KnowEvo: Knowledge Evolution for Protein Optimization

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large language models (LLMs) have shown promise in scientific discovery tasks such as protein optimization, but remain limited by the lack of a mechanism for persistent and reusable knowledge accumulation. Existing approaches adapt LLMs through parameter optimization or trajectory refinement, but in both cases, knowledge remains implicit, either encoded in opaque model weights or scattered across transient solution histories. As a result, knowledge cannot be systematically inspected, attributed, or reused. We propose *KnowEvo*, a framework that treats externalized knowledge itself as the optimization target. Our key insight is that scientific knowledge is inherently modular: multiple interacting mechanisms jointly determine the design outcomes. This makes knowledge evolution fundamentally challenging, as improvements cannot be attributed to individual components in isolation. To address this, *KnowEvo* represents knowledge as executable expertise blocks and organizes their refinements into strategy mutation trees, enabling structured exploration over a combinatorial knowledge space. The framework combines a forward exploration phase that adaptively allocates evaluation budget across strategy variants with a backward revision phase in which an agentic sandbox converts rollout evidence into localized program improvements. This design enables persistent accumulation, attribution, and refinement of mechanistic expertise across iterative experimentation. We evaluate *KnowEvo* on protein optimization tasks including thermostability and solubility. Results show that knowledge evolution consistently improves over initial expert heuristics and outperforms monolithic program evolution and prior LLM-guided evolution baselines across ranking metrics such as pass@k, MRR, MAP, and NDCG@5. Codes are available at `https://anonymous.4open.science/r/KnowEvo-6E3B/`.

1 Introduction

Scientific discovery is inherently iterative. In domains such as protein engineering, progress depends not only on generating candidate solutions, but also on accumulating, revising, and reusing knowledge across repeated rounds of experimentation. Human experts rarely solve such tasks through isolated reasoning alone; instead, they progressively refine mechanistic heuristics from experimental feedback and reuse them in future design decisions.

Large language models (LLMs) possess strong reasoning capabilities and have recently shown promise in scientific domains including chemistry, biology, and materials discovery [1]. However, current LLM systems remain fundamentally limited in their ability to accumulate scientific expertise over time. Although they can generate candidate hypotheses or mutation proposals, they typically lack a persistent and structured mechanism for refining reusable knowledge across iterative experiments.

Submitted to 40th Conference on Neural Information Processing Systems (NeurIPS 2026). Do not distribute.

36 Existing approaches adapt LLM behavior primarily through parameter optimization, for example
37 through reinforcement learning [2, 3]. While effective in some settings, the acquired knowledge
38 becomes entangled within opaque model weights, making it difficult to inspect, attribute, or selectively
39 revise. Recent approaches, such as AlphaEvolve [4] and prompt-evolution frameworks [5], repeatedly
40 revise generated artifacts based on evaluation feedback. However, these methods still treat learning
41 as trajectory optimization: experience remains transient, localized to individual rollouts, and weakly
42 structured for long-term reuse. As a result, current systems can improve outputs, but they do not
43 explicitly accumulate reusable scientific expertise.

44 In this work, we argue that LLMs fall short in scientific reasoning not due to insufficient capacity,
45 but because knowledge is not treated as a learnable object. This observation motivates a shift in
46 perspective: Instead of optimizing model parameters or solution trajectories, we propose to directly
47 optimize externalized, persistent knowledge that conditions the model's behavior. Under this view,
48 learning is no longer about modifying the model itself, but about constructing and refining an explicit
49 representation of domain knowledge.

50 However, optimizing knowledge directly introduces several fundamental challenges. First, knowledge
51 does not naturally form a continuous optimization space: it is discrete, compositional, and structurally
52 heterogeneous. Second, credit assignment becomes entangled because multiple knowledge compo-
53 nents jointly influence each generated solution while feedback is often global. Third, evaluation
54 signals are not directly usable for knowledge refinement, as scientific feedback is often sparse and
55 scalar, providing limited guidance for structured updates.

56 To address these challenges, we propose *KnowEvo*, a knowledge evolution framework for protein
57 optimization. *KnowEvo* is built on three key principles: (1) Transforming unstructured context into a
58 structured and modular knowledge space; (2) Enabling credit assignment at the level of knowledge
59 components; and (3) Converting evaluation signals into structured feedback for knowledge refinement.
60 Together, these mechanisms make it possible to perform effective optimization in a discrete knowledge
61 space. The main contributions of this paper are:

62 - We reformulate scientific discovery as optimization over externalized, persistent knowledge
63 rather than over model parameters, decoupling domain adaption from expensive finetuning.
64 - We represent knowledge as executable expertise blocks organized into mutation trees,
65 and propose a knowledge evolution mechanism that combines adaptive evaluation budget
66 allocation with agentic evidence-driven revision.
67 - We demonstrate that *KnowEvo* outperforms root expert heuristics, monolithic program
68 evolution, and prior LLM-guided evolution methods.

69 # 2 Methodology

70 ## 2.1 Overall Framework

71 **Problem Formulation.** We formulate protein optimization as an iterative optimization process over
72 a knowledge base K, which is a collection of executable expertise blocks, each encoding a design
73 mechanism as a concrete program. Let $\mathcal{S}$ denote the protein-state space, where a state $s \in \mathcal{S}$ contains
74 the current protein sequence and any task-specific features used by the strategy blocks. Let $A(s)$
75 be the set of valid protein edit actions for state s, and let $\mathcal{D}$ denote the distribution of initial protein
76 states.

77 Given a protein state s_t, the current knowledge base K selects an protein edit action $a_t \sim q(\cdot \mid K, s_t)$
78 via sampling method q. The action then produces a new protein state $s_{t+1} = T(s_t, a_t)$ and a
79 reward $r_t = R(s_{t+1})$, where T denotes the transition function and R is a task-specific objective
80 evaluated through computational tools or wet-labs. A rollout of length H yields a trajectory $\rho =$
81 $(s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_H)$. The objective is to find a knowledge base K^* that maximizes the
82 expected cumulative reward J over a distribution of protein states $\mathcal{S}$:

$$K^* = \arg\max_K \ J(K) = \mathbb{E}_{s\sim\mathcal{S}} \left[\sum_{t=0}^{H-1} r_t \right]. \quad (1)$$

83 Optimizing a knowledge base introduces three distinct challenges: (1) Representation: How to trans-
84 form unstructured scientific heuristics into a structured form that supports fine-grained optimization

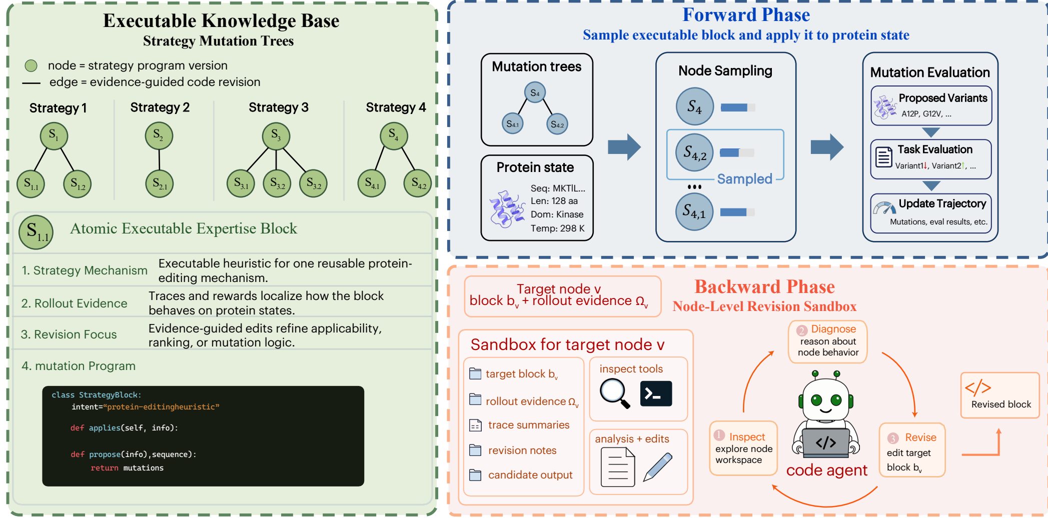

Figure 1: **Overall framework of KnowEvo.** KnowEvo evolves executable strategy mutation trees by alternating forward node evaluation with backward sandbox-based program revision.

and credit attribution? (2) Feedback localization: How to convert sparse, global evaluation signals into structured, localized feedback for individual knowledge components? (3) Context scaling: How to manage the large volume of execution trajectories without suffering from context rot, where critical information is lost or compressed beyond utility? To address these, we introduce three corresponding components: (1) *Executable expertise blocks* organized as *strategy mutation trees*, which offer a structured, externalized knowledge base for optimization; (2) *Sampling algorithm*, which focuses on budget allocation and effective computational resource assignment; (3) Code agent equipped with *agentic sandbox*, which maximizes the utilization of structured feedback from evaluation signals.

As shown in Figure 1, we organize the knowledge base as a mutation tree $\mathcal{T}_b = (\mathcal{V}_b, \mathcal{E}_b)$, where $\mathcal{V}_b$ and $\mathcal{E}_b$ denote the set of nodes and the set of edges in the tree. Each mutation tree $\mathcal{T}_b$ corresponds to a class of expert design mechanisms, and each strategy node $v \in \mathcal{V}_b$ stores a concrete block version, together with a set of trajectories $\{\rho\}_v$ from previous rollouts.

Training Process. Training proceeds through two asynchronous phases: the forward exploration phase and the backward evolution phase. The *forward phase* evaluates current strategies against sampled protein states. Given a protein state $s_t \in \mathcal{S}$ at step t, the framework samples candidate strategies from the mutation trees $\{\mathcal{T}_b\}$ and executes each as an independent block-level rollout. During each rollout, the block proposes a protein edit action a_t, which produces $s_{t+1} = T(s_t, a_t)$, and receives reward $R(s_{t+1})$ on the task objective. The resulting trajectory set $\{\rho\}_v$ is then stored as structured information within the corresponding strategy nodes v. In the *backward phase*, when a strategy node $v \in \mathcal{V}_b$ accumulates sufficient trajectories, i.e., $\#\{\rho\}_v \geq N_{\text{threshold}}$, a code agent revises its program within the sandbox; validated revisions are appended as child nodes in the same mutation tree $\mathcal{T}_b$.

Budget-constrained expertise evolution. Both node evaluation and strategy revision consume scarce resources. Let B_{eval} be the total number of reward queries and B_{work} the total number of agent actions available for code revision. The practical optimization problem reduces to a coupled budget allocation: given mutation trees $\{\mathcal{T}_b\}$ and budgets $B_{\text{eval}}, B_{\text{work}}$, allocate evaluation effort across nodes to estimate their utilities, and allocate revision effort to improve the most promising candidates, such that the expected deployment performance $J(K_{\text{final}})$ is maximized. The forward phase addresses evaluation allocation via Thompson-style posterior sampling; the backward phase addresses revision allocation through scalable agentic code refinement.

2.2 Executable Expertise Blocks

A central challenge in applying large language models to scientific discovery lies in transforming unstructured textual knowledge into a representation that allows better systematic optimization and

118 resource assignment. In this work, we adopt executable strategy blocks instead of natural-language
119 prompts as the form of externalized knowledge. Although natural language can describe scientific
120 heuristics, it lacks explicit operational semantics: applying a heuristic to a concrete protein state
121 requires the LLM to re-interpret the prompt at every invocation, making execution non-deterministic
122 and evaluation costly. We instead concretize each piece of knowledge as an executable program
123 with explicit inputs, parameters, and mutation outputs, making it deterministic and revisable at a
124 mechanism-level granularity.

125 Each executable expertise block can be viewed as a mechanism-level scientific hypothesis: it proposes
126 that a particular design mechanism is useful under a particular set of conditions. Rather than encoding
127 the entire protein-design policy in one monolithic program, a block isolates one strategy component
128 with explicit applicability logic and mutation behavior. This decomposition aligns the unit of
129 execution with the unit of evidence. When a block succeeds or fails, the resulting trajectory provides
130 a localized signal about that mechanism, rather than an entangled signal about a full policy. The
131 system can therefore revise the block's applicability boundary, action logic, or internal parameters
132 without overwriting unrelated knowledge.

133 We therefore organize the executable knowledge base as a collection of strategy mutation trees. For
134 each root expertise block $b \in \mathcal{B}$, the framework maintains a tree $\mathcal{T}_b = (\mathcal{V}_b, \mathcal{E}_b)$, where each node
135 $v \in \mathcal{V}_b$ is an executable version of the same mechanism-level strategy, and each edge $(u, v) \in \mathcal{E}_b$
136 records an accepted evidence-driven revision from parent u to child v. Thus, the knowledge base
137 $K = \{\mathcal{T}_b\}_{b\in\mathcal{B}}$ preserves the revision lineage of each mechanism while exposing all validated versions
138 as candidate strategies for future execution.

139 2.3 Evaluation Budget Allocation in Forward Phase

140 In the forward phase, the evaluation budget is allocated across candidate nodes in the mutation
141 trees. Each node $v \in \mathcal{V}_b$ is a complete, independently executable expertise block. Its utility U_v is
142 unknown a priori and needs to be inferred from execution outcomes. As the mutation tree grows,
143 the system continuously decides which nodes to evaluate, balancing exploitation of reliable nodes
144 against exploration of uncertain or recently revised ones. We therefore cast forward budget allocation
145 as an *online posterior allocation* problem over a dynamically expanding candidate node set $\mathcal{V}_b$. This
146 formulation differs from tree search methods such as Monte Carlo Tree Search (MCTS) [6], where
147 nodes represent intermediate states and the objective is path selection rather than allocation over
148 independent alternatives.

149 **Evaluation budget as online posterior allocation.** We model the outcome of executing a strategy
150 node v on a given input as a binary success/failure with unknown success probability p_v. The utility
151 U_v is identified with p_v.

152 Let $\mathcal{H}_v(t) = (S_v(t), F_v(t))$ be the cumulative numbers of successes and failures observed for node v
153 before the t-th forward rollout. The posterior distribution of p_v takes the following form:

$$p(p_v \mid \mathcal{H}_v(t)) = \mathrm{Beta}\big(S_v(t) + 1,\ F_v(t) + 1\big).$$

154 Specially, nodes without any observation inherit an uninformative prior $\mathrm{Beta}(1, 1)$, which is uniform
155 on $[0, 1]$. For each incoming task, the framework forms an *eligible node set* $\mathcal{A}_b(t) \subseteq \mathcal{V}_b$ for the
156 mutation tree and draws a posterior sample of the success probability for every candidate:

$$\tilde{p}_v(t) \sim \mathrm{Beta}\big(S_v(t) + 1,\ F_v(t) + 1\big), \qquad v \in \mathcal{A}_b(t).$$

157 These samples are converted into a probability distribution over nodes via a softmax with a fixed
158 temperature $\tau > 0$:

$$\pi_{b,t}(v) = \frac{\exp(\tilde{p}_v(t)/\tau)}{\sum_{u\in\mathcal{A}_b(t)} \exp(\tilde{p}_u(t)/\tau)}, \qquad v \in \mathcal{A}_b(t).$$

159 When $\tau \to 0$, the softmax reduces to the conventional Thompson sampling, concentrating on the node
160 with the largest sampled $\tilde{p}_v$. In this limit, the selection probability matches the posterior probability
161 that a node is optimal:

$$\Pr(v_t = v \mid \mathcal{H}_t) = \Pr\Big(v = \arg\max_{u\in\mathcal{A}_b(t)} p_u \mid \mathcal{H}_t\Big),$$

162 which, for a static set of arms, enjoys logarithmic regret and asymptotic optimality. In our setting,
163 however, the mutation tree continuously receives revised strategies as new nodes that may eventually
164 outperform the current best. Pure exploitation is therefore undesirable. Instead of the hard argmax
165 in standard Thompson sampling, we adopt a softmax sampling, giving newly created revisions a
166 non-negligible probability of being selected. This prevents the system from committing prematurely
167 to a potentially suboptimal strategy, while still allowing well-performing nodes to dominate the
168 allocation as their advantage grows.

169 **Discovery of new nodes.** A critical requirement is that a newly created revision v_{new} receives
170 evaluation budget quickly. Since $\tilde{p}_{\text{new}} \sim \text{Beta}(1,1)$ and the softmax function is strictly positive for
171 any finite $\tau > 0$, there exists a constant $\delta(\tau, |\mathcal{A}_b|) > 0$ such that

$$\Pr(v_{\text{new}} \text{ selected}) \;\geq\; \delta(\tau, |\mathcal{A}_b|),$$

172 where δ increases with τ and decreases with $|\mathcal{A}_b|$. Consequently, within any block of b consecutive
173 rollouts, the new node receives at least one evaluation with probability at least $1 - \big(1-\delta\big)^b$. Hence,
174 the temperature τ explicitly controls how aggressively the system surfaces promising revisions. A
175 larger τ accelerates the discovery of new strategies at the cost of diluting the budget spent on already
176 reliable nodes.

177 **Allocation dynamics.** For a total evaluation budget B_{eval}, the expected number of rollouts assigned
178 to node v is

$$\mathbb{E}\big[N_v(B_{\text{eval}})\big] = \sum_{t=1}^{B_{\text{eval}}} \mathbb{E}\big[\pi_{b,t}(v)\big].$$

179 Under softmax sampling, the selection probabilities never fully concentrate on a single node even as
180 evidence accumulates, because the posterior samples of suboptimal arms retain a non-zero chance of
181 exceeding those of the current best. In practice, the distribution $\pi_{b,t}$ shifts smoothly: reliable nodes
182 with high success counts receive a large share of the budget, while uncertain or recently revised nodes
183 continue to be probed. This yields a built-in exploration–exploitation trade-off that adapts online to
184 the evolving expertise encoded in the mutation tree. The evaluation budget is thus never committed
185 to a fixed strategy for an entire batch; it is continuously reallocated at the granularity of individual
186 samples, ensuring that the system remains responsive to improvements generated in the backward
187 phase.

188 ## 2.4 Agentic Revision Sandbox

189 Scientific optimization requires turning evaluation feedback into an improved executable strategy. In
190 our setting, each strategy node is a program. A direct revision operator could serialize the accumulated
191 feedback into an LLM prompt and ask for a revised program, but this reduces revision to a single-step
192 text generation process. Given the substantial and heterogeneous feedback in our framework, such a
193 prompt-only operation is insufficient and may suffer from context rot.

194 We therefore perform backward evolution inside a *revision sandbox*. The sandbox is an isolated
195 executable environment that materializes the target block, rollout evidence, runtime traces, task
196 schemas, analysis scripts, and validation tests as persistent external state. It is distinct from the task
197 environment $\mathcal{E}$, which evaluates candidate mutations. The code agent operates inside the revision
198 sandbox: it can inspect evidence on demand, run analyses, edit the target block, execute validation
199 scripts, and repair runtime failures before submitting a successor strategy.

200 For a strategy node v with executable block b_v, the forward phase spends an evaluation budget B_{eval}
201 in the task environment $\mathcal{E}$, producing an observation

$$\Omega_v^{(B_{\text{eval}})} \sim \text{Observe}_{B_{\text{eval}}}(b_v; \mathcal{E}).$$

202 The observation $\Omega_v^{(B_{\text{eval}})}$ is then materialized into the revision sandbox rather than compressed into a
203 flat prompt. The agent receives bounded views of this external state and acts through tool calls to
204 analyze evidence, modify the program, and validate the revised block.

205 We model the backward phase as an agentic revision operator with internal work budget B_{work}:

$$b_{v'} \sim \mathcal{A}_{\phi, B_{\text{work}}}(b_v, \Omega_v^{(B_{\text{eval}})}, J).$$

206 Here J denotes the task objective, and B_{work} controls how many agent actions can be spent on
207 inspection, analysis, editing, execution, and repair. A proposed successor is accepted only if it
208 satisfies the runtime contract and passes validation. An accepted proposal is inserted into the
209 same strategy mutation tree as a child node, where it can receive future evaluation budget through
210 subsequent rollouts.

211 This abstraction separates evidence scaling from agentic-work scaling. The forward budget B_{eval}
212 controls how much experimental evidence is generated, the retained sandbox state controls how
213 much evidence remains available outside the agent's immediate context window, and κ controls how
214 much internal work can be spent converting that evidence into a validated successor program. Thus,
215 large scientific feedback need not be compressed into a single prompt; it becomes an executable,
216 inspectable, and revisable external state.

217 3 Experiments

218 In the experiments, we try to answer the following questions: **RQ1: Does knowledge evolution**
219 **improve task performance?** We compare revised strategy trees against their root seed blocks and
220 mutation-generation baselines under matched reward-call budgets. **RQ2: How do the core design**
221 **choices affect performance?** We ablate the forward allocation policy, the revision sandbox, and
222 the mutation-tree structure. **RQ3: How does strategy evolution scale with evidence and agentic**
223 **work?** We vary the number of trajectories available for revision and the code agent's internal work
224 budget to separate information scaling from agentic-work scaling.

225 3.1 Tasks, Protocol, and Metrics

226 We evaluate on two BRENDA-derived protein optimization tasks: stability, scored by Rosetta
227 REF2015 energy, and solubility, scored by a SaProt-based solubility predictor. For each task, we use
228 5000 training proteins and 200 held-out test proteins after filtering invalid sequences. All methods
229 are evaluated on the same held-out split .

230 A candidate is counted as successful if it improves the active backend score relative to the parent
231 sequence. For evaluation, every mutation generator is exposed through a common ranked candidate-
232 list interface. Given a parent protein, a method returns $\mathcal{C} = (M_1, \ldots, M_m)$, $m \leq 5$, where each M_i
233 is a mutation set applied independently to the parent sequence, and the index i defines its proposed
234 rank. This includes the monolithic-program baseline, whose program is required to emit a ranked list
235 rather than a single mutation. We report pass@k, which measures whether any successful candidate
236 appears within the top-k list, together with MRR, MAP, and NDCG@5 to measure whether successful
237 candidates are ranked early. Unless otherwise stated, all backward revisions in the main experiments
238 and ablation studies use MiniMax-2.7[7] API to power code agent. Full dataset construction, backend
239 configurations, validity statistics, and metrics are given in Appendix B.

240 3.2 Baselines

241 We compare against methods that optimize different objects: root seed blocks, which evaluate the
242 initial executable heuristics without revision; a monolithic program baseline, which flattens the seed
243 heuristics into a single mutation generator but uses the same ranked candidate-list interface; and
244 Adaevolve [8], an LLM-guided program-evolution baseline with the same reward-call budget. When
245 available, we also include task-specific policy optimization baselines. Detailed baseline instantiations
246 are provided in Appendix B.

247 3.3 Main Results

248 Table 1 shows that knowledge evolution consistently improves held-out performance across both
249 tasks. On Rosetta stability, our method improves over root seed blocks from 42.5% to 50.5% in
250 pass@1 and from 74.2% to 82.8% in pass@3. On SaProt solubility, the gains are larger: pass@1
251 increases from 31.6% to 51.5%, and pass@3 increases from 57.7% to 83.2%. The same trend holds
252 for MRR, MAP, and NDCG@5, indicating that successful mutations are not only found more often
253 254 but also ranked earlier. Qualitative case studies in Appendix C illustrate how rollout evidence is
254 converted into executable residue-level rules.

Table 1: Comparisons across stability and solubility tasks. Bold indicates the best result for each task.

Method	Hit rate			Ranking quality		
	pass@1 ↑	pass@3 ↑	pass@5 ↑	MRR ↑	MAP ↑	NDCG@5 ↑
Stability / Rosetta						
Direct LLM proposal	37.7	70.4	82.4	54.7	54.7	61.7
pro-1 8B	42.6	69.0	77.5	56.4	53.2	61.0
Monolithic program	43.4	72.2	83.8	58.7	55.6	64.2
AdaEvolve	46.0	72.2	81.8	60.0	58.0	65.2
Root seed blocks	42.5	74.2	85.3	59.1	56.3	65.2
KnowEvo (Ours)	**50.5**	**82.8**	**88.4**	**66.4**	**63.2**	**71.2**
Solubility / SaProt						
Direct LLM proposal	50.5	61.7	63.8	56.3	56.3	58.2
Monolithic program	35.2	65.8	79.6	51.9	49.2	58.3
AdaEvolve	49.0	74.0	82.5	61.9	59.8	66.9
Root seed blocks	31.6	57.7	85.2	47.2	47.5	57.5
KnowEvo (Ours)	**51.5**	**83.2**	**90.8**	**67.8**	**64.2**	**72.8**

Table 2: Ablation study on the Solubility/SaProt tasks. All variants use the same reward-call budget and held-out candidate-list evaluation protocol.

Method	Hit rate			Ranking quality		
	pass@1 ↑	pass@3 ↑	pass@5 ↑	MRR ↑	MAP ↑	NDCG@5 ↑
Random node allocation	51.0	78.5	88.2	60.6	60.0	67.3
Single-step LLM revision	49.0	80.1	84.2	64.2	62.6	69.3
Chain-only revision	43.4	54.1	54.1	58.7	48.7	50.1
KnowEvo (Full method)	**51.5**	**83.2**	**90.8**	**67.8**	**64.2**	**72.8**

255 Compared with the monolithic program and Adaevolve, our method achieves the best early-rank
256 metrics on both tasks. This supports the main claim that optimizing a structured tree of executable
257 expertise blocks is a more effective target than evolving a single flattened mutation generator.

258 3.4 Ablation Study

259 We conduct a diagnostic ablation study on the solubility/SaProt task to isolate the contribution of
260 each core design choice in the knowledge-evolution loop. We use solubility for this analysis because
261 its reward backend is substantially cheaper than Rosetta and supports repeated controlled runs under
262 matched settings. All variants use the same seed blocks, task adapter, held-out split, reward-call
263 budget, and ranked candidate-list evaluation protocol.

264 We ablate three components. First, we replace the Thompson-style posterior allocation with uniform
265 random node sampling, testing whether adaptive evaluation over strategy nodes is useful. Second,
266 we remove the revision sandbox and ask a single LLM call to directly revise the target block from
267 serialized evidence, testing whether persistent external state and tool-based repair are needed. Third,
268 we replace the mutation tree with chain-only growth, where each strategy family keeps only the
269 latest accepted successor, testing whether retaining competing ancestors and siblings improves final
270 strategy selection.

271 Table 2 shows that each component contributes to the final performance. Random node allocation still
272 improves over the root seed blocks, because revision remains active and the final deployed nodes are
273 selected by the same LCB-based rule as in the full method. Thus, random allocation can still produce
274 useful nodes. Its weakness is not the absence of revision, but inefficient budget use: promising nodes
275 are not preferentially evaluated and therefore are not fully exploited. This explains why it remains
276 competitive in pass@1 but falls behind the full method on pass@3, MRR, MAP, and NDCG@5.

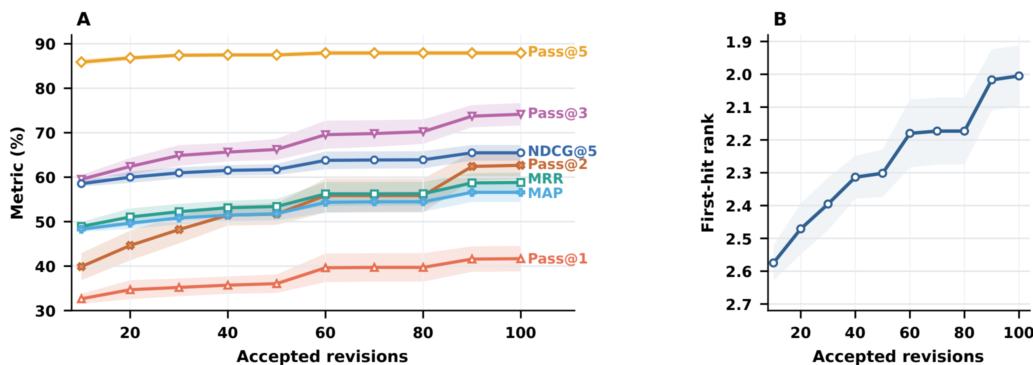

Figure 2: Evidence scaling through accumulated revisions on solubility/SaProt. Best-so-far checkpoint performance is averaged over six revision trajectories. (A) Ranking metrics across accepted revisions. MRR, MAP, and NDCG@5 are plotted as $100\times$metric. (B) Average first-hit rank: lower is better.

277 The single-step LLM revision variant also improves over weaker baselines but is consistently worse
278 than the full method, suggesting that serializing evidence into one prompt is less reliable than
279 preserving rollout traces, diagnostics, analysis scripts, and validation feedback as external sandbox
280 state. The chain-only variant performs worst, showing that linear revision is insufficient: later
281 accepted successors do not always dominate their ancestors or siblings, and discarding competing
282 versions removes useful strategy alternatives. Overall, these ablations support the coupled design
283 of our framework: revision creates useful successors, posterior allocation makes evaluation budget
284 responsive to evidence, and the mutation tree preserves alternatives for robust final LCB-based
285 selection.

286 3.5 Scaling External Knowledge Through Agentic Revision

287 RQ3 asks whether the revision sandbox can convert additional resources into better executable
288 strategy knowledge. We consider three resource axes. First, as more forward rollouts are accumulated,
289 the sandbox receives more trajectory evidence for revision; in our implementation this is reflected
290 by more accepted revisions in the strategy tree. Second, when the evidence input is fixed, the code
291 agent may spend more internal work converting that evidence into program changes. Third, under the
292 same evidence input and revision protocol, a stronger code agent may perform this evidence-to-code
293 conversion more effectively. We use solubility/SaProt as a dense diagnostic setting because it supports
294 frequent checkpoint evaluation.

295 **Evidence scaling through accumulated revisions.** We first study whether performance improves
296 as more rollout evidence is supplied to the revision process. Since each accepted revision is produced
297 from a batch of forward evaluation evidence, the number of accepted revisions serves as a checkpoint
298 proxy for cumulative information input. We run six independent revision trajectories and evaluate the
299 strategy tree every ten accepted revisions on the same frozen diagnostic split. For each trajectory, we
300 report the best-so-far checkpoint performance among all validated revisions available up to that point,
301 and then average across trajectories.

302 Figure 2 shows that additional revision evidence improves the ranked candidate shortlist. From
303 10 to 100 accepted revisions, pass@3 rises from 59.5% to 74.2%, while the average first-hit rank
304 decreases from 2.58 to 2.01. pass@5 is already high and changes only modestly, but early-rank and
305 ranking-sensitive metrics such as pass@1, pass@2, MRR, MAP, and NDCG@5 improve consistently.
306 Thus, more input evidence does not merely populate the mutation tree with more variants; it helps
307 discover strategy versions that move successful mutations earlier in the validation shortlist.

308 **Scaling the evidence-to-code conversion.** We next fix the accumulated evidence input and ask
309 whether the conversion from stored evidence to executable strategy code improves with more agentic
310 work or a stronger code agent. For the work-budget axis, we compare the common rev100 checkpoint
311 across revision-loop budgets. The revision-loop budget κ controls how much inspection, editing,

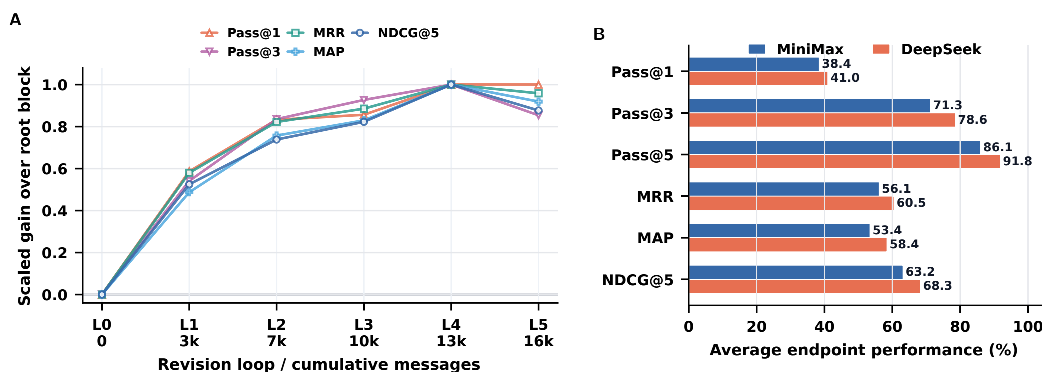

Figure 3: Scaling the evidence-to-code conversion on solubility/SaProt. (A) Agent-work scaling at fixed evidence input: realized code-agent work is measured by cumulative code-agent turns, and the y-axis reports scaled endpoint gains over the initial root-block selection. (B) Agent-capability diagnostic: average endpoint performance under the same revisions.

execution, and revision that the agent may spend inside the sandbox before committing each validated child node. For each run, we record the cumulative number of code-agent turns used to reach the checkpoint and use it as the realized agentic-effort proxy. The root-block selection is included as a zero-effort reference point.

For the capability axis, we compare MiniMax and DeepSeek under the same revision-sandbox protocol and matched evidence input. This comparison tests whether the sandbox benefits when the evidence-to-code operator becomes stronger, rather than testing a new reward model or evaluation protocol.

Figure 3 shows two complementary forms of scaling after evidence has been accumulated. In panel A, increasing sandbox work improves strategy quality under fixed evidence input. Because the comparison uses the same revision checkpoint, the gains cannot be explained by simply adding more nodes to the tree; instead, additional agentic work helps convert the same stored evidence into higher-quality accepted revisions.In panel B, replacing MiniMax with DeepSeek improves average diagnostic endpoint performance across all reported ranking metrics, Because stronger revision agents require higher agent-call cost, we use DeepSeekV4-flash[9] as a controlled capability diagnostic rather than as the default benchmark agent. The goal is to test whether the same revision-sandbox interface improves when the evidence-to-code operator becomes stronger.

Together, these results support the intended scaling behavior of the revision sandbox: more rollout evidence improves the best discovered strategy, more agentic work improves how fixed evidence is converted into revisions, and stronger code agents can further improve this conversion under the same external interface.

4 Conclusion

We presented *KnowEvo*, a framework that recasts LLM-based protein optimization as the iterative refinement of externalized, persistent knowledge rather than model parameters or transient solution trajectories. By combining executable expertise blocks, strategy mutation trees, and a Thompson-sampling forward allocator with an agentic revision sandbox in the backward phase, *KnowEvo* converts sparse scalar rewards into localized, inspectable program revisions. Experiments on Rosetta stability and SaProt solubility demonstrate that *KnowEvo* consistently improves over root seed blocks, monolithic program evolution, and LLM-guided evolution baselines across pass@k, MRR, MAP, and NDCG@5, and our scaling study shows that the gains compound with more rollout evidence and stronger code agents. These results indicate that making knowledge itself the optimization target is a practical path toward LLM systems that accumulate, attribute, and reuse scientific expertise over time.

345 # References

346 [1] Zhangde Song, Jieyu Lu, Yuanqi Du, Botao Yu, Thomas M. Pruyn, Yue Huang, Kehan Guo, Xiuzhe Luo,
347 Yuanhao Qu, Yi Qu, Yinkai Wang, Haorui Wang, Jeff Guo, Jingru Gan, Parshin Shojaee, Di Luo, Andres M
348 Bran, Gen Li, Qiyuan Zhao, Shao-Xiong Lennon Luo, Yuxuan Zhang, Xiang Zou, Wanru Zhao, Yifan F.
349 Zhang, Wucheng Zhang, Shunan Zheng, Saiyang Zhang, Sartaaj Takrim Khan, Mahyar Rajabi-Kochi,
350 Samantha Paradi-Maropakis, Tony Baltoiu, Fengyu Xie, Tianyang Chen, Kexin Huang, Weiliang Luo,
351 Meijing Fang, Xin Yang, Lixue Cheng, Jiajun He, Soha Hassoun, Xiangliang Zhang, Wei Wang, Chandan K.
352 Reddy, Chao Zhang, Zhiling Zheng, Mengdi Wang, Le Cong, Carla P. Gomes, Chang-Yu Hsieh, Aditya
353 Nandy, Philippe Schwaller, Heather J. Kulik, Haojun Jia, Huan Sun, Seyed Mohamad Moosavi, and Chenru
354 Duan. Evaluating large language models in scientific discovery. *https://arxiv.org/abs/2512.15567*, 2025.

355 [2] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon,
356 Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder,
357 Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement
358 with self-feedback. *https://arxiv.org/abs/2303.17651*, 2023.

359 [3] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao.
360 Reflexion: Language agents with verbal reinforcement learning. *https://arxiv.org/abs/2303.11366*, 2023.

361 [4] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner,
362 Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar,
363 Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and
364 Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint*
365 *arXiv:2506.13131*, 2025.

366 [5] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav
367 Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. GEPA: Reflective prompt evolution can
368 outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

369 [6] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and*
370 *Games*, 2006.

371 [7] MiniMax. Minimax m2.7: Early echoes of self-evolution. `https://www.minimax.io/news/`
372 `minimax-m27-en`, March 2026. Accessed: 2026-05-06.

373 [8] Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren,
374 Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, et al. Adaevolve: Adaptive llm driven zeroth-order
375 optimization. *arXiv preprint arXiv:2602.20133*, 2026.

376 [9] DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

377 [10] A. Rives, J. Meier, T. Sercu, S. Goyal, Z. Lin, J. Liu, D. Guo, M. Ott, C.L. Zitnick, J. Ma, and R. Fergus.
378 Biological structure and function emerge from scaling unsupervised learning to 250 million protein
379 sequences. *Proc. Natl. Acad. Sci. U.S.A.*, 2021.

380 [11] Zeming Lin, Halil Akin, Roshan Rao, Brian Hie, Zhongkai Zhu, Wenting Lu, Nikita Smetanin, Robert
381 Verkuil, Ori Kabeli, Yaniv Shmueli, Allan dos Santos Costa, Maryam Fazel-Zarandi, Tom Sercu, Salvatore
382 Candido, and Alexander Rives. Evolutionary-scale prediction of atomic-level protein structure with a
383 language model. *Science*, (6637):1123–1130, 2023.

384 [12] Ali Madani, Ben Krause, Eric R. Greene, Subu Subramanian, Benjamin P. Mohr, James M. Holton, Jose
385 Luis Olmos Jr., Caiming Xiong, Zachary Z. Sun, Richard Socher, James S. Fraser, and Nikhil Naik. Large
386 language models generate functional protein sequences across diverse families. *Nature Biotechnology*,
387 page 1099–1106, 2023.

388 [13] Haoran Sun, Liang He, Pan Deng, Guoqing Liu, Zhiyu Zhao, Yuliang Jiang, Chuan Cao, Fusong Ju, Lijun
389 Wu, Haiguang Liu, Tao Qin, and Tie-Yan Liu. Accelerating protein engineering with fitness landscape
390 modelling and reinforcement learning. *Nature Machine Intelligence*, page 1446–1460, 2025.

391 [14] Yi Wang, Hui Tang, Lichao Huang, Lulu Pan, Lixiang Yang, Huanming Yang, Feng Mu, and Meng Yang.
392 Self-play reinforcement learning guides protein engineering. *Nature Machine Intelligence*, page 845–860,
393 2023.

394 [15] Nathaniel Blalock, Srinath Seshadri, and Philip Romero. Functional alignment of protein language models
395 via reinforcement learning with experimental feedback. In *NeurIPS*, 2024.

396 [16] Ziwen Wang, Jiajun Fan, Ruihan Guo, Thao Nguyen, Heng Ji, and Ge Liu. Proteinzero: Self-improving
397 protein generation via online reinforcement learning. *arXiv preprint arXiv:2506.07459*, 2025.

398 [17] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James
399 Zou. Textgrad: Automatic differentiation via text. *arXiv preprint arXiv:2406.07496*, 2024.

400 [18] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large
401 language models as optimizers. In *The Twelfth International Conference on Learning Representations*
402 *(ICLR)*, 2023.

403 [19] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel.
404 Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*,
405 2023.

406 [20] Andres M. Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D. White, and Philippe Schwaller.
407 Augmenting large language models with chemistry tools. *Nature Machine Intelligence*, 6:525–535, 2024.

408 [21] Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with
409 large language models. *Nature*, 624:570–578, 2023.

410 [22] Sandra Placzek, Ida Schomburg, Antje Chang, Lisa Jeske, Marcus Ulbrich, Jana Tillack, and Dietmar
411 Schomburg. Brenda in 2017: new perspectives and new tools in brenda, 2016.

412 [23] Rebecca F Alford, Andrew Leaver-Fay, Jeliazko R Jeliazkov, Matthew J O'Meara, Frank P DiMaio,
413 Hahnbeom Park, Maxim V Shapovalov, P Douglas Renfrew, Vikram K Mulligan, Kalli Kappel, et al. The
414 rosetta all-atom energy function for macromolecular modeling and design, 2017.

415 [24] Jin Su, Chenchen Han, Yuyang Zhou, Junjie Shan, Xibin Zhou, and Fajie Yuan. Saprot: Protein language
416 modeling with structure-aware vocabulary. *BioRxiv*, pages 2023–10, 2023.

417 [25] Michael Hla. Pro-1 - Michael Hla, 2025.

418 # A Related Work

419 ## A.1 Large Language Models in Protein Engineering

420 Protein language models (pLM) capture the hierarchical structure of biological sequences through
421 unsupervised pre-training [10]. Models like ESM-2 [11] and ProGen [12] have demonstrated
422 that by predicting masked amino acids across billions of sequences, pLMs learn to approximate
423 the underlying biophysical constraints of protein folding. However, these models are primarily
424 descriptive or generative in a general sense, and they lack an inherent mechanism to optimize for
425 specific downstream functions, such as increased catalytic activity or specific binding affinity, without
426 further task-specific adaptation.

427 ## A.2 Reinforcement Learning for LLM Adaptation

428 Recent protein engineering has increasingly leveraged reinforcement learning to navigate the vast
429 sequence-function landscape. Pioneering frameworks such as μProtein [13] and EvoPlay [14]
430 demonstrate the efficacy of coupling protein language models with search algorithms like Monte Carlo
431 Tree Search (MCTS) to identify synergistic multi-point mutations that surpass natural activity levels.
432 Furthermore, recent alignment strategies, including Reinforcement Learning from Experimental
433 Feedback (RLXF) [15] and ProteinZero [16], have successfully steered generative models toward non-
434 differentiable objectives like thermal stability and structural fidelity, achieving significant reductions
435 in design failure rates compared to traditional inverse folding baselines.

436 ## A.3 LLM-Guided Evolution and Iterative Refinement

437 Prompt optimization methods improve LLM performance by iteratively refining input prompts using
438 gradient-based or search-based strategies [17, 18, 19]. These approaches typically treat prompts
439 as monolithic objects and rely on end-to-end evaluation signals. In contrast, our work focuses
440 on learning structured knowledge representations, enabling fine-grained attribution and persistent
441 accumulation, which goes beyond prompt-level optimization.

442 ### A.4 Multi-Agent Systems for Scientific Research

443 Multi-Agent Systems (MAS) have enabled the decomposition of complex scientific tasks into
444 specialized roles. In chemistry and materials science, agents like ChemCrow [20] and Coscientist [21]
445 have successfully integrated LLMs with external tools to plan and execute experiments.

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

449 ## B Experimental Setup

450 **Tasks and datasets.** We evaluate our framework on two protein optimization tasks: stability and
451 solubility. Both tasks use enzyme sequences from the BRENDA dataset[22]. We use a fixed split
452 with seed 42, consisting of 5000 training proteins and 200 held-out proteins. Because the two
453 reward backends impose different input-validity constraints, final aggregation is performed on the
454 backend-valid held-out subset for each task. This yields 198 held-out proteins for stability/Rosetta
455 and 196 held-out proteins for solubility/SaProt. The backend-valid subset is fixed before method
456 comparison, so all methods within a task are evaluated on identical proteins.

457 **Stability/Rosetta.** The stability task uses the Rosetta REF2015[23] energy function as the compu-
458 tational reward. The optimization objective is to decrease the total Rosetta energy of the mutated
459 sequence relative to the parent sequence. A candidate is counted as successful when the task adapter
460 reports a positive normalized score gain, corresponding to lower Rosetta energy under the active
461 backend. The selected strategy nodes are evaluated on the held-out split with the same Rosetta
462 backend.

463 **Solubility/SaProt.** The solubility task uses a SaProt-based [24] sequence classifier as the compu-
464 tational reward. The optimization objective is to increase the predicted soluble probability of the
465 mutated sequence relative to the parent sequence. A candidate is counted as successful when the task
466 adapter reports a positive normalized score gain .

467 **Comparison protocol.** All methods are evaluated with the same protein inputs, task adapter,
468 reward backend, held-out split, and candidate-list interface. Each method returns a ranked list of at
469 most five mutation candidates. During held-out evaluation, each candidate mutation set is applied
470 independently to the same parent sequence and scored independently. Search-based methods are
471 matched by reward-call budget. One rollout denotes one candidate evaluation under the active reward
472 backend, and the main search-based methods use a 5000-rollout budget before held-out evaluation.

473 **Evaluation metrics.** For each held-out protein, let $y_i \in \{0, 1\}$ indicate whether the candidate at
474 rank i improves the active task score. We report pass@k,

$$\text{pass@}k = \mathbb{I}\left[\sum_{i=1}^{k} y_i > 0\right],$$

475 for $k \in \{1, 3, 5\}$. pass@k measures whether at least one useful variant appears within the top-k
476 experimental shortlist.

477 To evaluate ranking quality beyond binary top-k coverage, we also report MRR, MAP, and NDCG@5.
478 Let

$$r^\star = \min\{i : y_i = 1\}$$

479 be the rank of the first successful candidate when at least one success exists. The reciprocal-rank
480 score for one protein is

$$\mathrm{RR} = \begin{cases} 1/r^\star, & \sum_{i=1}^{5} y_i > 0, \\ 0, & \sum_{i=1}^{5} y_i = 0. \end{cases}$$

481 MRR is the mean of RR over held-out proteins. It measures how early the first successful candidate
482 appears, which is important when only one or a few variants can be validated.

Average precision for one protein is

$$\mathrm{AP} = \begin{cases} \frac{1}{S} \sum_{i=1}^{5} \left(\frac{1}{i} \sum_{j=1}^{i} y_j \right) y_i, & S > 0, \\ 0, & S = 0, \end{cases} \qquad S = \sum_{i=1}^{5} y_i.$$

MAP is the mean of AP over held-out proteins. It rewards methods that place all successful candidates early in the ranked list, rather than only placing the first success early.

Finally, NDCG@5 is computed as

$$\mathrm{NDCG@5} = \begin{cases} \frac{\sum_{i=1}^{5} y_i / \log_2(i+1)}{\sum_{i=1}^{\min(S,5)} 1/\log_2(i+1)}, & S > 0, \\ 0, & S = 0. \end{cases}$$

NDCG@5 measures discounted top-5 utility: successes near the top contribute more than successes near the bottom, while normalization makes scores comparable across proteins with different numbers of successful candidates. Missing candidate positions are treated as unsuccessful. All metrics are averaged over the backend-valid held-out proteins for the corresponding task and reported as percentages.

baselines We compare the full method against baselines that optimize different objects. This distinction is important because the central claim of our approach is about the optimization target: externalized strategy knowledge.

Root seed blocks. The root-seed baseline freezes the initial task-pack blocks and evaluates them without any revision. At test time, the root node of each strategy family is run under the same mutation constraints and reward adapter as the full method; the resulting block-level candidates are ranked by the same candidate-list protocol used for the main evaluation. Thus this baseline measures the quality of the initial executable prior before any rollout-driven knowledge evolution.

Direct LLM proposal. We evaluate a direct LLM mutation proposer that receives the same knowledge inputs and proposes candidate mutations in one step. This baseline tests whether persistent strategy blocks are more useful than one-off natural-language reasoning over the current protein state. The detailed prompt is provided in D.3

Monolithic seed program. The monolithic program baseline uses the same seed mechanisms as the root blocks, but flattens them into a single Python generator with one `propose_candidates(protein_state, constraints, max_candidates)` entry point. The program therefore contains the same initial task heuristics as the root blocks: the difference is representational, not informational. In our framework, these mechanisms remain separated into strategy families with local evidence and independent revision lineages. In the monolithic baseline, they are flattened into one global policy whose candidate list is evaluated through the same top-5 protocol.

LLM-guided program evolution. Adaevolve evolves a single monolithic mutation-generator program under the same task contract, reward adapter, held-out split, and ranked-list output schema. The seed program is the flattened generator above, so the comparison does not deny the baseline access to the initial task heuristics; instead, it tests whether evolving one global generator is sufficient compared with externalized block population and block-local revision memory.

Reinforcement-learning baseline. For stability, we additionally compare with pro-1 8B, a task-specific reinforcement-learning protein optimization baseline [25]. We select the best available checkpoint and evaluate its proposed variants under the same Rosetta held-out scoring protocol. This baseline optimizes a policy directly rather than optimizing externalized executable strategy programs.

Rollout-budget matching. For our framework, a rollout is a single candidate sequence produced by one selected strategy node and scored by the active reward backend. For AdaEvolve, a rollout is one candidate emitted by the current monolithic program and scored by the same evaluator. Thus the 5000-rollout comparison corresponds to 5000 candidate reward calls in both systems. In the Adaevolve runs this is implemented as 20 search iterations, 50 training proteins per iteration, and five candidates per protein. In our runs, the trainer spends the same candidate-evaluation budget through the forward sampling loop and stores each scored trajectory on the responsible strategy node. Invalid candidates that fail before backend scoring are logged separately; backend scoring failures occupy their candidate slot and are counted as unsuccessful in the ranked-list metrics.

Paired bootstrap intervals. For the root-block comparison in the main table, we also compute paired nonparametric bootstrap intervals over held-out proteins. Each bootstrap resample draws backend-valid proteins with replacement, recomputes the mean metric difference between the full method and the root seed blocks, and reports the 2.5th and 97.5th percentiles over 20,000 resamples with seed 42. Pairing is by held-out protein identifier, so each resample compares the two methods on the same proteins. The point estimates below are the Ours–Root differences from the main table; the confidence intervals are computed from the matched per-protein evaluation records using the same candidate-list ordering.

Task	Metric	Δ Ours–Root (pp)	95% bootstrap CI (pp)
Stability/Rosetta ($n = 198$)	pass@1	+8.0	$[-0.1, 16.1]$
	pass@3	+8.6	$[2.0, 15.2]$
	pass@5	+3.1	$[-1.5, 7.7]$
	MRR	+7.3	$[1.5, 13.1]$
	MAP	+6.9	$[1.8, 12.0]$
	NDCG@5	+6.0	$[1.4, 10.7]$
Solubility/SaProt ($n = 196$)	pass@1	+19.9	$[10.7, 29.1]$
	pass@3	+25.5	$[16.8, 34.2]$
	pass@5	+5.6	$[-0.5, 11.7]$
	MRR	+20.6	$[13.5, 27.5]$
	MAP	+16.7	$[10.4, 22.9]$
	NDCG@5	+15.3	$[9.5, 21.1]$

Table 3: Paired bootstrap intervals for the full method versus root seed blocks on the two main tasks. Deltas are the main-table percentage-point differences.

C Case Studies

We provide two qualitative case studies showing how our framework converts sparse rollout feedback into executable and inspectable protein-editing rules. The goal of these case studies is to illustrate the mechanism of knowledge evolution: feedback from evaluated candidates is distilled into local program changes that can be inspected, attributed, and revised further.

C.1 Solubility Case Study: Residue-Context Rules for Tail Editing

We first consider the solubility/SaProt task and the selected terminal low-complexity editing strategy `StrategyTailLowComplexityEditor`. The selected node corresponds to block 4, node 2 in the revision run. It was chosen by the final LCB-based selection procedure, achieved an action success rate of 54.4% over 90 observations, and produced successful single-block held-out mutations for 96 of 196 proteins.

Before revision. The parent strategy used a broad C-terminal editing heuristic. In the revision packet that produced this node, each edited case proposed a single-residue C-terminal deletion. This operation was only weakly reliable: 12 of 23 edited cases improved the SaProt solubility score, while 11 decreased it. The failure mode was that deletion received an almost flat bonus, so the strategy often preferred deletion once a terminal window passed the composition gate.

Parent terminal-editing block

```
for each terminal window:
    lc = count_low_complexity_residues(window)
    hp = count_hydrophobic_residues(window)

    if lc < 2 and hp < 3:
        skip this window

    base = 0.45 + 0.12 * lc + 0.18 * hp
```

```
    add DEL candidate:
        score = base + 0.25              # flat deletion preference

    add INS candidate:
        residue = "D" if hp >= lc else "S"
        score = base

    add SUB candidate:
        replacement = residue_relief_map[terminal_residue]
        score = base - 0.10

return candidates sorted by score
emit the top candidate
```

554

555 **Feedback pattern.** The rollout evidence showed that deletion quality depended on two local 556 residue-context factors. First, the deleted terminal residue mattered: deleting terminal residues such 557 as I, Y, M, A, and C was often safe or beneficial, whereas deleting G, D, E, K, or H was more risky. 558 Second, deletion exposes the penultimate residue as the new C-terminus. Exposing polar or small 559 residues such as Q, T, E, N, D, and G was often beneficial, while exposing W, K, or hydrophobic 560 residues was harmful. Thus, the useful rule was not simply "delete low-complexity tails", but to score 561 deletion according to the residue removed and the residue exposed.

Table 4: Residue-context rules learned by the revised terminal-editing block.

Rule component	Positive examples	Negative examples
Deleted terminal residue	I, Y, M, A, C	G, D, E, K, H
Newly exposed C-terminus	Q, T, E, N, D, G	W, K, hydrophobic residues

562 **After revision.** The revised block replaces the flat deletion preference with two explicit residue-563 context scoring tables: one for the dispensability of the deleted terminal residue and one for the 564 desirability of the newly exposed C-terminal residue.

Revised terminal-editing block

```
CTERM_DISPENSABILITY = {
    I:+0.25, Y:+0.20, M:+0.15, A:+0.10, C:+0.10,
    H:-0.15, K:-0.15, E:-0.20, D:-0.25, G:-0.40, ...
}

EXPOSED_CTERM_BONUS = {
    Q:+0.15, T:+0.10, E:+0.08, N:+0.08, D:+0.05, G:+0.05,
    L:-0.10, A:-0.10, I:-0.10, F:-0.10, W:-0.25, K:-0.20, ...
}

for each terminal window:
    lc = count_low_complexity_residues(window)
    hp = count_hydrophobic_residues(window)

    if lc < 2 and hp < 3:
        skip this window

    base = 0.45 + 0.12 * lc + 0.18 * hp

    add DEL candidate:
        deleted = terminal_residue
        exposed = penultimate_residue
        score = (base + 0.05
                 + CTERM_DISPENSABILITY[deleted]
                 + EXPOSED_CTERM_BONUS[exposed])

    add INS candidate:
```

565

```
        residue = "D" if hp >= lc else "S"
        score = base

    add SUB candidate:
        replacement = residue_relief_map[terminal_residue]
        score = base

return candidates sorted by score
emit the highest-scoring diverse candidates
```

Equivalently, the deletion score becomes

$$s_{\mathrm{DEL}} = s_{\mathrm{base}} + b_{\mathrm{DEL}} + \phi_{\mathrm{delete}}(x_L) + \phi_{\mathrm{expose}}(x_{L-1}),$$

where x_L is the original C-terminal residue and x_{L-1} is the residue exposed after deletion. This makes the learned rule directly inspectable: a deletion can be traced to the residue being removed, the residue being exposed, or both. The same representation also exposes remaining shortcuts. For example, later descendants overused C-terminal `INS=D` in hydrophobic-rich windows; because the behavior was encoded in source code, the error could be localized to a specific action rule.

C.2 Stability Case Study: Learning When Not to Edit Flexible Loops

We next consider the stability/Rosetta task and the selected loop-editing strategy `StrategyLoopMicroIndelStabilizer`. The selected node corresponds to block 4, node 18 in the run used for the main-table stability/Rosetta result. The final LCB-based checkpoint selected the strategy set

$$\{(0,0),(1,1),(2,26),(3,19),(4,18)\},$$

which produced the main-table Rosetta row. This node is not used here as the sole explanation for the full-method improvement. Instead, it provides a clear mechanistic example of learning a negative rule: when an initially plausible loop indel should be suppressed.

Before revision. The parent strategy targeted flexible, low-contact residues and terminal residues, then emitted small deletions, insertions, or conservative substitutions. This is a reasonable initial mechanism because loops and termini can sometimes absorb small length edits. However, in the revision packet that produced node 18, the parent node accepted all 10 candidate cases but only 5 improved under Rosetta. The issue was not simple threshold calibration; the action-side rule was too permissive.

Parent loop micro-indel block

```
for each flexible or terminal residue:
    td = distance_to_nearest_terminus(position)
    fc = count_flexible_residues(local_window)
    cc = contact_count(position)
    bf = backbone_bfactor(position)

    if residue is not flexible and td > 2:
        skip this residue

    score = 0
    if cc <= 6:
        score += 0.90
    elif cc <= 9:
        score += 0.40
    else:
        score -= 0.45

    score += flexible-cluster, terminal, and small-residue bonuses

    is_interior_high_cluster = (fc >= 3 and td > 12)
    if is_interior_high_cluster and cc > 5:
        is_interior_high_cluster = false
```

```
    interior_penalty = 0.45 if is_interior_high_cluster else 0
    is_buried_del = (td >= 100)
    is_omega_prone_ins = (bf >= 0.95 and td >= 40 and fc >= 2)

    delete_score = score + deletion bonuses
    insert_score = score + insertion bonuses
    sub_score = score - substitution discount

    if is_interior_high_cluster:
        penalize INS, DEL, and SUB
    if is_buried_del:
        penalize DEL
    if is_omega_prone_ins:
        penalize INS

if risky interior DEL and risky interior INS are within 10 residues:
    lower the condition score

emit the highest-scoring spaced edits
```

588

589 **Feedback pattern.** Rosetta feedback separated safe extreme-terminal deletions from harmful
590 nonterminal loop deletions. Extreme-terminal deletion could be beneficial: for example, deletions
591 at $td = 1$ or $td = 0$ reduced `fa_rep` in successful cases. In contrast, harmful cases involved near-
592 terminal or semi-buried deletion contexts. Q6PWL5 proposed DEL@44 and INS@62 with $td = 43$
593 and $fc = 4$, worsening the Rosetta score by 580.0 and increasing `fa_rep` by 703.2. P9WMU7
594 showed a similar semi-buried deletion at $td = 44$, with `fa_rep` increasing by 1558.4. Q5CYE4
595 showed that even a near-terminal deletion can be risky: DEL@3 had $td = 2$ and $fc = 3$, and
596 increased `fa_rep` by 729.7.

597 The useful rule was therefore not "edit flexible loops". Edit safety depends on whether the deletion is
598 at the extreme terminus, in a near-terminal high-flexibility patch, in a semi-buried loop region, or
paired with a nearby insertion.

Table 5: Compact summary of the rules encoded by the revised Rosetta loop-indel node.

Rule type	Revised program behavior	Intended effect
High-flex loop gate	Expands the risky-loop condition from $fc \geq 3, td > 12$ to $fc \geq 3, td > 8$, and increases the penalty from 0.45 to 0.55.	Suppress borderline interior or near-terminal loop edits.
Deletion vetoes	Adds penalties for semi-buried deletions ($30 \leq td < 100$) and near-terminal high-flex deletions ($2 \leq td \leq 8, fc \geq 3$).	Avoid deletions that create packing voids or local clashes.
DEL–INS conflict	Widens the risky DEL–INS proximity check from 10 to 25 residues.	Avoid paired edits that induce backbone strain.

599

600 **After revision.** The revised node translates these observations into explicit veto rules. It lowers
601 the threshold for high-flexibility interior regions, adds penalties for semi-buried and near-terminal
602 high-flexibility deletions, and widens the DEL+INS proximity safeguard.

Revised loop micro-indel block

```
for each flexible or terminal residue:
    td = distance_to_nearest_terminus(position)
    fc = count_flexible_residues(local_window)
    cc = contact_count(position)
    bf = backbone_bfactor(position)

    score = low-contact, flexibility, terminal, and residue-type score
```

603

```
    is_interior_high_cluster = (fc >= 3 and td > 8)
    if is_interior_high_cluster and cc > 5:
        is_interior_high_cluster = false

    interior_penalty = 0.55 if is_interior_high_cluster else 0

    is_semi_buried_del = (30 <= td < 100)
    is_near_terminal_high_fc_del = (2 <= td <= 8 and fc >= 3)
    is_buried_del = (td >= 100)
    is_omega_prone_ins = (bf >= 0.95 and td >= 40 and fc >= 2)

    delete_score = score + deletion bonuses
    insert_score = score + insertion bonuses
    sub_score = score - substitution discount

    if is_interior_high_cluster:
        penalize INS, DEL, and SUB more strongly
    if is_semi_buried_del:
        delete_score -= 0.45
    if is_near_terminal_high_fc_del:
        delete_score -= 0.40
    if is_buried_del:
        delete_score -= 0.60
    if is_omega_prone_ins:
        insert_score -= 0.40

risky_del_positions = interior, semi-buried, or near-terminal high-flex DELs
risky_ins_positions = interior high-flex INS candidates

if a risky DEL and risky INS are within 25 residues:
    lower the condition score

emit the highest-scoring nonconflicting edits
```

This case shows that useful knowledge evolution can be negative: the selected node improves the strategy primarily by learning when not to perform a plausible indel. It also shows how Rosetta energy terms can be distilled into executable rules. The revised block does not use `fa_rep` at inference time; instead, rollout-level `fa_rep` failures are converted into terminal-distance, flexible-cluster, and proximity checks. These are Rosetta-proxy signals rather than wet-lab validations, so the case should not be read as a universal rule for protein indels. Its purpose is to show that sparse scalar feedback can be converted into concrete, auditable local decision logic.

D Additional Experiments and Ablation Details

D.1 RQ3 Revision-Budget Protocol

RQ3 evaluates whether the revision sandbox can convert stored rollout evidence and code-agent work into better executable strategy knowledge. For the resource-controlled RQ3 endpoint experiments, every task is run with the same backward-phase budget: `max_revision_jobs=100` and `revision_batch_size=25`. Thus each task receives at most 100 completed strategy-revision jobs, and one node becomes eligible for a revision job only after it has accumulated 25 new primary feedback records since its previous revision cursor.

The batch-size parameter controls evidence granularity, not held-out evaluation. A batch of 25 means that the revision packet is centered on 25 new scored cases for the target node. The trainer may also attach a small bounded set of auxiliary rejected or boundary cases when they are useful for diagnosing screening errors, but these auxiliary cases do not change the primary batch-size budget. The same held-out split, backend-valid aggregation rule, top-5 candidate interface, and ranking metrics are used at all RQ3 checkpoints.

The endpoint reported for a task is produced after the revision loop reaches the 100-revision budget. At checkpoint time, the strategy tree is frozen, one node is selected from each strategy family by the inference selection rule, and the resulting strategy set is evaluated on the held-out proteins. This

keeps training-time evidence collection separate from held-out scoring: held-out proteins are used only for evaluation, not for selecting or revising blocks.

D.2 Agentic Revision Loop Implementation

The agent loop is the implementation of the backward phase. It operates on one strategy node at a time and is append-only: a successful revision creates a child node in the same strategy tree, while the parent remains available for future sampling and comparison.

Forward evidence collection. During training, the forward loop samples a protein and selects strategy nodes from the current mutation trees. A selected block is screened by its local gating logic and then run as an isolated single-block candidate generator. The current training implementation can force execution after a screening rejection to collect counterfactual evidence; such cases are marked separately and are used to identify missed opportunities and true negatives. Each scored rollout stores the protein input, selected block identity, proposed mutations, applied mutations, mutation warnings or collisions, runtime trace, parent score, candidate score, raw score change, normalized score gain, task-success label, and backend-specific diagnostics.

Revision scheduling. Each node maintains cursors over feedback that has already been used for revision. When a node has at least 25 new primary feedback records and no queued or running revision job, the trainer queues a revision job for that node. The queued job records the target block id, node id, epoch, batch label, feedback cursor range, primary feedback batch, and any auxiliary rejected feedback selected for diagnosis. Evidence remains stored on the node after the job is queued; it is not consumed or deleted.

Sandbox construction. For each revision job, the backend creates an isolated sandbox. The sandbox contains the current target block, the runtime contract, task-specific self-check cases, a compact revision packet, representative case cards, aggregate summary statistics, diagnostics, a case index, compact and detailed per-case evidence records, a history summary, and a revision-lineage index. Rosetta stability sandboxs may expose structure and energy-term artifacts; SaProt solubility sandboxs remain sequence-only. The agent is instructed to treat the current batch as primary evidence and to use historical summaries only as secondary context.

Agent work loop. The revision-loop budget κ controls how much internal agent work is available before the final code is committed. When $\kappa = 1$, the agent performs one final pass that reads evidence, writes the analysis artifact,writes the revised block artifact, and runs the self-check. When $\kappa > 1$, the first $\kappa - 1$ passes are scratch diagnostic passes. These pre-final passes write scratch notes; they are not allowed to write the final analysis or final code. The final pass reads and critiques those notes, then produces the final analysis and revised block.

D.3 LLM Proposal Prompts

Stability Prompt

```
Instruction. Propose novel mutations to optimize protein stability
from the information above. Be creative when needed with insertions or
deletions, preserve activity or function as much as possible, and provide
protein-specific reasoning for every mutation.

Formation. Give step-by-step reasoning for block selection and
mutation choice. Explicitly cite the strategy block ID that inspired each
decision, then return the edits as \texttt{SUB}, \texttt{DEL}, or
\texttt{INS} operations referenced to the original sequence.

\item[Block 0: System architecture.] Treat the task as hypothesis-driven
exploration. Select a strategy block, apply its physical lens to the
current protein, and propose a mutation that maximizes that block's
objective.
```

```
[Block 1: Persona. Act as a structural biophysicist focused on
thermodynamic stability through energy minimization.

\item[Block 2: Reasoning.] Follow a hypothesis-verification workflow:
observe the flaw, propose a fix, and mentally simulate the energetic
impact before committing.

\item[Block 3: Mutation strategy.] Be creative toward sequences.

\item[Block 4: Constraints.] Do not alter active-site residues or
introduce uncompensated buried charges.
```

664

Solubility Prompt

```
Instruction. Propose novel mutations to optimize protein stability
from the information above. Be creative when needed with insertions or
deletions, preserve activity or function as much as possible, and provide
protein-specific reasoning for every mutation. Act as a mutation
hypothesis tester focused on exploration: select one strategy block, apply
that block's physical lens as the presumed bottleneck for the current
protein, and propose a mutation that maximizes the selected block's
objective.

Formation. Give step-by-step reasoning for block selection and
mutation choice. Explicitly cite the strategy block ID that inspired each
decision, then return the edits as \texttt{SUB}, \texttt{DEL}, or
\texttt{INS} operations referenced to the original sequence.

Block 0: Core packing. Treat stability as a packing-density problem:
locate small buried residues such as \texttt{A}, \texttt{V}, or
\texttt{G}, then mutate them to larger hydrophobic residues such as
\texttt{L}, \texttt{I}, \texttt{F}, or \texttt{M} to fill putative voids
and increase van der Waals contacts.

\item[Block 1: Loop rigidification.] Treat flexible non-helical or
non-sheet regions as entropic liabilities: target loop regions enriched in
\texttt{G} or \texttt{S}, and rigidify them with \texttt{P} when the
local secondary-structure context makes this plausible.

Block 2: Surface engineering. Treat exposed hydrophobic or neutral
polar residues as solvent-interaction defects: replace likely exposed
\texttt{V}, \texttt{I}, \texttt{L}, \texttt{F}, or \texttt{W} residues
with charged residues such as \texttt{K}, \texttt{R}, \texttt{E}, or
\texttt{D} to increase favorable surface charge.

\item[Block 3: Chemical cleanup.] Treat chemically reactive residues as
long-term stability weak links: locate \texttt{C} and \texttt{M}, then
neutralize them with chemically inert replacements such as
\texttt{S}, \texttt{A}, or \texttt{L} when they are not required for
function or structural integrity.

Block 4: Conformational normalization. Treat unusual residues in
their local chemical environment as possible stability outliers: replace
context-inconsistent residues, such as partially buried charged residues
or bulky residues in tight turns, with canonical high-propensity residues
such as alanine for helices or valine for sheets.
```

665

E Limitations

667 Despite the consistent gains reported in Section 3, KnowEvo has several limitations that will lead to
668 natural extensions of the framework.

669 First, our evaluation uses computational rewards rather than wet-lab validation, so the reported im-
670 provements should be interpreted as proxy-optimization results. Rosetta energy and SaProt solubility
671 predictors capture useful aspects of stability and solubility, but they may not fully reflect experimental
672 fitness, expression yield, or downstream function. KnowEvo also starts from human-written seed
673 expertise blocks; although the method revises and selects among them automatically, performance
674 may depend on the coverage and quality of the initial mechanisms. In addition, the revision sandbox
675 introduces extra LLM-side computation beyond reward calls, so our main comparisons emphasize
676 reward-budget matching rather than identical inference cost. Finally, the current experiments cover
677 two protein-design objectives; extending the framework to activity, binding, and multi-objective
678 constraints remains future work.

679 # F Computing Resources

680 All experiments were run with local reward-backend workers and external LLM API calls for
681 program generation or code-agent revision. For stability/Rosetta, the backend used a GPU-backed
682 structure/scoring service with approximately 10GB peak GPU memory per worker, together with
683 CPU execution for Rosetta energy evaluation. For solubility/SaProt, the predictor service used
684 approximately 3GB peak GPU memory per worker. Backend concurrency was controlled by the
685 server-side worker configuration.

686 # G Method Details

687 ## G.1 Initial Expertise Blocks

688 Our method starts from a small set of root expertise blocks. Each block represents one protein-design
689 idea as an executable editing rule. A block reads the current protein sequence, task metadata, mutation
690 limits, protected residues, and task-specific artifacts when available, then proposes normalized edits:

$$m \in \{\textsc{Sub}(p, a, a'), \textsc{Del}(p_1, p_2), \textsc{Ins}(p, u)\}.$$

691 Here p is a one-based residue position, a and a' are source and target amino acids, and u is an inserted
692 amino-acid string. Each proposed edit is checked against the task mutation schema before reward
693 evaluation.

694 **Stability/Rosetta root blocks.** For the Rosetta stability task, the initial blocks use residue-level
695 structural signals when a wild-type structure is available. The main signals are local contact count,
696 nearby residues, terminal distance, and backbone B-factor. These signals are used to choose candidate
697 residues and edit types; the final reward is still computed by Rosetta after applying the mutation.

698 **Solubility/SaProt root blocks.** For the SaProt solubility task, the initial blocks are sequence-only.
699 They use local residue class, hydrophobic density, aromatic density, charge distribution, terminal
700 distance, and low-complexity composition.

701 ## G.2 Selection Criterion for Inference

702 Training uses stochastic node sampling to allocate evaluation budget across both strong and under-
703 tested strategy variants. Inference uses a more conservative rule: after training, we freeze one node
704 from each strategy family using the lower confidence bound (LCB) of its posterior success probability.

705 For node N_i, let $N_{\text{success},i}$ be the number of successful evaluations assigned to the node and $N_{\text{total},i}$
706 be the total number of evaluated cases. With a uniform Beta prior, the posterior is

$$\theta_i \mid \mathcal{D}_i \sim \text{Beta}(\alpha_i, \beta_i), \qquad \alpha_i = N_{\text{success},i} + 1, \quad \beta_i = N_{\text{total},i} - N_{\text{success},i} + 1.$$

707 The LCB score is the q-th percentile of this posterior, with $q = 0.05$:

$$S_{\text{LCB}}(N_i) = F^{-1}(q; \alpha_i, \beta_i), \tag{2}$$

708 where F^{-1} is the percent point function of the Beta distribution.

709 For each strategy family b, the selected node is

$$N_b^\star = \arg\max_{N_i \in V_b} S_{\text{LCB}}(N_i).$$

710 The final inference context is

$$K_{\text{final}} = \{N_b^\star\}_{b=1}^{B}.$$

711 This rule favors nodes with reliable evidence over nodes that only have a high empirical success rate
712 from a small number of trials.

713 ### G.3 Sandboxed Agent Interface

714 The backward phase uses an agent as a local block-revision operator. For a target strategy node v,
715 the operator receives the parent block b_v, a task-level contract, and a bounded sandbox state $\mathcal{S}_v$. It
716 returns a written revision rationale and a candidate child block $b_{v'}$:

$$(a_{v'}, b_{v'}) = R_\phi(b_v, \mathcal{S}_v).$$

717 The rationale $a_{v'}$ states the evidence pattern and the intended mechanism-level change. The candidate
718 child block $b_{v'}$ must preserve the same input-output interface as the parent and emit mutation
719 operations in the same normalized schema.

720 For each revision, the implementation materializes a fresh sandbox centered on one parent strategy.
721 The sandbox contains the target executable block, a runtime contract, a task brief, curated trajectory
722 evidence, short historical summaries for the same strategy family, and local diagnostic tools. It
723 does not expose unrestricted project context or the full training trace. This creates a clear evidence
724 boundary: every child node is traceable to one parent block, one current evidence batch, and one task
725 contract. The runtime contract records the visible input schema, mutation budget, protected residues,
726 allowed mutation operations, available task artifacts, and expected block identity, so the agent revises
727 the same interface that is used during rollout evaluation.

728 **Hierarchical trajectory organization.** The sandbox is organized so that the agent sees compact
729 evidence first and opens detailed evidence only when it changes the diagnosis. This is important
730 because a single rollout can contain sequence context, mutation normalization, structured trace events,
731 backend scores, task feedback, and backend-specific diagnostics. Pasting all of this into the prompt
732 would make the revision problem harder rather than more precise.

733 Each rollout trajectory is therefore represented through linked views rather than a flat field dump. The
734 input-and-edit view keeps the visible protein context, mutation windows, and the edits that survive
735 schema normalization. The trace view keeps a compact digest of the block's internal reasoning,
736 such as which local features were measured, which candidates were ranked, and why an edit was
737 emitted or skipped. The feedback view connects the resulting sequence to the task score, normalized
738 improvement signal, and task-specific feedback. The agent can compare these views across many
739 cases before reading any large detail record.

740 **Task-specific evidence.** For Rosetta stability, the sandbox can include wild-type structure artifacts
741 and structure-derived residue summaries. The compact evidence reports whether structure was
742 used when the rollout was produced and whether a local structure artifact is available for revision.
743 Rosetta feedback is also split into a simple card and a detailed diagnostic layer: the card exposes the
744 pre-mutation and post-mutation stability scores, total energy change, and the largest favorable and
745 unfavorable energy-term deltas; the detailed layer can be queried for per-residue energy windows
746 around edited sites. Thus the agent can reason about terms such as repulsive contacts, attractive
747 interactions, solvation, torsion, and hydrogen-bond changes without reading raw structure text by
748 default.

749 For SaProt solubility, the sandbox remains sequence-only. The agent receives local sequence windows,
750 residue-class changes, mutation-depth information, predicted solubility changes, and task feedback
751 from the solubility evaluator. The instructions explicitly keep this task separate from structure-aware
752 reasoning, so the revision must be justified by sequence-level evidence.

753 **Skill and instruction design.** The sandbox includes a block-revision skill that specifies how
754 evidence should be used. The skill is not a source of new biological rules; it is a local discipline for
755 the agent. It requires the agent to start from aggregate evidence, inspect representative successful
756 and harmful cohorts, use historical summaries only as background, write a mechanistic diagnosis
757 before editing, and keep the revised block deterministic and task-aware. For Rosetta, the skill directs
758 the agent to use targeted energy queries instead of reading large raw payloads by default. For

759 sequence-only tasks, it prevents structure-based explanations unless explicit structure evidence is
760 present.

761 The prompt itself has three parts. The role instruction defines the agent as a protein-design revision
762 specialist and tells it to rely on local evidence first. The task instruction binds the target strategy,
763 active objective, mutation constraints, available artifacts, and curated evidence layers. The output
764 contract requires a written rationale and one revised executable block. The rationale must connect
765 proposed changes to helpful and harmful rollout patterns; the revised block must keep the same
766 strategy identity, preserve the same interface, respect the mutation schema, and avoid hidden runtime
767 inputs.

Agent role instruction

```
You are the biological revision code agent.

Revise exactly one task-scoped strategy block using only the current sandbox.
Use local sandbox evidence first. If the block-revision skill is available,
call it before substantive work and follow it as the primary evidence-use
policy. This instruction is navigation only: do not assume structure text,
sequences, reports, or block code are embedded in the prompt.

You must ground the revision in explicit evidence analysis before editing code.
Act like a senior protein-design and enzymology researcher. Do not behave like
a metric-tuning assistant that only nudges thresholds or rewrites prose. Prefer
mechanistic hypotheses, cohort analysis, and task-aware reasoning over
superficial parameter changes.

Hard requirements:
- Work only inside the current sandbox.
- Keep the class name unchanged.
- Respect the runtime contract exactly.
- Do not ask the user questions.
- Run the local self-check before finishing.
- Keep the analysis bounded, not shallow: start from the curated evidence,
  then inspect only the specific detailed cases that materially change the
  diagnosis.
- Summarize helpful and harmful cohort patterns before proposing code changes.
- Use revision-lineage summaries only as optional background context; the
  current batch stays primary.
- Do not stop at contract cleanup alone when the evidence supports a stronger
  task-aware heuristic.
- For sequence-only tasks, do not make structural claims unless the sandbox
  contains explicit structural evidence that you actually inspected.
- Avoid broad context crawling, repeated reads of the same material, and
  repeated attempts to access information outside the sandbox contract.
- Write the analysis artifact before writing the revised block artifact.
```

768

Block-revision skill instruction

```
Use this skill when revising one strategy block from an isolated sandbox.

Evidence-use policy:
1. Read the task brief first for the workspace-specific objective, evidence
   layout, and output contract.
2. Read the target block and runtime contract before editing code.
3. Start with the front-door revision packet, representative case cards,
   aggregate summary, historical summary, diagnostics, case index, and
   revision-lineage index.
4. Use the aggregate evidence to summarize helpful versus harmful cohort
   patterns before proposing code changes.
5. Treat historical summaries and revision lineage as optional background
   context only. Keep the current batch evidence primary.
6. Open only the specific detailed case records referenced by representative
   cases or strongly suggested by the aggregate diagnostics.
```

769

```
7. Inspect at least two successful cases, at least three harmful accepted
   cases, and one auxiliary or boundary case when available before finalizing
   the diagnosis.
8. For larger cohorts, use scratch analysis only when it materially sharpens
   the diagnosis: mutation-position hotspots, residue-class changes,
   mutation-depth effects, sequence-window motifs, helpful-versus-harmful
   comparisons, and simple cohort tables.
9. For sequence-only tasks, do not invent structural explanations. Only
   discuss structure if the sandbox includes explicit structural evidence that
   you actually inspected.
10. Do not open raw Rosetta provider payloads, per-residue records, or PDB
    structures by default. Use the targeted evidence-query tool for residue-
    level energy checks.
11. Never paste raw structure text into the analysis or final code.
12. Write the analysis before editing the final class. The analysis should
    include contract issues, cohort pattern summary, helpful patterns, harmful
    patterns, mechanistic hypothesis, concrete design rules implied by the
    evidence, whether historical lineage agrees with or contradicts the
    current batch, and the self-check result.
13. Keep the revised class deterministic, task-aware, and compatible with the
    runtime contract.
14. Do not stop at contract cleanup alone when the cohort evidence supports a
    better task-aware heuristic.
15. Treat the task as scientific diagnosis and redesign, not parameter tuning.
    Explain the mechanistic hypothesis behind each important code change.

Evidence priorities:
- Prefer local sandbox evidence over external search.
- Prefer biological reasoning over score chasing.
- Use curated evidence first and pull detailed case evidence only when it
  changes the diagnosis.
- Treat aggregate summaries, diagnostics, and the case index as first-class
  local evidence, not optional extras.
- Treat revision lineage as a routing aid for historical analyses, not as
  authority over the current cohort.
```

Built-in sandbox tools. The agent has local read, search, edit, and command-execution tools scoped to the sandbox. These tools are intentionally useful but bounded: the agent can inspect the target block and evidence, run small scratch analyses over the local case records, and produce the revised block, but it is not expected to search the entire project.

The sandbox also provides a dedicated evidence-query helper. Its operations cover four common needs: listing cases by outcome group, opening a compact case summary, ranking Rosetta energy-term deltas, and extracting per-residue energy-term windows around selected positions. A local contract checker is available to test the revised block against the required interface, mutation schema, task-specific validation cases, and trajectory reporting requirement. These tools make the agent's evidence use targeted: simple summaries guide the revision, while heavy Rosetta diagnostics are pulled only when they answer a specific mechanism question.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction explicitly state the paper's core claim: that elevating externalized, persistent knowledge to the optimization target improves protein optimization. The contributions are clearly enumerated in the introduction. Experimental results in Section 3 provide empirical support across protein optimization tasks.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations of the work are stated in Appendix E.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: This paper does not include theoretical results.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We disclose all information needed for reproduction in Appendix B.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Data, code, and instructions are available at `https://anonymous.4open.science/r/KnowEvo-6E3B/`.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: We Provide experimental details in Section 3 and Appendix B.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes].

   Justification: Please refer to Appendix B

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Please refer to Appendix F

829 9. **Code of ethics**

830 Question: Does the research conducted in the paper conform, in every respect, with the
831 NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

832 Answer: [Yes]

833 Justification: We have read and understood the code of ethics and have made every effort to
834 adhere to it.

835 10. **Broader impacts**

836 Question: Does the paper discuss both potential positive societal impacts and negative
837 societal impacts of the work performed?

838 Answer: [N/A]

839 Justification: Our work contributes to a novel knowledge optimization method. It does not
840 impact society at large.

841 11. **Safeguards**

842 Question: Does the paper describe safeguards that have been put in place for responsible
843 release of data or models that have a high risk for misuse (e.g., pre-trained language models,
844 image generators, or scraped datasets)?

845 Answer: [N/A]

846 Justification: We do not release any new datasets or models. Our work uses publicly available
847 LLMs, models and datasets, which does not pose risk of misuse.

848 12. **Licenses for existing assets**

849 Question: Are the creators or original owners of assets (e.g., code, data, models), used in
850 the paper, properly credited and are the license and terms of use explicitly mentioned and
851 properly respected?

852 Answer: [Yes]

853 Justification: We use publicly available LLMs, models and datasets, all of which are properly
854 cited in the paper. We adhere to the licenses and terms of use as specified by their original
855 creators.

856 13. **New assets**

857 Question: Are new assets introduced in the paper well documented and is the documentation
858 provided alongside the assets?

859 Answer: [Yes]

860 Justification: We release our codebase along with detailed documentation.

861 14. **Crowdsourcing and research with human subjects**

862 Question: For crowdsourcing experiments and research with human subjects, does the paper
863 include the full text of instructions given to participants and screenshots, if applicable, as
864 well as details about compensation (if any)?

865 Answer: [N/A]

866 Justification: The paper does not involve crowdsourcing nor research with human subjects.

867 15. **Institutional review board (IRB) approvals or equivalent for research with human**
868 **subjects**

869 Question: Does the paper describe potential risks incurred by study participants, whether
870 such risks were disclosed to the subjects, and whether Institutional Review Board (IRB)
871 approvals (or an equivalent approval/review based on the requirements of your country or
872 institution) were obtained?

873 Answer: [N/A]

874 Justification: The paper does not involve crowdsourcing nor research with human subjects.

875 16. **Declaration of LLM usage**

876 Question: Does the paper describe the usage of LLMs if it is an important, original, or
877 non-standard component of the core methods in this research? Note that if the LLM is used
878 only for writing, editing, or formatting purposes and does *not* impact the core methodology,
879 scientific rigor, or originality of the research, declaration is not required.

880 Answer: [Yes]

881 Justification: LLMs are used in revision code agent, an important component of our method
882 in this research. The usage of LLMs (code agent) are described in Section 2.4, Section 3
883 and Appendix D.2.

Block	Mechanism	Editing rule
Glycine entropy relief	Surface or loop glycines can increase backbone flexibility and raise the entropy cost of folding. Replacing selected glycines with alanine can reduce excess flexibility with small steric cost.	Rank glycine residues by low contact count, high local backbone motion, local polar support, and absence of dense glycine/proline clustering. Propose $G \rightarrow A$ substitutions at the best ranked positions.
Proline rigidity relief	An isolated proline in a flexible, weakly packed region can introduce a local kink or restrict favorable backbone conformations. Removing this rigidity can sometimes improve local packing.	Rank prolines that are low-contact, locally isolated, and not part of a dense proline/glycine cluster. Replace selected prolines with alanine in more polar or flexible windows, or leucine in more hydrophobic windows.
Polar/acidic swap	Small polar and acidic residues can often be adjusted within the same chemical family without large structural disruption. Such edits can improve local polar packing or reduce unfavorable side-chain placement.	Rank low-contact polar or acidic residues in local polar clusters. Propose conservative swaps such as $D \rightarrow E$, $E \rightarrow Q$, $N \rightarrow D$, $Q \rightarrow E$, and $S \leftrightarrow T$.
Core packing rescue	A high-contact site occupied by a small or polar residue can indicate an under-packed core or weak hydrophobic neighborhood. A conservative bulkier replacement can improve packing.	Rank targetable residues by high contact count, hydrophobic neighbors, aromatic neighbors, low backbone motion, and distance from termini. Propose hydrophobic or packing-preserving substitutions, while spacing edits across distinct sites.
Loop micro-indel stabilizer	Flexible loops and termini can sometimes tolerate small length changes or conservative reshaping, but interior loop indels can also create packing voids or backbone strain.	Rank flexible or terminal residues by contact count, local flexible-residue density, terminal distance, and backbone motion. Depending on allowed mutation types, propose a small deletion, insertion, or conservative substitution, while spacing edits across the sequence.

Table 6: Root expertise blocks for the stability/Rosetta task. Each block encodes one editing mechanism and one candidate-ranking rule.

Block	Mechanism	Editing rule
Hydrophobic patch relief	Local hydrophobic clusters can reduce solubility by favoring exposed self-association or aggregation-prone patches.	Rank hydrophobic residues in windows enriched for hydrophobic or aromatic residues and poor in charged residues. Replace selected residues with more polar or less aggregation-prone amino acids.
Charge redistribution	Weakly charged surface-like windows can often be made more soluble by adding charged or polar residues, especially when the local window is hydrophobic or charge-imbalanced.	Rank residues in locally hydrophobic or charge-poor windows. Propose substitutions that add charge or polarity, such as $L \to K$, $V \to E$, $S \to K$, or $Q \to E$.
Aromatic patch breakup	Aromatic clusters can promote self-association through local aromatic stacking and hydrophobic patch formation.	Rank aromatic residues in windows with multiple aromatic residues. Replace selected aromatics with less aggregation-prone alternatives, such as $F \to Y$, $W \to Y$, $Y \to H$, or $H \to Q$.
Multi-site solubilizer	Some proteins need more than one low-risk solubility edit. Combining edits from different residue mechanisms can improve the shortlist without relying on a single residue class.	Pool candidates from hydrophobic-patch relief, charge redistribution, and aromatic-patch breakup. Select a spaced and mechanism-diverse set of edits.
Terminal low-complexity editor	Low-complexity or hydrophobic terminal tails can be poorly soluble and can sometimes be improved by a small terminal edit.	Inspect the N- and C-terminal windows. If a terminal window is low-complexity or hydrophobic, propose a one-residue deletion, a short polar insertion, or a fallback substitution, depending on the allowed mutation types.

Table 7: Root expertise blocks for the solubility/SaProt task. These blocks use sequence-level residue-class rules and do not use structural artifacts.