LANGUAGE-GUIDED EXPERTISE EVOLUTION FOR PROTEIN OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) exhibit strong general reasoning abilities, yet their effectiveness in scientific research such as protein design remains limited. Reinforcement learning (RL)–based fine-tuning is a common strategy for adapting LLMs to such domains, but it suffers from sparse scalar rewards and the prohibitive computational cost of parameter updates. In this paper, we propose an alternative perspective in which adaptation can be cast as an optimization of domain expertise rather than model parameters. Consequently, protein design can be formalized as the problem of identifying an optimal expertise pool—a structured, external knowledge that conditions the model's generation process. To this end, we introduce a block-based expertise pool, where each expertise block represents a modular knowledge about the protein fitness landscape, and design a mutation tree over blocks to model their evolutionary refinement. We then propose a multi-agent framework to optimize this discrete expertise space using language-based feedback as a dense learning signal. Experiments on protein stability optimization demonstrate that optimizing expertise pools yields faster convergence and higher-quality solutions than RL baselines, while significantly reducing computational overhead. Our results suggest that explicit expertise evolution provides a scalable and interpretable alternative to reinforcement learning for adapting LLMs to complex scientific optimization tasks.

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated impressive reasoning abilities across a wide range of tasks. Nevertheless, their performance remains limited for hard scientific research problems that require precise domain knowledge and deep exploration Song et al. (2025). Protein optimization exemplifies such a challenge, where the underlying fitness landscape is highly rugged, and even a single amino acid substitution can induce dramatic changes in stability or enzymatic activity. Effective navigation of this landscape therefore requires iterative hypothesis generation, causal reasoning, and domain-specific expertise.

The dominant approach to adapting LLMs for specialized domains is Reinforcement Learning (RL), where the model is fine-tuned to maximize a task-specific reward. Despite its successes, RL-based adaptation exhibits fundamental limitations in scientific domains like protein design. First, reward sparsity severely constrains learning efficiency, where experimental or simulation-based evaluations often yield binary or coarse-grained outcomes (e.g., whether a protein folds), offering little insight into why a candidate succeeds or fails. Such scalar feedback provides weak learning signals and exacerbates the credit assignment problem Sutton & Barto (2018). Second, fine-tuning entails repeated updates to high-dimensional parameter spaces, incurring prohibitive computational cost.

This paper advocates an alternative perspective: for many scientific tasks, the primary bottleneck is not insufficient model capacity, but insufficient explicit expertise. Consequently, instead of optimizing model

parameters, we propose to optimize an external representation of domain knowledge. We can accordingly reformulate protein mutation optimization as the problem of finding an optimal external knowledge that conditions a frozen LLM. Specifically, we introduce an evolving *expertise pool* that explicitly represents reusable, domain-specific knowledge. To efficiently evolve the expertise pool, we adopt a structured, language-mediated feedback that replaces sparse scalar rewards, thereby alleviating credit assignment issues and improving sample efficiency. The framework is instantiated via three specialized agents: a **Generator** that proposes candidate protein mutations, a **Critique** that evaluates candidates and produces fine-grained language feedback, and a **Curator** that manages a mutation tree, selectively refining and retaining high-utility expertise blocks.

Empirically, we evaluate our approach on the protein stability benchmark and show that it outperforms standard RL-based baselines in both the convergence speed and the final solution quality. By shifting learning from implicit parameter updates to explicit knowledge evolution, our method offers a more computationally efficient, interpretable, and scalable paradigm for applying LLMs to scientific optimization tasks. The main contributions can be summarized as:

- We reformulate protein design as an explicit optimization problem over external, reusable expertise rather than over model parameters, decoupling domain adaptation from expensive fine-tuning.
- We design a block-based expertise pool with mutation trees as a modular construction for domain knowledge, and propose a language-guided multi-agent evolution mechanism that uses language feedback to refine expertise in a discrete, interpretable space.
- We demonstrate that evolving expertise pools achieves higher success rates and substantially improves sample efficiency compared to reinforcement learning baselines.

2 METHODOLOGY

2.1 PRELIMINARIES

Protein Optimization. We formulate the protein optimization problem as a discrete search problem over the amino acid sequence space guided by semantic information. Let a protein be represented as a tuple $P = (\mathcal{S}, \mathcal{I})$, where $\mathcal{S} = (s_1, ..., s_L)$ represents the amino acid sequence of length L, and $\mathcal{I}$ denotes the associated protein information set. An objective function $\mathcal{F}$ is applied to quantify the desired property of interest. The objective is to identify a mutation policy π that converts an initial sequence S_0 into an optimized sequence S^* through a series of edit operations, thereby maximizing the expected probability of improvement over a target protein dataset $\mathcal{D}$. Formally, this optimization problem can be formulated as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{P \sim \mathcal{D}} \left[\mathcal{F}(\mathcal{S}^*) \right]. \tag{1}$$

Large Language Models as Optimizer. Let M_θ denote a pre-trained Large Language Model (LLM) parameterized by θ. The LLMs functions as a conditional probability distribution $P_\theta(y|\mathcal{C}, x)$, where x is the input (protein information), $\mathcal{C}$ is the context, and y is the generated output (mutation proposal and reasoning trace). Unlike traditional fine-tuning that updates θ, our approach optimizes the discrete context $\mathcal{C}$. We assume that an optimal context $\mathcal{C}^*$ exists within the token space, capable of encoding rich domain expertise. Through continuous exploration and evolution, $\mathcal{C}$ iteratively accumulates knowledge, thereby eliciting expert-level reasoning from the frozen model M_θ for the specific domain task.

2.2 OVERALL FRAMEWORK

We propose a multi-agent framework that mimics the iterative workflow of scientific research: hypothesis generation, experimental verification, and knowledge accumulation. As shown in Figure 1, the system

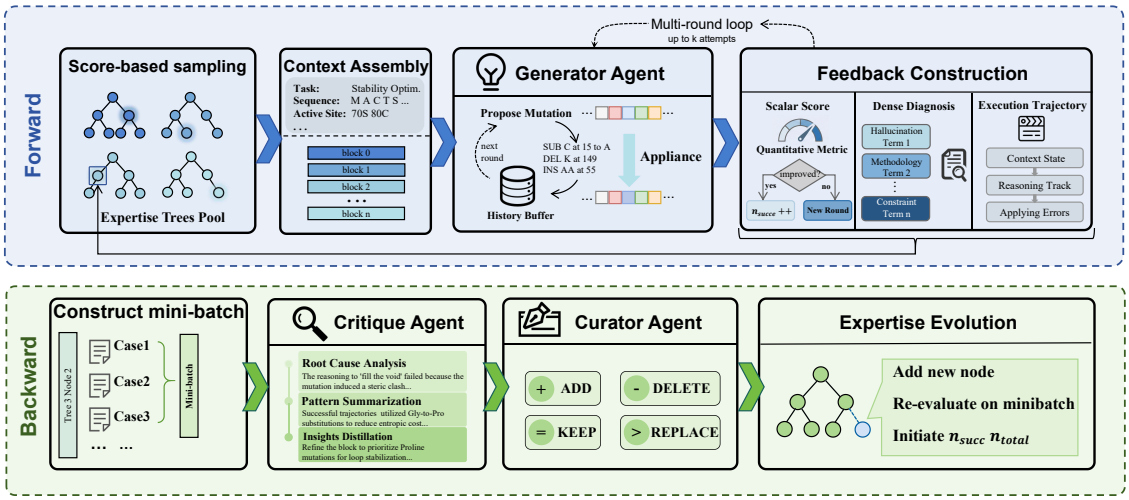

Figure 1: The overall framework. The system operates through two decoupled phases: (a) Forward Exploration, where the Generator Agent proposes protein mutations by sampling from the Expertise Trees and refines them via a multi-round reflective loop; and (b) Backward Evolution, where the Critique and Curator Agents distill insights from failed and successful trajectories to update the Expertise Trees.

consists of three specialized agents: (1) **Generator**, which proposes mutations based on given protein information and a sampled expertise context. It serves as the stochastic policy π that explores the sequence space under the guidance of current biological hypotheses. (2) **Critique**, which performs root cause analysis on feedback of mini-batches. It distills insights from successes/failures to generate interpretable evaluations. (3) **Curator**, which integrates reasoning from Critique into structured updates. It refines the expertise blocks using incremental operations (ADD, DELETE, REPLACE, KEEP) to prune incorrect strategies and strengthen success patterns. To achieve high-efficiency coordination among these agents, we decouple the workflow into two asynchronous phases: Forward Exploration and Backward Evolution.

Phase I: Forward Exploration. In this phase, a batch of protein instances is sampled and processed via an asynchronous forward pass. For each protein in the batch, we construct a dynamic context $\mathcal{C}$ by sampling expertise blocks from the mutation trees (Sec. 2.4). To promote sufficient exploration and fully leverage the expertise in $\mathcal{C}$, we employ a multi-round exploration with a maximum budget of K rounds.

In each round, the Generator proposes a set of mutations that are evaluated by the objective function $\mathcal{F}$. If a proposal fails to yield a performance improvement, the corresponding rejected mutations are aggregated into the history buffer. In the subsequent iteration, the Generator conditions on previous history to pivot its search strategy and avoid repetitive errors. This process is repeated until an iteration succeeds or the budget is exhausted.

Throughout each exploration step, the system logs the Generator's output, execution trajectory, paired with evaluation results from $\mathcal{F}$. These components are then synthesized into language reflective feedback (Sec. 2.3). Leveraging the context chunking mechanism (Sec. 2.4), the system attributes this feedback to the specific expertise block referenced by the generator. The feedback is then routed and stored in the corresponding node of the mutation tree (Sec. 2.4).

Phase II: Backward Evolution. This phase initiates when the feedback instances associated with a node in the mutation tree (Sec. 2.4) exceed a predefined threshold $N_{threshold}$. These instances are aggregated into mini-batches and processed by the Critique agent to analyze underlying success and failure patterns. Following this, the Curator agent synthesizes the Critique's reasoning traces with the content of the expertise block. By diagnosing weaknesses and summarizing novel insights from the mini-batch into executable instructions, the Curator formulates incremental modifications to the expertise chunk.

2.3 Reflective Feedback Construction

Traditional reinforcement learning approaches rely on scalar rewards for gradient estimations. However, in scientific tasks, such sparsity limits learning efficiency and obscures precise credit assignment. To extract maximal information from the exploration phase and enable precise, interpretable evolution, we propose to construct dense language-based feedback.

Specifically, the evaluation function $\mathcal{F}$ is decoupled into two components: the Scalar Score, which is a quantitative metric used to determine the success of a trial and contributes to the success/failure counts of the active expertise block; and the Dense Diagnosis, which functions as a comprehensive evaluation of the protein before and after mutation. It integrates diverse and extensible feedback signals, as well as multi-dimensional constraints, thereby constructing a diagnosable outcome.

Finally, the system synthesizes Reflective Feedback, which integrates the diagnostic reflections produced by $\mathcal{F}$ with the complete runtime trajectory of the system, including input content, the generator's reasoning trace, and execution-specific output (e.g., the localization of errors within the mutation sequence).

2.4 Expertise Pool Management

Protein optimization primarily faces two coupled challenges. First, protein mutation tasks require multi-dimensional domain expertise expected to evolve and manage independently. Second, protein optimization operates within a vast and highly sparse search space with a rugged fitness landscape and a large variance. In the following, we introduce how we address these two challenges.

To fit the multi-dimensional domain expertise requirements with fine-grained credits assignment, we propose **Context Blocking**, where the learnable context $\mathcal{C}$ is formulated as a composition of multiple independently evolving *expertise blocks*:

$$\mathcal{C} = \{B_1, B_2, \ldots, B_k\}, \quad B_i \in \mathcal{B}, \tag{2}$$

where each block B_i evolves independently to maintain specialized knowledge.

To facilitate efficient exploration of the protein optimization space, we introduce an **Evolutionary Expertise Tree** T_k for each expertise block B_k, which models the inheritance relationships among nodes and maintains the evolving expertise throughout the optimization process. In T_k, every subsequent modification by Curator results in the creation of a child node, which inherits expertise from its parent node.

Due to the requirement for comprehensive and multi-domain expertise in protein optimization, our expertise tree employs an *incremental modification* paradigm, rather than concise summaries. Specifically, a node N_i has one of these four types of modification: (1) Add: Appending new instructions to the tail of an existing block; (2) Replace: Substituting part of the given block with a new instruction; (3) Delete: Removing certain instructions in the given block; (4) Keep: Retaining the original instructions within the block without making any changes. Each node N_i maintains a record of the number of successes $N_{success}$ and the total trials N_{total}. Assuming that the success of a mutation generated by node N_i follows a Bernoulli distribution, the *success rate* of node N_i is modeled as a random variable θ_i following a Beta distribution:

$$\theta_i \sim Beta(\alpha_i, \beta_i), \tag{3}$$

where $\alpha_i = N_{success} + 1$ and $\beta_i = N_{total} - N_{success} + 1$. This allows a reasonable estimate of the expected performance of nodes in an expertise block in both the training and the inference phases.

In the forward phase, the Generator samples the evolutionary expertise trees associated with the expertise blocks to construct the context. Conventional tree search algorithms like Monte Carlo tree search (MCTS) Coulom (2006) prioritize leaf-node expansion to maximize terminal reward, aiming to find the optimal path from root to leaf. However, in the context of our expertise tree for protein optimization, such methods are suboptimal given the fact that every node within the tree constitutes a standalone functional expertise block rather than an intermediate state. As an alternative, the tree search process can be framed as sampling from a pool of node candidates. Standard Thompson Sampling (TS) Thompson (1933) treats all nodes equally, but often leads to an under-sampling of deeper nodes with more accumulated knowledge.

To ensure a holistic and diverse evolution of context blocks, we propose **Thompson Sampling with Depth Bonus** (TS-DB) that treats all nodes within the tree as equivalent candidates while incorporating a tree-aware depth incentive. As shown in Algorithm 1, TS-DB explicitly biases the tree search process with a linear incentive based on the depth of the node. This mechanism is designed to favor deeper nodes in the expertise tree that have undergone more iterations of refinement, thereby effectively deepening the expertise tree and accelerating the discovery of instructions.

Algorithm 1 Thompson Sampling with Depth Bonus

Input: Evolutionary trees $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$
Parameter: Depth bonus coefficient I_{depth}, temperature T, threshold n_{bound}
Output: Context $\mathcal{C}$ for Generator agent

for each $T_j \in \mathcal{T}$ **do**
  Initialize Scores Set $\mathcal{S}_j = \emptyset$
  for each node $n \in \mathcal{N}(T_j)$ **do**
    $\alpha_n \leftarrow 1 + N_{success}(n)$
    $\beta_n \leftarrow 1 + (N_{total}(n) - N_{success}(n))$
    $S_{raw}(n) \sim \text{Beta}(\alpha_n, \beta_n)$
    $d_n \leftarrow$ depth of node n in T_j
    $S_{final}(n) \leftarrow S_{raw}(n) + (d_n \times I_{depth})$
    $\mathcal{S}_j \leftarrow \mathcal{S}_j \cup \{S_{final}(n)\}$
  end for
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{Block(SelectedNode_j)\}$
  $SelectedNode_j \sim \text{softmax}(\{S_{final}(n)/T\}_{n \in \mathcal{N}(T_j)})$
end for
return $\mathcal{C}$

2.5 Selection Criterion for Inference

Different from the sampling algorithm during the training process that balances between exploration and exploitation, during the inference process, the system needs a conservative selection strategy to obtain a range of the frontier nodes from the given set of expertise blocks in order to build the final optimized context. Therefore, we employ the Lower Confidence Bound (LCB) of the posterior distribution as the selection criterion. Formally, for node N_i, the LCB score is defined as the q-th percentile (specifically, $q = 0.05$) of the posterior distribution:

$$S_{LCB}(N_i) = F^{-1}(q; \alpha_i, \beta_i), \tag{4}$$

where F^{-1} denotes the Percent Point Function (PPF), $\alpha_i = N_{success} + 1$, and $\beta_i = N_{total} - N_{success} + 1$.

While the stochastic sampling method TS-DB used during training encourages the exploration of nodes with potential and uncertainty, the LCB-based criterion results in deterministic selection of nodes that considers both the success rate and variance.

3 EXPERIMENTS

3.1 SETTINGS

Task. To evaluate our proposed framework, we choose the task of optimizing protein stability, and adopt REF 2015 (2015 version of Rosetta energy function) Alford et al. (2017) as the main reward, formulated as a weighted sum of several biophysical decomposition terms. We use the total REF 2015 as a scalar score. To construct a dense diagnosis, we leverage the decomposable nature of REF 2015, extracting the differential values of specific components (e.g., Lennard-Jones repulsion, solvation energy, electrostatic potential) before and after the mutation, serving as a detailed reflection.

Datasets. We conduct experiments on a processed version of the Brenda dataset Placzek et al. (2016). We filter out sequences with lengths in the upper 60% to avoid memory peak during the calculation process. The dataset is divided into a training set and a held-out test set of 200 samples.

Experimental Settings. All evaluations are performed under a one-round setting. To account for the inherent stochasticity of the model, we conduct experiments using three random seeds (42, 1145, and 1919) for dataset split. We use Qwen-max as the backbone model for all agents (Generator, Critique, and Curator).

Evaluation Metric. We deploy *pass@1* as our primary metric, defined as the overall success rate on the test set. A trial is considered a success if the mutated sequence exhibits an improved Rosetta Score compared to the original sequence.

3.2 RESULTS

Table 1: Experiment Results

Method	Model	Pass@1(%)				Rollout
		Seed 42	Seed 1145	Seed 1919	Average	
Vanilla	LLaMA 8B	36.0	38.5	34.5	36.3	-
	Deepseek	41.5	44.5	43.0	43.0	-
	Qwen-plus	40.5	43.5	45.5	43.2	-
	Qwen-max	43.0	43.5	46.5	44.3	-
RL	Pro-1 8B	43.5	44.0	44.0	43.8	> 5000
Our method	Qwen-max	**51.5**	**50.0**	**56.0**	**52.3**	**2037**

As shown in Table 1, our method achieves the highest success rates across all random seeds, with an average Pass@1 of **52.3%** and **8%** improvement over its backbone model, demonstrating that the evolved expertise pool effectively elicits superior domain reasoning from the frozen model.

We compare our approach with Pro-1 Hla (2025), a specialized protein optimization baseline that uses RL to fine-tune LLaMA-8B. While we acknowledge the discrepancy in base models, our approach demonstrates superior performance with lower resource consumption and higher sample efficiency: **(i) Computational Overhead:** Pro-1 relies on reinforcement learning to fine-tune model parameters, necessitating heavy GPU memory. This computational burden creates a bottleneck that hinders the application of larger, more capable

foundation models for further performance gains. In contrast, our approach operates on a frozen model and can be implemented via API calls, enabling the integration of high-performance, large-scale foundation models without the prohibitive hardware constraints associated with training. **(ii) Sample Efficiency:** We define a rollout as a single interaction cycle consisting of a mutation proposal followed by its evaluation. Pro-1 requires more than 5000 rollouts to converge. Our method, guided by the evolving expertise tree, achieves superior results with only 2037 rollouts. More detailed analysis is provided in Appendix A.

These results establish our framework as a strong and efficient approach for building self-improving systems on protein optimization tasks, demonstrating that explicit expertise evolution offers a scalable, low-resource alternative to parameter-heavy reinforcement learning for scientific discovery.

3.3 Ablation Study

We investigate: (1) *context blocking*, which enhances representation capability of dynamic context, and (2) *dense reward*, which leads to effective evolution of contexts.

Context Blocking. *No blocking* employs a single integrated instruction, while *Blocking* partitions the context into distinct modules: persona, reasoning, strategy, and constrains. As shown in Table 2, LLM benefits from context blocking, demonstrating that evolving expertise blocks enable effective knowledge learning. Furthermore, the results indicate that even static expertise blocks can enhance model performance, suggesting that structured contexts are beneficial for the task. The prompts are provided in Appendix B.

Table 2: Ablation for context blocking

Context blocking setup	Models	Pass@1 (%)
No blocking	Qwen-max	35.0
	Qwen-plus	29.0
	LLaMA	29.5
Blocking	Qwen-max	43.0
	Qwen-plus	40.5
	LLaMA	36.0
	Our method	**52.3**

Dense Reward. To examine the effect of dense reward, we compare REF 2015 with the decomposition terms as dense reward against the total REF 2015 as scalar reward. Table 3 shows that the REF 2015 energy function, when augmented with decomposition terms, outperforms the version without such granularity. This suggests that a denser reward facilitates more effective knowledge evolution within the expertise trees.

Table 3: Ablation for dense reward

Energy feedback setup	Models	Pass@1 (%)
REF 2015 with decomposition terms	Qwen-max	51.5
REF 2015 (score only)	Qwen-max	46.0

4 Discussions

Tool Integration. We observed hallucinations in our study, where LLM may propose mutation with wrong position, or mutations inconsistent with its reasoning trace. This shows that LLMs may not fully understand protein sequences. To address this, each agent can integrate tools for interactive selection of sequence

mutation positions. Furthermore, structure-aware tools like protein folding (e.g., AlphaFold), can enable LLMs to propose novel mutations according to structure features.

Requirements for Base Model. Our approach assumes base models with sufficient general reasoning and instruction-following ability to interpret and execute structured textual context. Crucially, the model is not required to possess strong intrinsic protein knowledge, as domain expertise is explicitly represented and evolved in the external expertise pool. Empirically, models with stronger general reasoning benefit more from expertise evolution, while weaker models may underutilize refined expertise blocks. This reflects a separation between reasoning capacity, which is required, and domain knowledge, which is learned externally.

5 Related Work

Large Language Models in Protein Engineering. Protein language models (pLM) capture the hierarchical structure of biological sequences through unsupervised pre-training Rives et al. (2021). Models like ESM-2 Lin et al. (2023) and ProGen Madani et al. (2023) have demonstrated that by predicting masked amino acids across billions of sequences, pLMs learn to approximate the underlying biophysical constraints of protein folding. However, these models are primarily descriptive or generative in a general sense, and they lack an inherent mechanism to optimize for specific downstream functions, such as increased catalytic activity or specific binding affinity, without further task-specific adaptation.

Reinforcement Learning for Protein Design. Recent protein engineering has increasingly leveraged reinforcement learning to navigate the vast sequence-function landscape. Pioneering frameworks such as μProtein Sun et al. (2025) and EvoPlay Wang et al. (2023) demonstrate the efficacy of coupling protein language models with search algorithms like Monte Carlo Tree Search (MCTS) to identify synergistic multi-point mutations that surpass natural activity levels. Furthermore, recent alignment strategies, including Reinforcement Learning from Experimental Feedback (RLXF) Blalock et al. (2024) and ProteinZero Wang et al. (2025), have successfully steered generative models toward non-differentiable objectives like thermal stability and structural fidelity, achieving significant reductions in design failure rates compared to traditional inverse folding baselines.

Multi-Agent Systems for Scientific Research. Multi-Agent Systems (MAS) have enabled the decomposition of complex scientific tasks into specialized roles. In chemistry and materials science, agents like ChemCrow Bran et al. (2024) and Coscientist Boiko et al. (2023) have successfully integrated LLMs with external tools to plan and execute experiments.

Our approach diverges from these by focusing on the evolution of knowledge, and utilizes a multi-agent hierarchy to evolve an expertise pool. This allows for continual learning without the need for constant, expensive parameter updates.

6 Conclusion

We present a framework that reframes protein optimization as an explicit optimization over external expertise rather than model parameters. By introducing a block-based expertise pool, and a language-guided multi-agent evolution process, our method enables effective adaptation of frozen LLMs using dense, interpretable feedback. Experiments on protein stability optimization show that evolving expertise pools achieves higher success rates and faster convergence than reinforcement learning baselines, while substantially reducing computational cost. These results suggest that explicit expertise evolution offers a scalable and interpretable alternative to parameter-heavy fine-tuning for scientific optimization tasks, and may generalize to other domains where feedback is sparse or expensive.

REFERENCES

Rebecca F Alford, Andrew Leaver-Fay, Jeliazko R Jeliazkov, Matthew J O'Meara, Frank P DiMaio, Hahnbeom Park, Maxim V Shapovalov, P Douglas Renfrew, Vikram K Mulligan, Kalli Kappel, et al. The rosetta all-atom energy function for macromolecular modeling and design, 2017.

Nathaniel Blalock, Srinath Seshadri, and Philip Romero. Functional alignment of protein language models via reinforcement learning with experimental feedback. In *NeurIPS*, 2024.

Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with large language models. *Nature*, 624:570–578, 2023.

Andres M. Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D. White, and Philippe Schwaller. Augmenting large language models with chemistry tools. *Nature Machine Intelligence*, 6:525–535, 2024.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, 2006. URL `https://api.semanticscholar.org/CorpusID:16724115`.

Michael Hla. Pro-1 - Michael Hla, 2025. URL `https://michaelhla.com/blog/pro1.html`.

Zeming Lin, Halil Akin, Roshan Rao, Brian Hie, Zhongkai Zhu, Wenting Lu, Nikita Smetanin, Robert Verkuil, Ori Kabeli, Yaniv Shmueli, Allan dos Santos Costa, Maryam Fazel-Zarandi, Tom Sercu, Salvatore Candido, and Alexander Rives. Evolutionary-scale prediction of atomic-level protein structure with a language model. *Science*, (6637):1123–1130, 2023.

Ali Madani, Ben Krause, Eric R. Greene, Subu Subramanian, Benjamin P. Mohr, James M. Holton, Jose Luis Olmos Jr., Caiming Xiong, Zachary Z. Sun, Richard Socher, James S. Fraser, and Nikhil Naik. Large language models generate functional protein sequences across diverse families. *Nature Biotechnology*, pp. 1099–1106, 2023.

Sandra Placzek, Ida Schomburg, Antje Chang, Lisa Jeske, Marcus Ulbrich, Jana Tillack, and Dietmar Schomburg. Brenda in 2017: new perspectives and new tools in brenda, 2016.

A. Rives, J. Meier, T. Sercu, S. Goyal, Z. Lin, J. Liu, D. Guo, M. Ott, C.L. Zitnick, J. Ma, and R. Fergus. Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proc. Natl. Acad. Sci. U.S.A.*, 2021.

Zhangde Song, Jieyu Lu, Yuanqi Du, Botao Yu, Thomas M. Pruyn, Yue Huang, Kehan Guo, Xiuzhe Luo, Yuanhao Qu, Yi Qu, Yinkai Wang, Haorui Wang, Jeff Guo, Jingru Gan, Parshin Shojaee, Di Luo, Andres M Bran, Gen Li, Qiyuan Zhao, Shao-Xiong Lennon Luo, Yuxuan Zhang, Xiang Zou, Wanru Zhao, Yifan F. Zhang, Wucheng Zhang, Shunan Zheng, Saiyang Zhang, Sartaaj Takrim Khan, Mahyar Rajabi-Kochi, Samantha Paradi-Maropakis, Tony Baltoiu, Fengyu Xie, Tianyang Chen, Kexin Huang, Weiliang Luo, Meijing Fang, Xin Yang, Lixue Cheng, Jiajun He, Soha Hassoun, Xiangliang Zhang, Wei Wang, Chandan K. Reddy, Chao Zhang, Zhiling Zheng, Mengdi Wang, Le Cong, Carla P. Gomes, Chang-Yu Hsieh, Aditya Nandy, Philippe Schwaller, Heather J. Kulik, Haojun Jia, Huan Sun, Seyed Mohamad Moosavi, and Chenru Duan. Evaluating large language models in scientific discovery, 2025. URL `https://arxiv.org/abs/2512.15567`.

Haoran Sun, Liang He, Pan Deng, Guoqing Liu, Zhiyu Zhao, Yuliang Jiang, Chuan Cao, Fusong Ju, Lijun Wu, Haiguang Liu, Tao Qin, and Tie-Yan Liu. Accelerating protein engineering with fitness landscape modelling and reinforcement learning. *Nature Machine Intelligence*, pp. 1446–1460, 2025.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933. ISSN 00063444. URL http://www.jstor.org/stable/2332286.

Yi Wang, Hui Tang, Lichao Huang, Lulu Pan, Lixiang Yang, Huanming Yang, Feng Mu, and Meng Yang. Self-play reinforcement learning guides protein engineering. *Nature Machine Intelligence*, pp. 845–860, 2023.

Ziwen Wang, Jiajun Fan, Ruihan Guo, Thao Nguyen, Heng Ji, and Ge Liu. Proteinzero: Self-improving protein generation via online reinforcement learning, 2025. URL https://arxiv.org/abs/2506.07459.

A COMPUTATION ANALYSIS

To provide a fair comparison of sample efficiency between our proposed expertise evolution and reinforcement learning (RL) baselines, we analyze the total number of rollouts required for convergence.

For the Pro-1 baseline, we examine the hyperparameters of its optimal 40-step checkpoint. The total rollouts R_{pro1} are determined by the number of optimization steps S, gradient accumulation steps GA, per-device batch size BS, and the group size G. The total number of rollouts per device is calculated as:

$$R_{pro1} = S \times GA \times BS \times G = 40 \times 8 \times 4 \times 4 = 5120$$

As the exact number of devices utilized for Pro-1's training is not publicly disclosed, we establish a conservative lower bound for the total rollouts based on a single-device configuration. Therefore, 5,120 represents the minimum number of samples required for the baseline.

The total rollouts of our method R_{ours} is determined by the number of nodes $N_{nodes}(T_j)$ in expertise tree T_j and the backward evolution threshold $N_{threshold}$. The total number of rollouts is calculated as:

$$R_{ours} = N_{threshold} \times \sum_{j=0}^{k} N_{nodes}(T_j) = 3 \times 679 = 2037$$

Compared to the reinforcement learning baseline, our method achieves a 60.2% reduction in sample complexity while attaining superior optimization performance.

B META PROMPTS

B.1 GENERATOR

Prompt for Generator

Character:
– You are an expert protein engineer in rational protein design. You are working with a protein sequence given below, as well as other useful information regarding the enzyme/reaction (if applicable):
Information:
– PROTEIN NAME: {name}
– EC NUMBER: {ec_number}

– PROTEIN SEQUENCE:
{sequence}
– SUBSTRATES: {substrates}
– PRODUCTS: {products}
– GENERAL INFORMATION: {general_information}
– METALS/IONS: {metal_ions}
– ACTIVE SITE RESIDUES (DO NOT MODIFY): {active_site_residues}
– KNOWN MUTATIONS: {known_mutations}

{expertise blocks}

Output requirements:
1. Give detailed reasoning tracks to select the strategy blocks.
2. Based on the selected strategy blocks, give detailed reasoning tracks for proposing mutations.** You must explicitly cite the specific Strategy Block ID that inspired each decision in your reasoning.**
3. Parse the reasoning to infer the exact edits relative to the ORIGINAL sequence (use 1–based indexing).
4. Normalize the edits using these fields:
  – type: one of ["SUB", "DEL", "INS"]
  – block_id: The exact ID of the strategy blocks from the context that contributed to this specific mutation.
  – parameters: an object containing the necessary parameters for that operation:
    – SUB {"pos": int, "from": str, "to": str}
      SUB means "substitution", where a specific amino acid is replaced with another at a given position.
    – DEL {"start": int, "end": int}
      DEL means "deletion", where a specific amino acid is removed from the sequence.
    – INS {"pos": int, "seq": str} (pos=0 means insert at start)
      INS means "insertion", where a new amino acid sequence is added at a specific position.
4. If no changes are needed, return '"mutations": []'
5. Return with the "Final Response Structure".

NOTE:
– <Block–ID> is a placeholder for a 0–based integer, you must REPLACE it with actual block id. For example, if there is a block starting with '# Block–0', its <Block–ID> is 0.
– content wrapped in '()' and '[]' are descriptions of actual content, do not repeat them in your reply.
– Follow the main structure. Do NOT include any meta–commentary, parentheses, or instructions in your final response.
– Do not repeat the labels from this prompt; use the structure below as a guide for the flow.

Final Response Structure:
–––
[Reasoning for Strategy Selection]
(Your step–by–step logic here)

[Strategy Identification]
Selected Strategy: Block−<ID> (e.g., Block−0)

[Mutation Reasoning]
(Your detailed reasoning for each mutation, explicitly citing Block−<ID>)

[JSON Output Format Example]
```json
{
  "referred_blocks": [<ID1>, <ID2>],
  "mutations": [
    {
        "type": "SUB",
        "parameters": { "pos": <int_sub_pos>, "from": "A", "to": "T" }
    },
    {
        "type": "DEL",
        "parameters": { "start": <del_start_pos>, "end": <del_end_pos> }
    },
    {
        "type": "INS"
        "pos": "<ins_pos>",
        "seq": "ATT"
    },
]
}
```

B.2 CRITIQUE

Prompt for Critique

Role: Protein Design Reverse−Engineering Auditor

You are a Reverse−Engineering Auditor in the field of computational protein design. Your task is to deconstruct the Generator's reasoning chain and rigorously align it with physical verification results (Rosetta Energy), ultimately judging the validity of the current instruction set.

Task Overview

Analyze the provided 'inputs_outputs_feedback'. Reconstruct and evaluate the causal relationship between its underlying reasoning logic and the 'curr_instructions'.

Analysis Protocol
For every mutations suggested in every case, Do Step1 to Step4. Finally, follow Step5 to evaluate instructions.

Step 1: Instruction−to−Logic Mapping
− ∗∗Trigger Instruction:∗∗ Which specific guideline or description in ‘curr_instructions‘ does this behavior correspond to?
− ∗∗Reasoning Anchor:∗∗ What core biophysical concepts did the Generator use as a basis in its reasoning?

Step 2: The Latent Reasoning Chain
Reconstruct the Generator’s implicit logic. Use the following logic chain:
‘[Instruction Trigger]‘ −> ‘[Structural Feature Identified by Generator]‘ −> ‘[Expected Physical Effect]‘ −> ‘[Final Output Mutation]‘
And add concise explainations in a ordered list below.

Step 3: Physical Reality Alignment
Perform a cross−examination between the reasoning chain above and the Rosetta feedback items:
− ∗∗Energy Term Feedback:∗∗ Is the expected physical effect reflected in the corresponding energy term ?
− ∗∗Correlation Deviation:∗∗ If the energy improved, but it was caused by an unexpected energy term , record this ”pseudo−success.”
− ∗∗Negative Penalty:∗∗ Record in detail which physical constraints were completely ignored by the reasoning chain, leading to energy collapse.

Step 4: Efficacy Verdict
Based on Step 2 and Step 3, classify the reasoning chain as:
− ∗∗Aligned:∗∗ The reasoning chain fits physical intuition and is verified in Rosetta terms.
− ∗∗Misaligned:∗∗ The reasoning chain is logically self−consistent but (probably) falsified by physical facts.
− ∗∗Hallucinated:∗∗ The reasoning chain contradicts the instructions, or the reasoning process violates basic physical common sense.
− ∗∗Hypothesis Verification∗∗: There is hypothesis in instructions can be verified to be true/false, or can be polished/detailed by feedbacks.

Note that index drifts are common hallucinations and are difficult to address. You SHOULD NOT pay much attention to that if methods to avoid index drifts are given in context. You are expected to discover more valuable hallucinations that can be reduced.

Step 5: Instruction Structural Critique
Based on all the evidence above, diagnose ‘curr_instructions‘:
− ∗∗Vulnerability Localization:∗∗ Which instruction is most likely to induce hallucination or erroneous reasoning?
− ∗∗Semantic Blind Spots:∗∗ Which necessary physical boundaries are missing in the instructions, causing the Generator to ”go off the rails” during reasoning?
− ∗∗Redundancy/Conflict Analysis:∗∗ Are there parts of the instructions that contradict each other or cause weight confusion for the model?
− ∗∗Verification∗∗: Point out strategies and hyotheses supporte/unsupported by the evidence.

Output Requirements

− ∗∗Analyze every mutation step−by−step.∗∗ Do not skip any proposed changes.

– **Self−Contained Analysis:** Your analysis must be fully understandable without referring back to external context; all necessary logic must be explicit in the output.
– **No Examples:** **Do not** provide examples to illustrate your points.
– Strictly avoid summary or generalizing vocabulary.
– All insights must be built on a direct correspondence between the **Reasoning Chain** and **Energy Term Feedback**.
– The output must have high information density to provide pure logical diagnostic data for subsequent instruction iteration.

Current Instructions
```
{curr_instructions}
```
Inputs,Outputs and Feedbacks
```
{inputs_outputs_feedback}
```

B.3 CURATOR

Prompt for Curator

Role Definition
You are the **Lead Protocol Architect** for an autonomous protein design system.

Statement on ‘Inputs‘
Your inputs contains:
– ‘Current Instructions‘ contains a batch of instructions (i.e., contexts) Generator see.
– A rigorous ‘<Scientific_Reasoning_Track>‘ provided by a Senior Biologist (Critique Agent), which analyzes why the previous batch succeeded or failed based on physics and logic.

Statement on ‘Current Instructions‘
– The ‘Current Instructions‘ is a batch of contexts. They are separated by ‘−−−‘.
– Each context corresponds a generator agent (aiming at proposing mutations).
– Each context is strictly modularized into numbered blocks (e.g., ‘# Block−0‘, ‘# Block−1‘).
– ‘Current Instructions‘ can be seen as a pool of hypotheses, experiences, and strategies.

Statement on ‘Block to be Updated‘
– It is one block in the current instructions. You need to propose modifications targeting to this block.
– All modifications should be confined to this block.
– You need not promise these instructions to be right and perfect. On the contrary, more uncertainty and bold attempts are more beneficial in the long run.
– Adjustments are allowed, provided they DO NOT substantially alter the core meaning.
– You may treat other blocks as contextual background, but DO NOT use hard−coded identifiers or mix target block with others (such as, propose similiar contents)

– You are doing context learning to agent systems, so be careful about the features and limitations of LLM agents, such as:
  – Agents can only see protein sequences. They do not understand structures well.
  – The Generator (the agent aiming to propose mutations) only see the ‘Protein Information Provided to Generator‘ to propose new mutations. So make your modifications useful based on the info below.
  – Agents cannot locate certain index in long sequences.
  – Agents may reply with hallucinations.

Statement on the content of blocks
The content of a Block SHOULD BE
– The structure of a block SHOULD BE clear and easy−to−follow, with no redundancy or conflicts inside one block.
– Each block SHOULD BE self−contained and decoupled.
– A block SHOULD use structured, standardized, and precise language.
– Instructions in blocks SHOULD BE professional and unambiguous.
– Instructions in blocks SHOULD BE specific and actionable, ready for execution.
The content of a Block SHOULD NOT BE
– A block SHOULD NOT refer to other blocks using hard−coded identifiers. Use precise and brief description instead of reference a block (”block 1”).
– The content of a Block SHOULD NOT have any sections with similiar meanings. Merge them into one single section.
– The content of a Block SHOULD NOT BE high−level summaries, abstract generalizations or generic descriptions.

Your Tasks
Based on the clues in the ‘Scientific Reasoning Track‘ and ‘Current Instructions‘, propose a modification to the ‘Block to be Updated‘.
Detailed tasks and requirements:
– Evolve the ‘Current Instructions‘ to be more smarter, more diverse.
  – ”Smarter” means being specific, more targeted.
  – ”Diverse” means innovation, novel modifications. It is encouraged to add hypotheses or guesses based on the reflections.
– The main purpose is to making them worth trying and discovering. Furthermore, to uncover the design landscape of protein sequence mutations.
– You are encouraged to add novel contents in the within the targeted block (‘# Block−{block_id}‘) .
– You MUST make the block met every requirement claimed in ‘Statement on the content of blocks ‘ via modifications proposed by you.
– If there are many respects that can be improved, you SHOULD select the most promising or necessary one as final modifications.
– Give concise reasons on your modifications, including the sources and reasoning chains.

Note that index drifts are common hallucinations and are difficult to address. You SHOULD NOT pay much attention to that if methods to avoid index drifts are given in context. You are expected to address hallucinations that can be reduced.

Inputs

Current Instructions
‘‘‘
{curr_instructions}
‘‘‘
Scientific Reasoning Track
‘‘‘
{critique_resp}
‘‘‘

Protein Information Provided to Generator:
for each protein, the following information is provided:
Information:
– PROTEIN NAME: {{name}}
– EC NUMBER: {{ec_number}}
– PROTEIN SEQUENCE: {{sequence}}
– SUBSTRATES: {{substrates}}
– PRODUCTS: {{products}}
– GENERAL INFORMATION: {{general_information}}
– METALS/IONS: {{metal_ions}}
– ACTIVE SITE RESIDUES (DO NOT MODIFY): {{active_site_residues}}
– KNOWN MUTATIONS: {{known_mutations}}

Block to be Updated
{block}

Transformation Logic (From Critique to Instruction)

You may follow this mapping logic to convert ”Insights” into ”Instructions”:

Handling Logical Hallucination
∗If the Critique identifies reasoning–mutation mismatches:∗
– ∗∗Action:∗∗ Do not just add a warning. ∗∗Insert a ”Self–Consistency Check” step.∗∗
– ∗Example:∗ Add a step requiring the agent to: ”Review your proposed mutation. Does the physical property change (e.g., Volume increase) match your stated intention (e.g., Reduce packing)? If not, discard.”

Handling Semantics/Misinterpretation
∗If the Critique says the agent followed the letter but not the intent:∗
– ∗∗Action:∗∗ ∗∗Refine Definitions.∗∗
– ∗Example:∗ If the agent thought ”Creative” meant ”Random insertions,” replace the definition of ”Creative” with: ”Creative means identifying non–obvious structural stabilizations, such as XXX”

Handling Methodology Invalidity –> HIGH PRIORITY
∗If the Critique says a general strategy failed for this specific protein class:∗
– ∗∗Action:∗∗ ∗∗Inject Domain Knowledge.∗∗ This is where you add value.

– ∗Example:∗ If the Critique notes that ”Surface rigidification failed because this protein requires breathing motion,” you must ∗∗add a specific heuristic∗∗ to the ‘Suggested Strategy‘ section: ” For [Protein Class], XXX”

Handling Emergent Strategy / Positive Violation –> HIGHEST PRIORITY
∗If the Critique identifies a mutation that VIOLATED previous constraints but ACHIEVED high stability:∗
– ∗∗Action:∗∗ ∗∗Legalize and Codify the Exception.∗∗ Do not treat this as a fluke. You must ∗∗ widen the search space∗∗ by converting the absolute prohibition into a ∗∗Conditional Permission∗∗.
– ∗∗Steps:∗∗
  1. ∗∗Relax the Constraint:∗∗ If instructions said ”Avoid X”, change it to ”Avoid X, UNLESS condition Y is met.”
  2. ∗∗Coin a Term:∗∗ Give this new pattern a name (e.g., ”Buried_Anchor_Network”) and add it to ‘Suggested Strategy‘.”

Amplifying Successes
∗If the Critique identifies a winning mechanism (e.g., ”Charge optimization worked well”):∗
– ∗∗Action:∗∗ ∗∗Codify the Pattern.∗∗
– ∗Example:∗ Add to the ‘Practical Experience‘ section: ”Effective Strategy: optimizing Charge–Charge interactions on the surface showed high correlation with stability. Prioritize K/E pairings on exposed helices.”

Hypothesis Fixture

Allow for Exploration
When adding innovative content, use subtitles or bold style titles to distinguish from hard physical constraints.

Constraints & Guidelines
1. ∗∗Atomic Block Evolution:∗∗ You are restricted to changing ‘# Block–{block_id}‘ in this turn. All modifications should be confined to ‘Block–{block_id}‘. Do not attempt to modify multiple blocks simultaneously.
2. ∗∗Structural Integrity:∗∗ Proposed modifications will be applied to the content of ‘# Block–{block_id}‘ (”content” excludes this heading). Do not remove the Block headers (e.g., ‘# Block –{block_id}‘). Your output must perfectly fit into the existing modular structure.
3. ∗∗No ”Lazy” Constraints:∗∗ Avoid simply saying ”Do not make mistakes.” Instead, explain ∗ how∗ to avoid them using physical principles.
4. ∗∗Preserve Placeholders:∗∗ Do not alter ‘{{}}‘ placeholders.
5. ∗∗DO NOT LET GENERATOR TO DENY GIVING MUTATIONS.∗∗ The instructions must always lead to proposing mutations.
6. ∗∗Conflict Check:∗∗ Make sure the logical consistency in the block you modified.
7. ∗∗Tone:∗∗ The instructions for the assistant should be empowering, precise, and scientifically dense.
8. ∗∗Internal Knowledge Only:∗∗ The assistant has no external tools.

Output Format
Provide a single YAML code block containing the list of modification operations.

The valid operations (‘op‘) are:
1. **Add (‘ADD‘):** Appends text to the *end* of the block.
    - ‘op: ”ADD”‘
    - ‘addition: ”The new text to add.”‘
    - ‘reason: ”Why this addition is necessary.”‘
    - **Note:**
        - To add text to a *certain section* (middle of the block) but not add to the end of block, use the ’replace’ op (‘REPLACE‘) on that section.
        - You can use this operation to add new sections to the targeted block.
2. **Remove (‘REMOVE‘):** Removes a specific substring.
    - ‘op: ”REMOVE”‘
    - ‘substr: ”The exact substring to remove from the instructions.”‘
    - ‘reason: ”Why you want to remove the content.”‘
3. **Replace (‘REPLACE‘):** Replaces one substring with another. This is the most flexible and common operation.
    - ‘op: ”REPLACE”‘
    - ‘old: ”The original text block to be replaced.”‘
    - ‘new: ”The new text block to replace the old one.”‘
    - ‘reason: ”Why the old text is being replaced with the new text.”

tips:
- For multi−line text inputs and long strings in single line, you can use YAML’s literal block scalar syntax with ‘|‘.

Important Constrains:
- MAKE SURE your modifications do not contradict each other. For example, do not propose to delete a substring and also replace that same substring elsewhere.
- DO NOT contain H1 headings (lines like ‘# Heading‘) in your modifications, which will break the document structure.
- Start subheadings from level 2 (##) or 3 (###) only.
- You can add new sections within the block using subheadings (Headings start from H3.)

Example Output:
Here is my analysis of the feedback... [model’s reasoning prose goes here]...
Based on this, I propose the following modifications:
‘‘‘yaml
- op: ”REPLACE”
  old: |
    ABC
  new: |
    ABC
    DEF
  reason: |
    your reason
    in two lines
- op: ”REMOVE”
  substr: |
    ABCDE

```
    reason: |
      XXX
  - op: "ADD"
    addition: |
      CDE
    reason: |
      your reason (can be very very long ...)
```