. We implement SEIMEI, an agentic inference pipeline that navigates the repository and proposes fixes, and augment it with RMSearch, a reward-model-based reranker that selects task-relevant knowledge snippets during inference. Preference data for RMSearch is constructed from inference runs and trained using a DPO-style objective.

Experiments on 113 repair tasks (85 train / 28 test) show that RMSearch training reduces evaluation loss and improves reranking accuracy across iterations. When RMSearch is integrated into SEIMEI, repair accuracy improves by about 7% per iteration and exceeds 20% improvement by the third iteration compared to a baseline pipeline without RMSearch. These results suggest that domain-specific retrieval and lightweight preference training can yield practical gains in scientific code repair with modest computational cost.

Contents

1. Introduction

1.1 Numerical simulations of fusion plasmas (big picture of fusion research)

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). (MPG.PuRe)

Fusion devices are large-scale, and plasma phenomena span wide spatiotemporal scales that cannot be fully measured. Numerical simulation is therefore essential for device design, operation scenarios, and understanding plasma physics. To cover wide scales and incorporate detailed physics, simulation codes are large and interdependent; advances in computing have improved reproducibility but raised the barrier to new contributors.

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 small physics change (e.g., a collision term, equilibrium interface, or diagnostic) often requires edits across 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 understand unfamiliar codebases and implement changes more reliably.

1.2 Ultimate research goal: problem solving with LLMs (knowledge pool and retrieval learning are key)

The long-term goal is to build an LLM inference system that improves reasoning and conversation with plasma researchers:

1. Improve reasoning by providing the system with source code and relevant papers/documents.
2. Improve conversation by capturing expert knowledge during interactions.
3. Reuse that knowledge in later sessions to solve new problems more effectively.

1.3 Bachelor thesis goal: error removal (subject to change)

For this bachelor thesis, the near-term goal is error removal and evaluation of an LLM-based system:

- Create a dataset from a fusion simulation codebase.
- Train the system and evaluate its performance on repair tasks.
- Refine the workflow and knowledge pool based on observed failure modes.

1.4 Organization of the thesis

The paper is organized as follows:

- Section 2 reviews LLM research and the SEIMEI/RMSearch framework.
- Section 3.2 describes dataset construction with (problem, code, answer) tuples.
- Section 3.3 presents the inference pipeline for codebase analysis and patch generation.
- Section 3.4 explains knowledge creation for problem solving.
- Sections 3.5 to 3.7 describe knowledge search training and evaluation.
- Sections 4 to 6 present experiments, discussion, and conclusions.

2. Large Language Model

2.1 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) 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. (arXiv)

(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. (arXiv) 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). (arXiv)

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.

(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. (arXiv) The broader "learning from preferences" setup is also well-established in RL. (arXiv)

(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. (arXiv)

(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. (arXiv)

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.

(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. (OpenReview)
- **Self-consistency**: sample multiple reasoning traces and pick the majority-consistent answer. (Astrophysics Data System)
- **Tree of Thoughts (ToT)**: explicitly search a tree of partial solutions, backtrack, and evaluate branches. (OpenReview)

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). (arXiv) This motivates exploring **RL-free** (or “more supervised-like”) alternatives such as DPO where appropriate. (arXiv)

2.2 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 search model which integrates agents and knowledge. (GitHub)

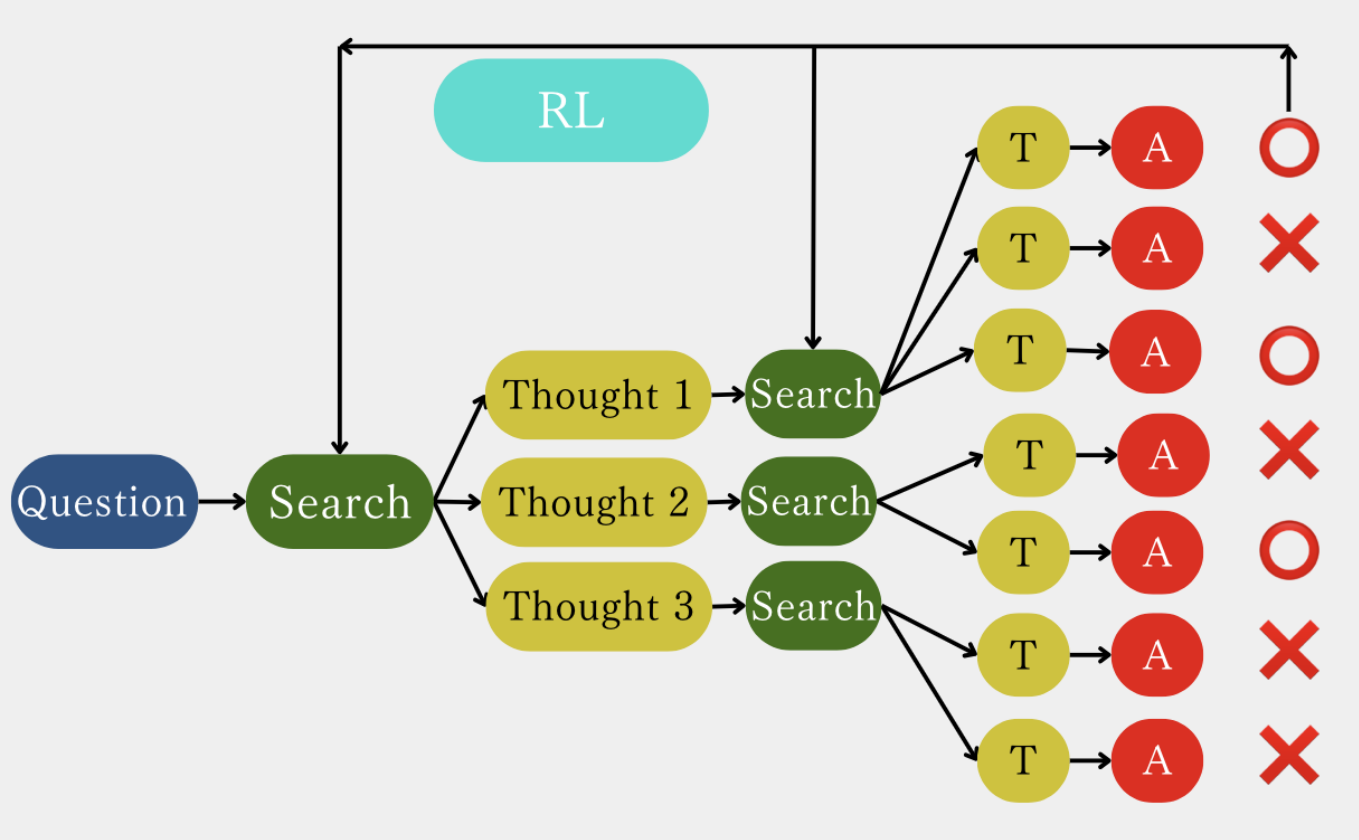


Figure | SEIMEI's inference and learning pipeline. Reinforcement-learning on search model improves the model reasoning path through guiding its thought.

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

1. Training search model doesn't break the core inference LLM, which prevents core inference collapsion (!need to add continuous learning citation).
2. Adapting search model to one domain requires much less calculation cost than training next-token-generation LLM.

Search has been a key technology for knowledge expansion. Search-engine has connected numerous amount of documents created by citizens and enabled human-beings to enhance the whole search-engine system by adding their own knowledge. This flexiblity to adding knowledge is key to expanding knowledge for an AI system. SEIMEI has a potential to become a new AI system - search model integrates not only simple knowledge but also how to think - beyond current workflow agent paradigm.

2.3 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. (GitHub)

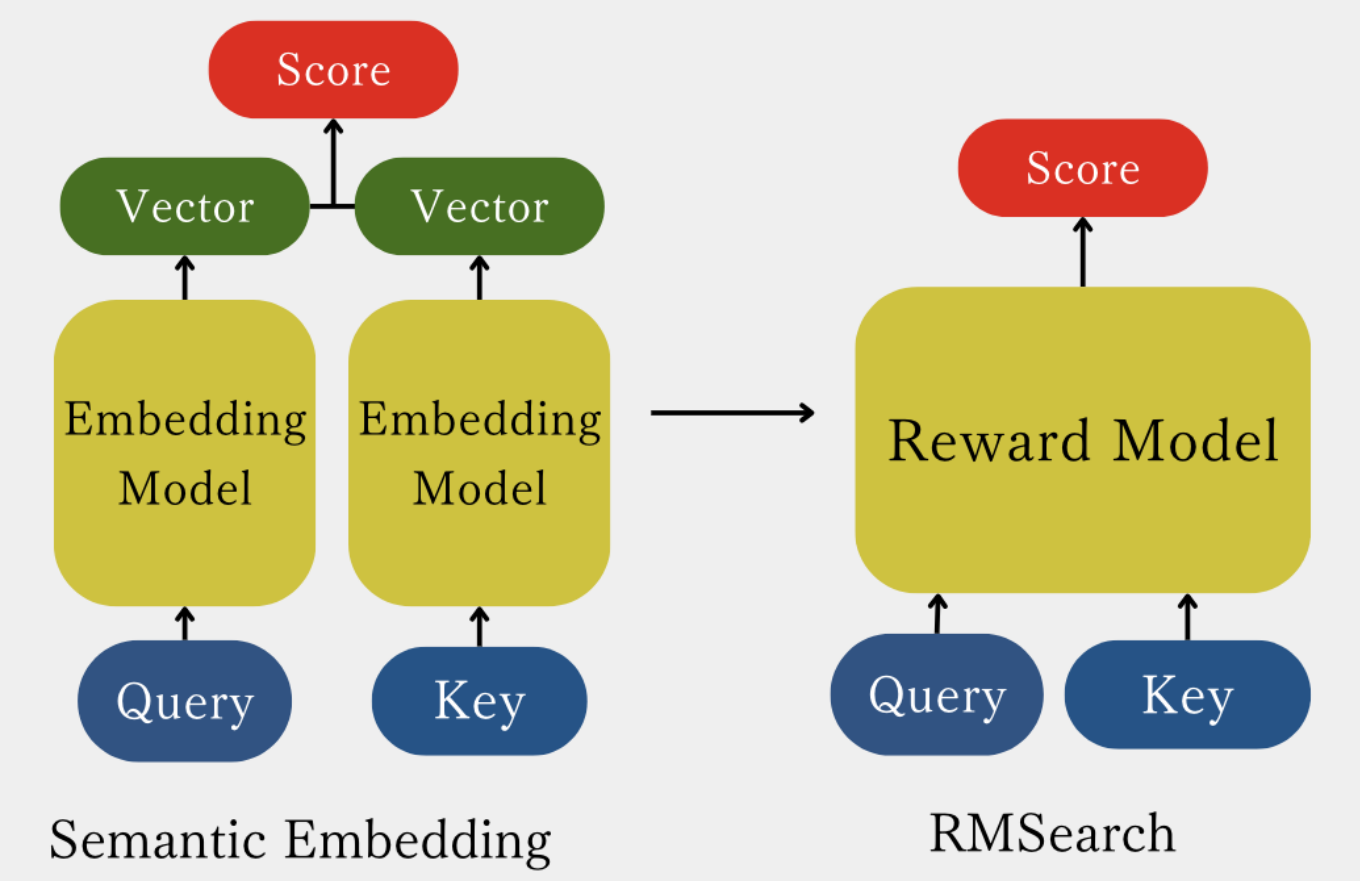


Figure | Model architecture difference between conventional vector search model and RMSearch.

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. (arXiv)

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. (GitHub)

2.4 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, per your request.)

2.5 Related Research

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

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 work combines a neural retriever with a generator for knowledge-intensive tasks. (arXiv) Related lines include retrieval during pretraining (REALM) (arXiv) and retrieval-enhanced generation at very large scale (RETRO). (arXiv) Dense retrieval also became a standard baseline for open-domain QA and retrieval pipelines. (arXiv)

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.

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). (arXiv)
- Modular architectures mixing LMs with specialized components (MRKL). (arXiv)
- Reason+act prompting patterns (ReAct), useful for multi-step tasks. (GitHub)
- Web/tool-assisted answering and reference collection (WebGPT), relevant for grounded reasoning workflows. (arXiv)
- Software-engineering agents with repository navigation and editing interfaces (SWE-agent). (arXiv)

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

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. (arXiv)

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.

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. (arXiv) ColBERT provides an efficiency–quality tradeoff via late interaction. (arXiv)

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.

---

3. Approach

3.1 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.

3.2 Dataset Creation

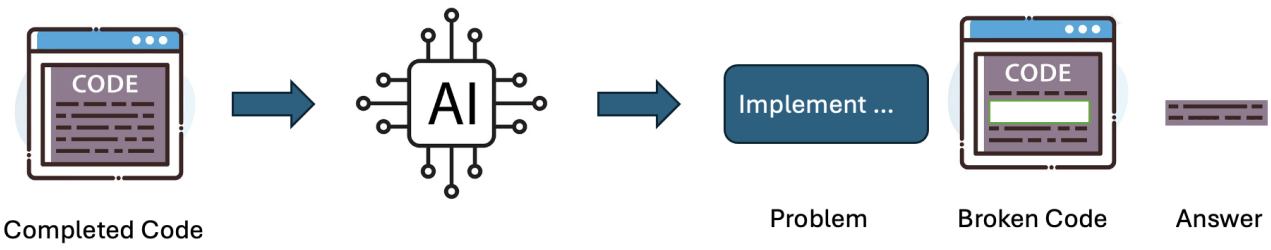


Figure | Dataset generation pipeline. Make LLM intentianlly break complete code and generate problem related to repairing the code snippet.

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

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 ask LLM to fix the issue.
3. An LLM debug patches: a patch often is not valid when it is applied to the file, so an LLM trys to modify the patch file from the error message for several times in this stage. If some patches still get error after this process, they are removed.

Why "break-and-repair" is useful:

- **Correctness:** It provides a clean answer patch because the correct fix is known. LLM can easily check if the LLM output is correct by comparing it with the original code.
- **Scalability:** This method only requires complete code and LLM request. It can be easily scaled to other complete code about plasma physics or even to other fields.
- **Realistic:** It creates physics-centered code edition problem which researchers often encounter in small code addition or fix related to plasma equations.

3.3 Inference Pipeline

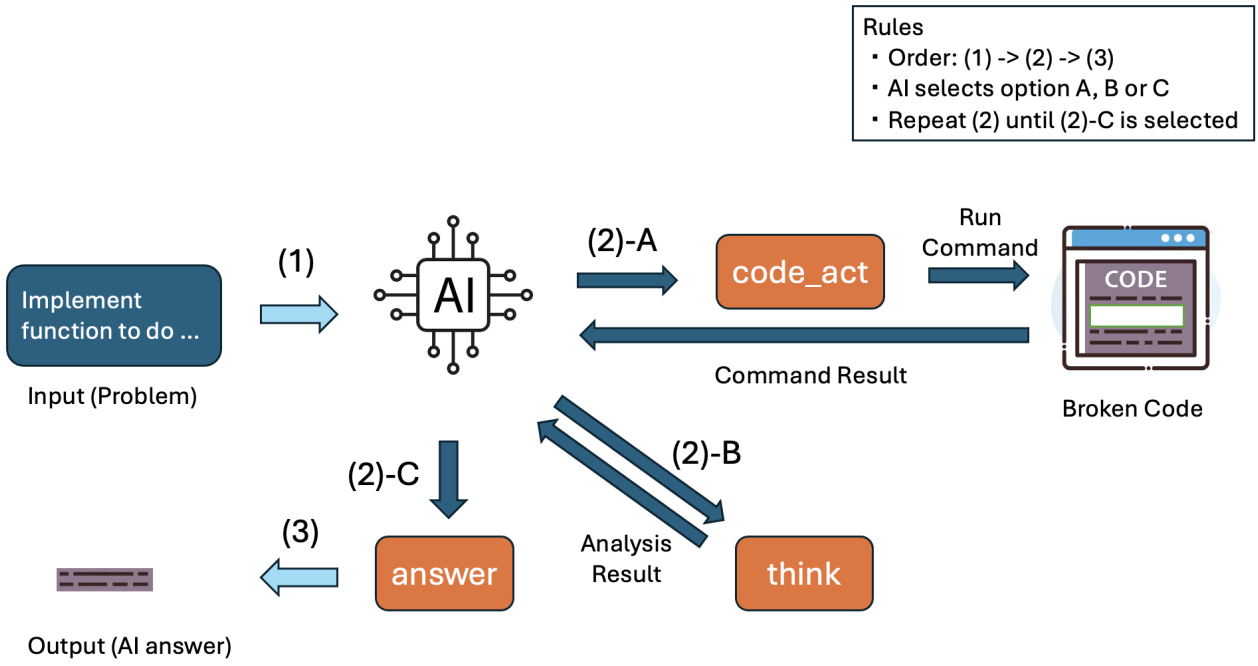


Figure | Normal approach to access code base and solve problem.

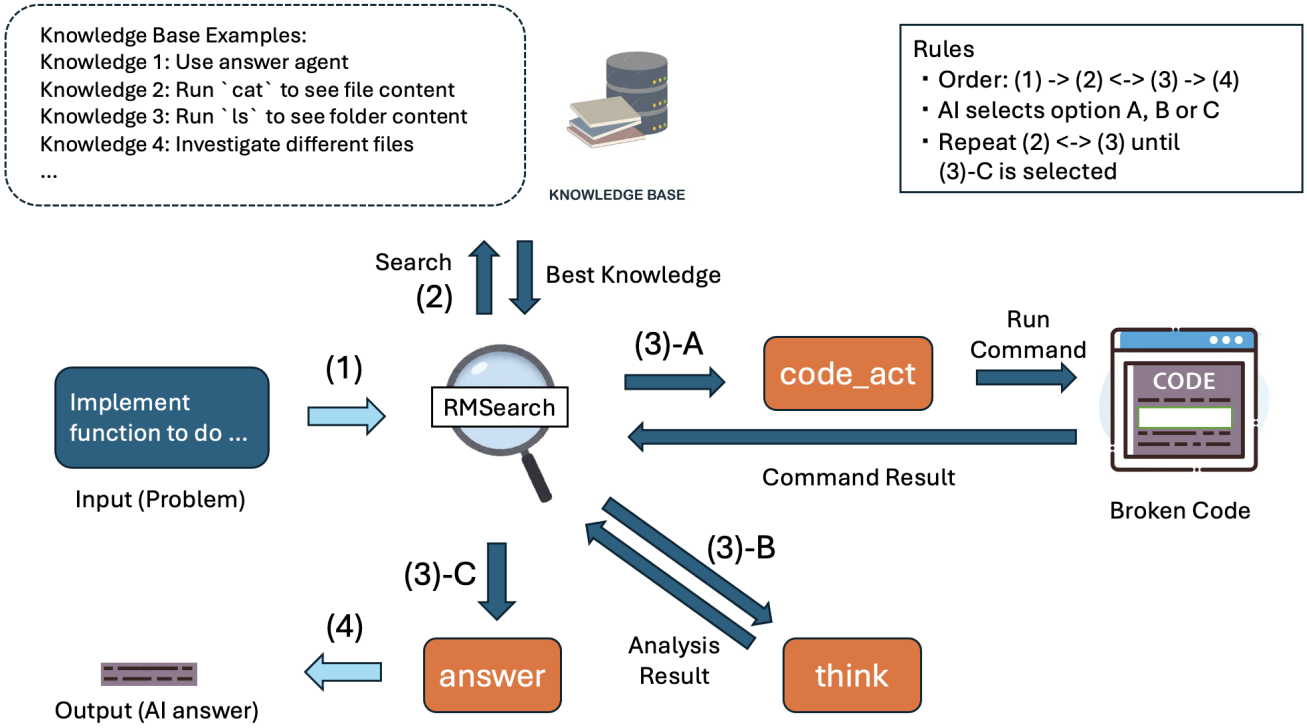


Figure | SEIMEI approach to access code base and solve problem.

We implement the inference loop using SEIMEI. (GitHub)

There are basically 3 agents used in analyzing gkv code.

- **code_act agent**: runs unix command (ex. `cat`, `ls`, `rg`) or python code to analyze local folder and files.
- **think agent**: see all command results and analysis results before, provide thought and decide what to do next.
- **answer agent**: gives final answer summarizing all the analysis result of code_act agent

This aligns with patterns demonstrated in prior agent work for tool use and repository-scale editing:

- **ReAct**-style reasoning+acting loops for stepwise progress. (GitHub)
- Tool-augmented LMs (Toolformer) and modular systems (MRKL). (arXiv)
- SWE-agent shows that carefully designed “agent–computer interfaces” improve code-editing success on benchmarks. (arXiv)
- OpenHands is an example of an open platform focused on coding agents. (GitHub)

Compared to the methods above, SEIMEI has an additional feature, **Routing and augmentation:** at each step, SEIMEI can call RMSearch to retrieve domain-relevant knowledge snippets, then incorporate them into the next model call. (GitHub)

We use RMSearch for integrating search model rather than conventional semantic-embedding model. It is because RMSearch, often refered to as reranker model, calculates relevance-score more directly and therefore is more adaptable to specific domain than semantic-embedding model. (GitHub)

3.4 Knowledge Generation

Knowledge pool is generated in two steps:

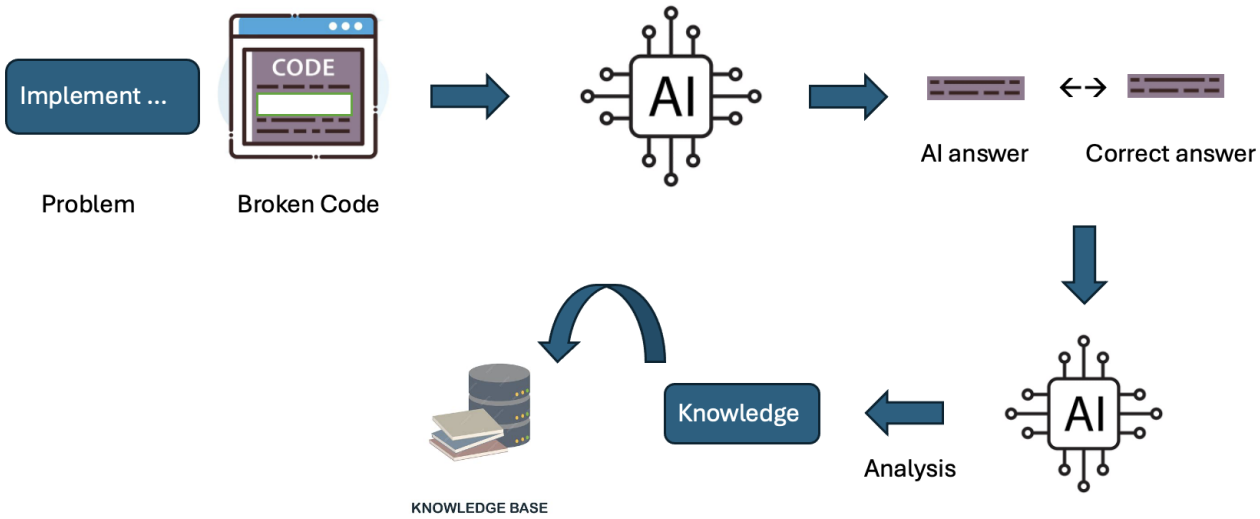


Figure | Knowledge update pipeline from correct answer.

1. **Manual knowledge:** Domain experts (or careful readers) write concise instructions about the task such as:

    - "Run answer agent because enough agent outputs are obtained."
    - "Run `ls` command to see what are in current folder."
    - "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."

    Writing this requires careful analysis of reasoning steps and AI answers.

2. **Automatic knowledge update (from repairs):** Compare the agent's output patch to the expected fix and extract reusable statements such as:

    - "When you act on `cat` command, act `ls` first to check file path in the current folder."
    - "You often cause errors with `rg` command. Be sure to follow the format: __."
    - "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.

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

3.5 Knowledge Sampling for RMSearch Dataset

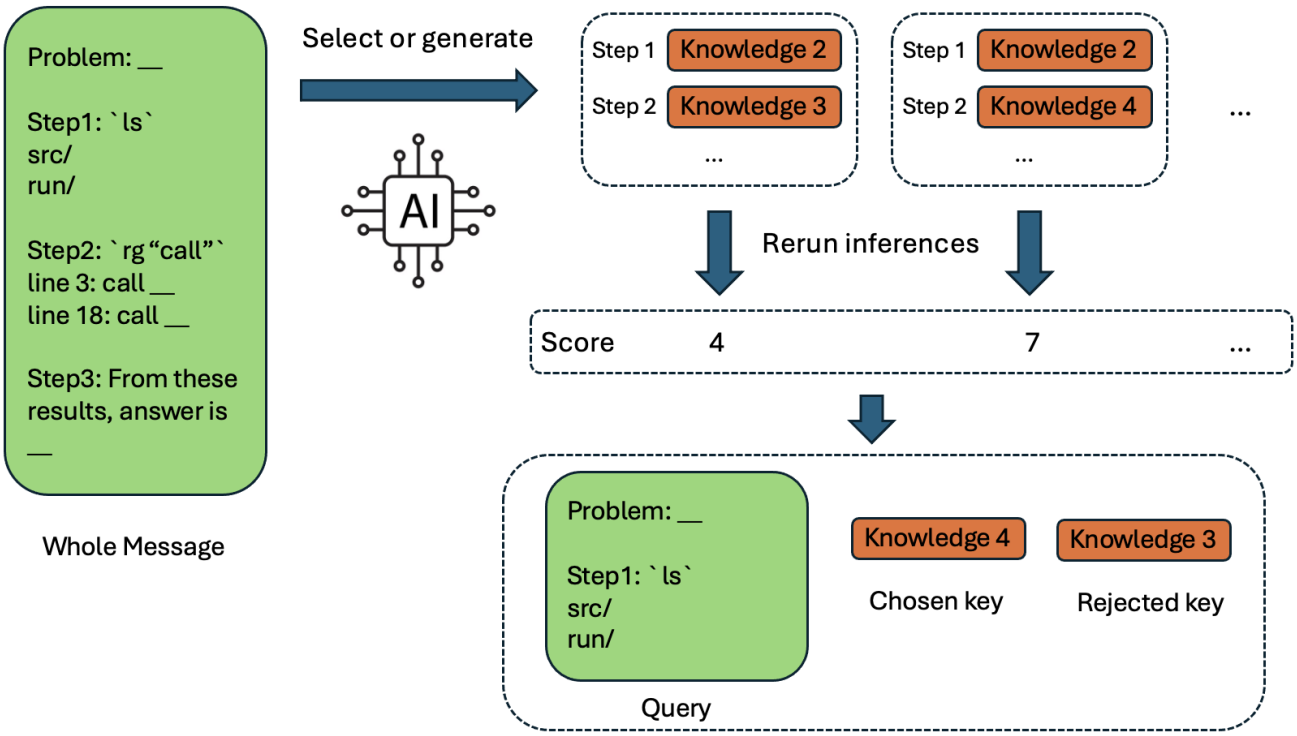


Figure | Preference-style dataset creation for DPO training (conceptual).

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. (arXiv)

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.
- **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 knowledge) 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.

3.6 RMSearch Training

3.6.1 Reward Design

The key difference from a standard RAG pipeline is that RMSearch is trained on a reward derived from the quality of the final repair. The reward is computed by comparing the model-generated patch with the ground-truth patch (the original code before deletion). We use a 0--10 score with the following components:

- +2 for modifying the correct file.
- +2 for modifying the correct location in that file.
- +3 for restoring the same functional code as the deleted portion.
- +3 for how directly the retrieved knowledge contributes to the reasoning steps (heuristically: +1 if the knowledge identifies the correct file, +1 if it identifies the correct code snippet, +1 if it materially improves the reasoning path).

This design keeps the reward tied to correctness while also encouraging RMSearch to select knowledge that is actually used by the agent to reach the fix.

3.6.2 Loss Function

RMSearch is trained with **DPO-style preference optimization**, treating the RMSearch as a model that should assign higher relevance-score to preferred items (knowledge snippets or patches). (arXiv)

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

$$ \mathcal{L}*{\mathrm{DPO}} = -\log \sigma\left(\beta\left[s*{\theta}(q,k^+) - s_{\theta}(q,k^-)\right]\right) $$

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 (reward) function, s_{ref} is a fixed reference scorer used to keep updates conservative, 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 -> reward** scores so that preferred knowledge/patches receive higher scores, which directly improves the reranking behavior at inference time.

3.6.3 Batch construction (following InstructGPT's reward model 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). 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. (arXiv)

3.7 Training Iteration

RMSearch training and SEIMEI inference form a bootstrapping loop. RMSearch is trained on data produced by SEIMEI runs, and the improved RMSearch then changes which knowledge is retrieved in the next round of SEIMEI inference. Concretely, each iteration follows:

1. Run SEIMEI with the current RMSearch (or a baseline retriever in iteration 0) to generate repairs and collect logs.
2. Build preference data from these runs (knowledge chunks or candidate patches) using the reward defined in Section 3.6.1.
3. Train RMSearch with the DPO objective.
4. Use the updated RMSearch for the next iteration of inference and data collection.

This loop is intended to steadily improve retrieval quality, which in turn improves the downstream repair accuracy without retraining the base LLM.

3.8 RMSearch & SEIMEI Evaluation

We use two metrics to evaluate RMSearch:

- **Evaluation loss:** the DPO loss defined in Section 3.6.2, which should decrease as the model learns to prefer better knowledge.
- **Accuracy:** the percentage of cases where RMSearch assigns the highest score to the preferred key for a given query.

For SEIMEI, we evaluate the inference pipeline using the same reward function defined in Section 3.6.1. We report accuracy based on whether the generated patch is judged correct under this scoring rule, and we also track reward trends across iterations.

---

4. Experiment

We evaluate on the dataset generated by the method in Section 3.2 using the GKV codebase. The dataset contains 113 repair tasks, split into 85 training problems and 28 test problems.

Models

- Inference model: `openai/gpt-oss-20b`
- Reward model / RMSearch backbone: `Skywork/Skywork-Reward-V2-Qwen3-4B` (referred to as `qwen4b-reward`)

Key hyperparameters

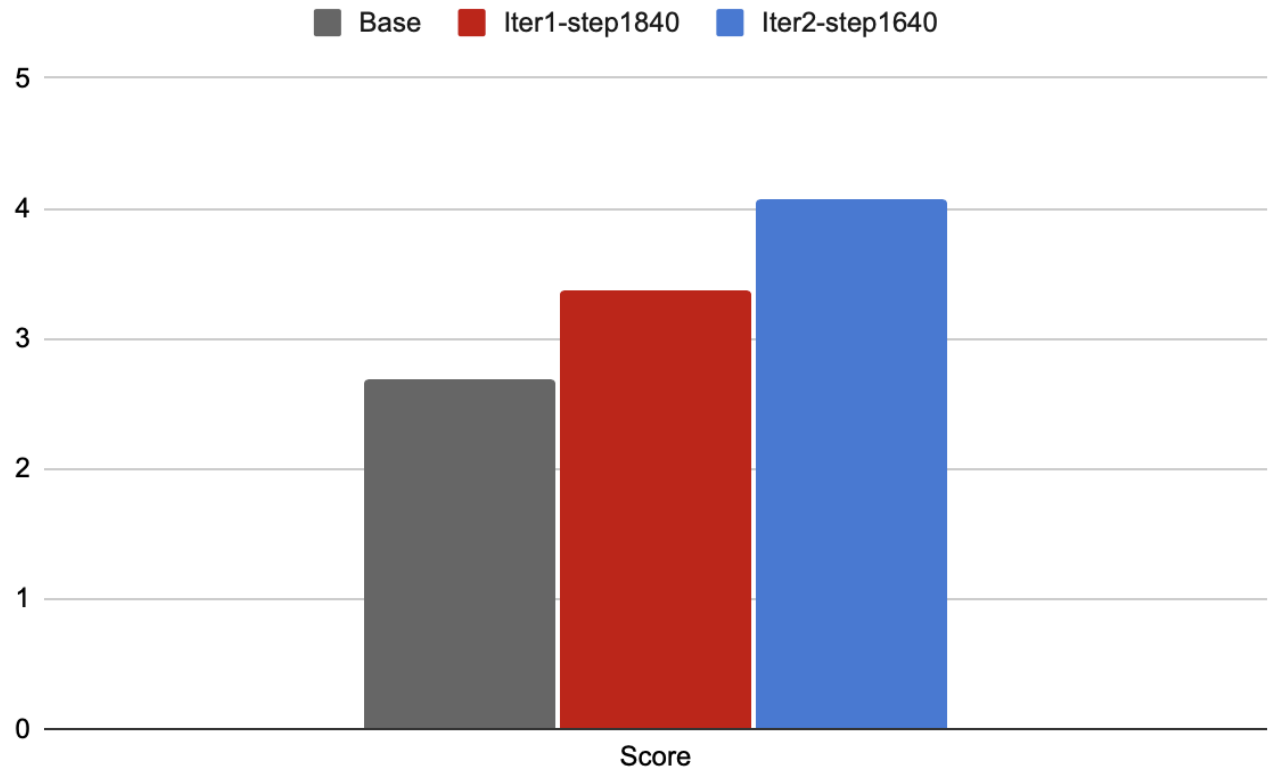
- Evaluation: 7 sampling runs per problem.
- Training: preference threshold = 0.5.
- Iteration 1: knowledge sampling model = `gpt-oss-20b`, distribution decay = 0.8, random sampling rate = 0.1, sampling number = 14.
- Iteration 2--3: knowledge sampling model = RMSearch trained in the previous iteration, distribution decay = 0.5, random sampling rate = 0.1, sampling number = 14.

4.1. RMSearch Evaluation

Figure | Evaluation log in RMSearch training for all iterations.

The evaluation loss decreases steadily across iterations, while accuracy increases, indicating that the RMSearch model becomes better at selecting the preferred knowledge/key for each query. Iteration 1 shows a different accuracy trajectory from iterations 2 and 3, which is expected because iteration 1 uses the base LLM for knowledge sampling, while later iterations use the trained RMSearch from the previous round.

4.2. SEIMEI Evaluation



Fugure | SEIMEI evaluation for all iterations. Base means that normal inference pipeline depected in section

3.3.

Compared with the base pipeline (Section 3.3 without RMSearch), the SEIMEI pipeline augmented by RMSearch shows consistent gains. Accuracy improves by roughly 7% in each iteration, and by the third iteration the cumulative improvement exceeds 20%. This suggests that better retrieval and reranking translate into tangible improvements in end-to-end repair performance.

4.3. Human Evaluation

4.4. GPU and Cost

We used the following hardware:

- `gpt-oss-20b` inference: RTX A5000 (24 GB VRAM)
- `qwen4b-reward` search: RTX A5000 (24 GB VRAM)
- `qwen4b-reward` training: A40 (48 GB VRAM)

Per iteration, the runtime and cost were approximately:

- ~6 hours for knowledge sampling (2x RTX A5000)
- ~6 hours for RMSearch training (1x A40)

The total cost was about $6 per iteration using a Runpod rental server.

---

5. Discussion

5.1. Interpretation of Experimental Reaults

The results are promising because they show measurable improvement from a single domain-specific codebase without retraining the base LLM. The gains suggest that RMSearch does more than refine existing model outputs: by selecting task-relevant knowledge, it injects information the base model would not reliably recall on its own.

The iterative improvement across three rounds indicates that the bootstrapping loop is effective: better reranking produces better inference traces, which in turn yields higher-quality preference data for the next round. Importantly, this improvement is achieved with relatively low compute compared to full language-model training, because RMSearch training focuses on reranking rather than next-token generation.

At the same time, these experiments are limited to one codebase and a break-and-repair task formulation. The results should therefore be interpreted as evidence that the approach is viable and cost-effective, but not yet as proof of broad generalization.

5.2. Unsuccessful Attempts

When generating dataset and sampling knowledge, we also encountered failures and setbacks along the way. We share our failure experiences here to provide insights, but this does not imply that these approaches are incapable of developing effective search models.

5.2.1. Monte Carlo Tree Search (MCTS)

In early experiments, we integrated MCTS to guide step-by-step knowledge selection. The approach did not work well because the reward signal at each step was too noisy; the search tended to optimize short-term signals without improving the final repair quality. As a result, the system failed to evolve meaningfully.

5.2.2. Knowledge Chunks Reward

To address the failure of step-wise rewards, we introduced "knowledge chunks," which are bundles of knowledge spanning multiple steps and generated by an LLM from a base inference trace. Scoring these chunks produced a more stable reward signal than step-level scoring. However, this method reduces flexibility: the knowledge is fixed for multiple steps and can conflict with the model's evolving needs during inference, which suppressed adaptive sampling and limited overall gains.

5.3. Possibility of Domain Specific LLM

Although we do not retrain the base LLM in this study, the results suggest a pathway toward near real-time learning in a domain. The key is to treat knowledge retrieval and reranking as the adaptive component: new repairs, discussions, or expert notes can be added to the knowledge pool immediately, and RMSearch can be updated with lightweight preference training. Because this training operates on a small reranker rather than the full LLM, it can be performed frequently and at low cost, enabling rapid adaptation to new code changes or newly discovered best practices.

In practice, a real-time learning loop would consist of (1) logging successful and failed repairs, (2) extracting reusable knowledge, and (3) periodically or continuously updating RMSearch. The base LLM remains stable, while the search model and knowledge pool evolve with the repository. This separation provides a form of real-time learning without the risks of catastrophic forgetting or the high compute cost of full model fine-tuning. Future work should test continuous updates and establish safeguards (e.g., validation gates) to prevent noisy or incorrect knowledge from degrading performance.

---

6. Conclusion

This thesis presented an LLM-assisted code repair pipeline for gyrokinetic plasma simulations. We constructed a break-and-repair dataset from the GKV codebase, implemented an agentic inference system (SEIMEI), and trained a reward-model-based reranker (RMSearch) using preference data derived from inference runs. Experiments demonstrated consistent improvements in reranking accuracy and end-to-end repair accuracy across iterations, with modest GPU cost.

The results indicate that domain-specific retrieval and lightweight preference training can enhance scientific code repair without retraining the base LLM. Future work includes expanding to additional codebases, refining evaluation beyond synthetic break-and-repair tasks, and improving automatic knowledge generation to reduce reliance on manual analysis.

References

1. Vaswani et al., *Attention Is All You Need*, NeurIPS 2017. (arXiv)
2. Kaplan et al., *Scaling Laws for Neural Language Models*, 2020. (arXiv)

3. Hoffmann et al., *Training Compute-Optimal Large Language Models*, NeurIPS 2022. (arXiv)
4. Brown et al., *Language Models are Few-Shot Learners*, NeurIPS 2020. (arXiv)
5. Ouyang et al., *Training language models to follow instructions with human feedback (InstructGPT)*, NeurIPS 2022. (arXiv)
6. Schulman et al., *Proximal Policy Optimization Algorithms*, 2017. (arXiv)
7. Christiano et al., *Deep Reinforcement Learning from Human Preferences*, NeurIPS 2017. (arXiv)
8. Ziegler et al., *Fine-Tuning Language Models from Human Preferences*, 2019. (arXiv)
9. Stiennon et al., *Learning to summarize from human feedback*, NeurIPS 2020. (arXiv)
10. Rafailov et al., *Direct Preference Optimization (DPO)*, 2023. (arXiv)
11. Wei et al., *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*, 2022. (OpenReview)
12. Wang et al., *Self-Consistency Improves Chain of Thought Reasoning in Language Models*, 2022. (Astrophysics Data System)
13. Yao et al., *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*, NeurIPS 2023. (OpenReview)
14. Yao et al., *ReAct: Synergizing Reasoning and Acting in Language Models*, ICLR 2023. (GitHub)
15. Schick et al., *Toolformer: Language Models Can Teach Themselves to Use Tools*, 2023. (arXiv)
16. Karpas et al., *MRKL Systems*, 2022. (arXiv)
17. Nakano et al., *WebGPT: Browser-assisted question-answering with human feedback*, 2021. (arXiv)
18. Yang et al., *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*, NeurIPS 2024. (arXiv)
19. Lewis et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, NeurIPS 2020. (arXiv)
20. Guu et al., *REALM: Retrieval-Augmented Language Model Pre-Training*, 2020. (arXiv)
21. Borgeaud et al., *Improving language models by retrieving from trillions of tokens (RETRO)*, ICML 2022. (arXiv)
22. Karpukhin et al., *Dense Passage Retrieval for Open-Domain QA*, EMNLP 2020. (arXiv)
23. Nogueira & Cho, *Passage Re-ranking with BERT (monoBERT)*, 2019. (arXiv)
24. Khattab & Zaharia, *ColBERT*, 2020. (arXiv)
25. Khattab et al., *DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines*, 2023. (arXiv)
26. Chen et al., *Evaluating Large Language Models Trained on Code (Codex)*, 2021. (arXiv)
27. GKV-developers, *gkvp: GyroKinetic Vlasov simulation code (repository)*. (GitHub)
28. Görler et al., *The global version of the gyrokinetic turbulence code GENE*, 2011. (MPG.PuRe)
29. Dorland et al., *Gyrokinetic Simulations of Tokamak Microturbulence* (GS2-related overview), 2000. (Princeton Plasma Physics Laboratory)
30. Holod et al., *Gyrokinetic particle simulations… (GTC-related)*, 2008. (AIP Publishing)
31. Nakata et al., *Validation studies of gyrokinetic ITG and TEM turbulence…*, *Nucl. Fusion* 2016. (NIFS Repository)
32. OpenHands, *OpenHands (coding agent platform, repository/site)*. (GitHub)
33. KyotoAI, *SEIMEI (repository)*. (GitHub)
34. KyotoAI, *RMSearch (repository)*. (GitHub)
35. DeepSeek, *DeepSeek-R1 (report/repository and coverage)*. (arXiv)