ChemReason: A Chemical Code-Driven Reasoning LLM via Verifiable Reinforcement Learning

Anonymous authors
Paper under double-blind review

Abstract

In chemistry, most research on large language models has centered on knowledge question answering and retrieval. However, these approaches fall short on core tasks such as open molecular generation and optimization: they lack explicit reasoning processes, and their outputs cannot be systematically verified, leading to severe issues of scientific hallucination. To overcome these limitations, we propose ChemReason, a chemical LLM grounded in generative code reasoning. Unlike non-reasoning LLMs, ChemReason is a code-driven reasoning model that provides transparent inference for molecular editing, generation, and optimization. By dynamically generating and executing chemical verification code during reasoning, the model validates each step, ensuring the scientific reliability of its answers under open-domain conditions. On the TOMG benchmark, ChemReason achieves state-of-the-art performance and demonstrates end-to-end verifiable inference for open molecular tasks. More broadly, the proposed "code–verification–reflection" paradigm offers an extensible pathway for AI for Science, providing a generalizable architecture for addressing complex scientific computing challenges.

1 Introduction

Artificial intelligence has increasingly intersected with a wide range of scientific disciplines, accelerating progress in fields such as physics, biology, and materials science Jiang et al. (2022). In chemistry, large language models (LLMs) hold tremendous but still underexplored potential. Existing chemical LLMs have primarily focused on tasks such as knowledge question answering, information retrieval, and molecular generation. Among these, the core practical capabilities most relevant to experts are *molecular generation, optimization, and editing* Li et al. (2024). However, current chemical LLMs face fundamental limitations when tackling these problems Xu et al. (2019).

First, most models *directly output answers without undergoing iterative reasoning or self-reflection*, leading to unreliable predictions. Second, the *lack of rigorous verification* in molecular tasks results in severe hallucination: generated SMILES strings often appear plausible but contain subtle errors that render them chemically invalid. Although recent "thinking"models such as DeepSeek Guo et al. (2025a), GLM Zeng et al. (2025), Doubao Guo et al. (2025b), and Qwen3 Yang et al. (2025)improve accuracy by encouraging step-by-step reasoning before producing answers, they still fail to incorporate molecular validation, and hallucination remains unsolved. With the advent of the agent paradigm, tool-augmented models (e.g., DeepSearcher Shen et al. (2009), DeepCoder Balog et al. (2016)) have emerged that autonomously invoke external tools to assist reasoning. Inspired by this trend, we consider a novel reasoning pipeline where the model alternates among *thinking, code generation, and reflection*. Specifically, the model learns to verify candidate SMILES strings by writing and executing validation code, and then updates its reasoning based on the feedback. This approach directly addresses both the lack of systematic reasoning and the severe hallucination problem in open-domain molecular generation. Fig. 1 compares the responses of the non-reasoning chemistry model with those of ChemReason. Non-reasoning models can out-

put corresponding molecular formulas, but they only look correct and are completely wrong in reality. However, ChemReason, through multiple rounds of code verification, makes the generated molecular formulas true and reliable. This is a breakthrough in the open field of chemical molecular formulas.

However, no prior work has established a systematic chemical code reasoning pipeline. Relying on manually crafted trajectories is time-consuming and infeasible at scale. To bridge this gap, this paper makes the following contributions:

- We design an **automated code-augmented reasoning framework** that requires only a small set of manually designed chemical validation functions. These functions are then diversified via LLM-driven refactoring to yield robust and varied code templates. Unlike directly generating validation code with LLMs, which often produces fragile or incomplete functions, our approach ensures high robustness and comprehensive coverage of chemical constraints.
- We propose a **two-stage training paradigm**, consisting of a cold-start phase followed by dense-reward reinforcement learning. Cold-start enables the model to quickly adapt to the *think–code–reflect* reasoning format, while reinforcement learning with carefully designed dense rewards encourages the model to write correct code, reflect on execution results, and iteratively refine outputs to produce valid chemical structures.
- We release the **ChemReason-8B** model, which achieves state-of-the-art performance on the TOMG-Bench benchmark for open-domain molecular generation, optimization, and editing tasks.

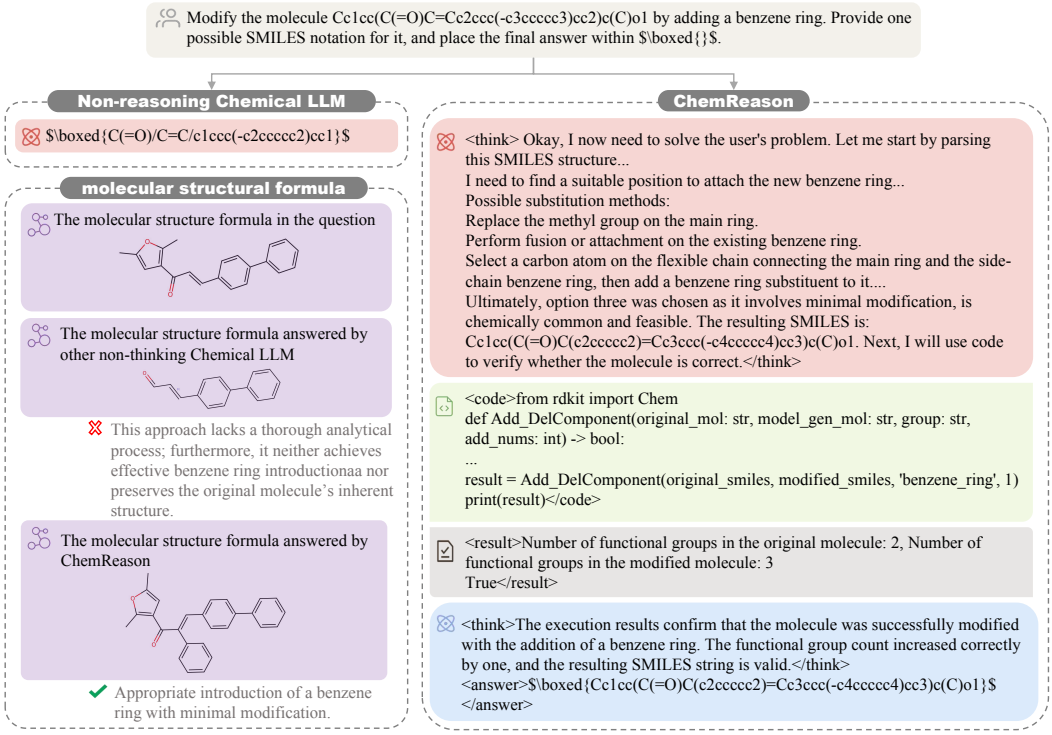

Figure 1: Response Quality: Non-reasoning Chemical LLM LLM vs. ChemReason

2 Related Work

With the rapid development of agentic paradigms, a new line of research has focused on *native tool-using models* trained with reinforcement learning Kwak et al. (2025). TORL

Li et al. (2025) demonstrates that an LLM can autonomously perform mathematical computation in Python via reinforcement learning, where reward functions and a maximum code-calling limit are jointly designed. This yields a 14% improvement over the best existing tool-integrated reasoning (TIR) models. ToolRL Qian et al. (2025) systematically studies reward function design for tool selection in RL, and introduces a principled reward scheme tailored for tool-using tasks by jointly matching tool names, parameter names, and parameter values to compute composite scores. ReTool Feng et al. (2025) highlights data construction, converting textual reasoning into adaptive code generation. Through a two-stage training strategy of cold-start followed by PPO, it achieves high code response rates, strong utilization, and self-correction ability. Finally, OTC Wang et al. (2025) addresses the limitation that most prior work emphasizes only answer correctness while overlooking tool efficiency. It proposes to reward solutions that are not only correct but also minimize the number of tool calls, thereby encouraging reasoning paths that are both accurate and concise.

Parallel to advances in standalone chemical models, a growing line of research has explored tool-augmented large models, where language models are equipped with the ability to call external tools, run code, or query knowledge bases Hammer & Zdonik (1980). General-purpose efforts illustrate that LLMs can be extended into agents capable of reasoning while acting on external environments. In scientific domains, this paradigm has inspired prototypes such as ChemCrow Bran et al. (2023), which integrates GPT-based reasoning with chemistry software like RDKit and PubChem, and SciAgent Ma et al. (2024) or Synapse, which couple LLMs with symbolic solvers and domain-specific APIs for hypothesis testing and experimental design. These studies highlight a promising trend: large models need not remain passive predictors but can actively orchestrate scientific workflows through tool use.

While tool-augmented agents have demonstrated proof-of-concept potential in chemistry, they remain far from realizing the vision of autonomous scientific assistants Chauhan et al. (2024). They still struggle to unify reasoning, tool invocation, and execution into a coherent, learnable process—an essential gap our work seeks to address.

To address these gaps, we advocate a native tool-calling paradigm where chemical tool use is treated as a first-class modeling objective rather than an add-on. We introduce ChemReason, which synthesizes high-quality tool-use trajectories via chemistry-guided sampling and difficulty-aware filtering, and trains models in two stages: a cold-start phase guided by tool feedback, and a reinforcement phase with self-critique rewards. This design equips ChemReason with the ability to think while computing, seamlessly combining reasoning with external software execution for chemistry tasks. Experiments on TOMG confirm state-of-the-art performance, highlighting the potential of domain-native tool-calling models as practical scientific agents.

3 Methodolog

The ChemReason model is trained in two stages: initial cold-start learning followed by dense reward reinforcement learning. The required training data is generated through the synthesis of chemical code routes.

3.1 Chemical code data synthesis route

We adopt a Code-Enhanced Reasoning paradigm that enables the model to interleave natural-language reasoning with executable code, run the code in an isolated sandbox, and feed verified results back into subsequent reasoning. Concretely, each trajectory is structured with `<think>` (deliberation), `<code>` (executable snippet), `<result>` (sandbox output or traceback), and a final `<answer>` segment. This design converts symbolic chemical constraints into verifiable computations and supplies instant correctness feedback to stabilize long-horizon reasoning.

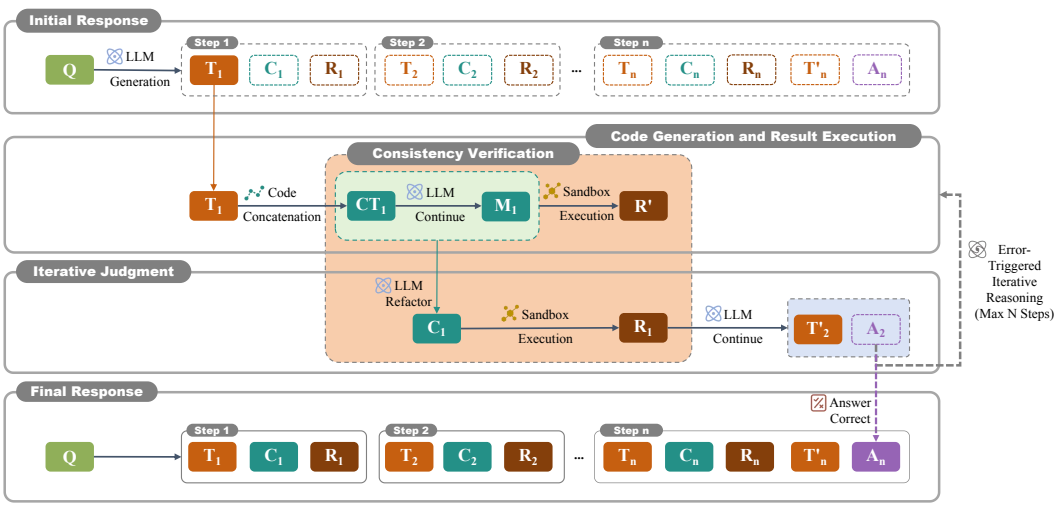

Figure 2: Data Generation Pipeline

3.1.1 Chemical code template

To address the lack of robust chemical code reasoning paths, we design a family of *task-specific code templates* that encapsulate verification utilities while leaving invocation logic and key arguments to the model. We cover three task families, including MolCustom, MolEdit, and MolOpt, with nine concrete tasks:

- MolCustom: **AtomNum** (generate by atomic counts), **BondNum** (generate by bond counts), **FunctionalGroup** (generate by functional groups);
- MolEdit: **AddComponent** (add group), **DelComponent** (delete group), **SubComponent** (substitute group);
- MolOpt: **LogP** (octanol–water partition), **MR** (molar refractivity proxy), **QED** (quantitative estimate of drug-likeness).

Eight code templates are constructed for the nine tasks, with **AddComponent** and **DelComponent** sharing one template due to symmetric verification logic. Each template exposes canonical validators (e.g., `mol_prop` for atomic/bond counts; property calculators for LogP/MR/QED) and a light-weight driver. To enhance diversity while keeping semantics invariant, we prompt a strong code model to *refactor* each complete validator (rename variables, reorder control flow, or modularize I/O), and then perform *equivalence checking* by executing both versions on identical inputs and requiring identical outcomes; only mutually consistent pairs are retained as diversified templates.

3.1.2 Data synthesis trajectory

We construct executable, verifiable trajectories that interleave reasoning and code, starting from the TOMG-Bench training pool (∼1.2M items) and subsampling fewer than 2×10^4 instances for experimentation. To broaden linguistic coverage, a portion of the samples is paraphrased into Chinese. All generations follow a unified protocol with `<think>` (deliberation), `<code>` (executable snippet), `<result>` (sandbox output or traceback), and `<answer>` (final SMILES); the exact `prompt` and format are provided in Appendix §C. Given an input specification, we synthesize a multi-round trajectory via four stages:

1. **Deliberation priming.** A base LLM produces the initial `<think>` segment; decoding halts at `</think>` via a stop token, preventing premature answers.
2. **Template completion.** We insert a task-matched chemical code template into `<code>`. Since arguments and the driver are intentionally omitted, a strong code

model (e.g., DeepSeek-R1) continues from the prefix `<think>+<code>` to synthesize (i) missing function arguments, (ii) a minimal `main` driver, and (iii) a proper `</code>` closure.

3. **Sandbox execution & feedback.** The snippet executes in a restricted Python sandbox. On success, numerical outputs (e.g., atom/bond counts or property values for LogP/MR/QED) are wrapped into `<result>`. On failure, we capture only the final traceback line to keep signals concise and place it in `<result>`. The triplet `<think>+<code>+<result>` is fed back to the model to enable self-correction.

4. **Batched continuation with difficulty control.** We replicate the updated context four times to promote diverse continuations. Two outcomes are observed: (a) the model emits `<answer>` with a SMILES string; correct answers are retained, or (b) the model triggers another `<code>+<result>` round. We cap the total code-call rounds per item to avoid runaway trajectories; the *observed code-call count* acts as a proxy for difficulty and later supports curriculum-style training.

The data construction can be seen in Fig. 2. We keep a trajectory only if (i) the final `<answer>` is *consistent* with validator outputs (answer–code agreement), and (ii) its round count falls within a pre-set window for the target difficulty bucket (0–1 calls for "easy," 2–3 for "medium," and $\geq$4 for "hard"). The resulting corpus contains $\sim 1.2\times10^4$ verified trajectories across the nine tasks (three families MolCustom/MolEdit/MolOpt); Fig. 7 reports per-task round-count distributions, from which we observe that tasks with structural editing constraints (e.g., **SubComponent**) generally induce more code calls than purely numeric property targets. Empirically, more rounds correlate with (i) harder discrete constraints (stoichiometry, valency, functional-group presence) and (ii) a higher incidence of tool-side exceptions. Logging rounds disentangles *reasoning errors* from *tooling errors* and provides calibrated supervision signals that are leveraged by our two-stage training (cold-start SFT on easy/medium, followed by RL with self-critique on hard), improving robustness without inflating prompt budgets.

$$P(<\text{ans}>|\mathcal{T}) = \prod_{t=1}^{k} \underbrace{P(\text{think}_t|s_{t-1})}_{\text{Reasoning}} \cdot \underbrace{P(\text{code}_t|\text{think}_t, s_{t-1})}_{\text{Coding}} \cdot \underbrace{\delta(\text{result}_t = \text{E}(\text{code})_t)}_{\text{Execution}} \tag{1}$$

The above process is organized into formula Eq. 1 and expressed as follows: Chemical reasoning tasks are denoted as $\mathcal{T}$, the state of the model at time t is represented by s_t, the number of reasoning steps can be expressed as k, and E stands for the chemical code executor.

3.2 Cold start method

After generating data trajectories in the previous section, we obtain reasoning paths augmented with code execution. Due to their unique output format, which differs from the base reasoning model' s native style, training directly with GRPO Shao et al. (2024) requires a large number of steps before the model learns to adapt its outputs. This leads to substantial GPU time wasted on format alignment rather than on improving reasoning. Ideally, reinforcement learning should focus on generalization, strengthening the model' s ability to handle chemical formulas and code generation. Therefore, we introduce a *cold-start stage* before reinforcement learning to align the model with the required output style. In this stage, the data portion of the synthesized trajectories is used for initialization. Within only a few steps, the loss curve shows that the model rapidly adapts to the expected format. During cold-start training, the content within `<result>...</result>` is excluded from loss computation by applying a masking strategy. The rationale is that these results depend on sandbox execution, which introduces external information not predicted by the model itself. Consequently, such parts are masked out both in the cold-start and in the subsequent reinforcement learning stage.

3.3 RL with Code

3.3.1 C-GRPO Algorithm

After the cold-start stage, the model has preliminarily mastered the required reasoning patterns. The next step is to leverage large-scale data to further generalize its performance. For this purpose, we adopt GRPO as the reinforcement learning algorithm. A key strength of GRPO lies in its scoring mechanism: instead of assigning absolute rewards, GRPO normalizes rewards within each group. In evaluation, while other RLHF approaches require a dedicated reward model, GRPO can flexibly rely on any scoring function or even a stronger LLM to assess solution quality. The corresponding reinforcement learning algorithm is shown in Fig. 3

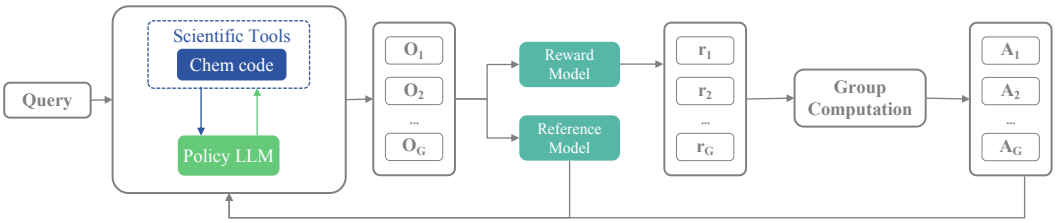

Figure 3: C-GRPO Algorithm Structure

3.3.2 Dense Reward Function

In the reward design stage, relying solely on sparse rewards is insufficient for fine-grained learning. To guide effective exploration, the model requires dense gradient signals that enable progressive learning—starting from simple problems, advancing to intermediate cases, and eventually handling more complex tasks. Our reward function consists of two main components: a code-calling reward and a direct reasoning reward. The code-calling reward is composed of three sub-rewards: (i) a format reward that enforces proper use of special tags (e.g., `<code>`, `<result>`), (ii) a correctness reward that verifies whether the executed output matches the ground truth, and (iii) a code execution success reward that encourages the generation of syntactically valid and executable programs. Together, these signals guide the model from merely producing code to producing correct and functional code that solves the given task. In contrast, the direct reasoning reward differs in format since no special code-related tags are involved. Instead, the correctness reward is computed by evaluating the content enclosed in `\boxed{}`, which directly reflects the validity of the predicted molecular structure or property. The dense reward function formula is shown as Eq. 2. The code success nums is assigned 0.5 points. The corresponding score is obtained by multiplying the total number of code calls by 0.5 based on the number of correct code executions.

$$R = \begin{cases} \begin{cases} 1 & \text{(format is success)} \wedge \text{(answer is success)} \\ 0.1 - 0.6 & \text{(format is success)} \wedge \text{(answer is error)} \wedge \text{(code success nums)} \\ 0 & \text{(format is error)} \end{cases} & \text{Tool} \\ \begin{cases} 1 & \text{(format is success)} \wedge \text{(answer is success)} \\ 0.1 & \text{(format is success)} \wedge \text{(answer is error)} \\ 0 & \text{(format is error)} \end{cases} & \text{No Tool} \end{cases} \tag{2}$$

3.3.3 RL training use code

During reinforcement learning, we guide the model to terminate reasoning at appropriate points by setting stop tokens and by imposing a maximum number of code-calling rounds to prevent excessively long contexts from distorting the reasoning process. When the model outputs `</code>`, generation halts, and the program enclosed in the `<code>` tag is extracted

and executed in a fully isolated sandbox environment. The execution result is then wrapped within `<result>` tags and concatenated with the reasoning trace, enabling the model to continue inference with feedback from real-world computation. Importantly, during RL parameter updates, the tokens inside `<result>` are excluded from the loss calculation, since these results are produced by external environment interaction rather than model prediction, thereby avoiding spurious supervision.

3.3.4 Code sandbox

The sandbox Vouvoutsis et al. (2025) module is designed to provide a secure and efficient execution environment for generated code, while supporting asynchronous scheduling and result feedback. Its core functionalities include asynchronous execution, and multi-state result handling. The sandbox executor supports asynchronous processes Rahman & Shi (2024), significantly reducing latency and improving training efficiency. For abnormal termination, only the *last line* of the error message is returned instead of the full traceback Zhang et al. (2025), avoiding excessive token consumption while still allowing the model to locate errors for debugging.

4 Experiment

The TOMG-Bench is the first comprehensive benchmark designed to evaluate the capability of LLMs in open-domain molecular generation. TOMG-Bench defines three main tasks: **molecular editing (MolEdit)**, **molecular optimization (MolOpt)**, and **custom molecular generation (MolCustom)**. Each task is further divided into three subtasks, with 5,000 test samples per sub-task. We adopt the *success rate* as the primary evaluation metric, which measures whether the generated molecules satisfy task-specific constraints.

We compare against three categories of baselines:

- **Closed-Source Models:** Claude-3.5, Gemini-1.5-pro, GPT-4-turbo, and Doubao.
- **Open-Source Models:** Qwen3-235B-A22B, DeepSeek-R1, Qwen3-32B, ChemDFM and Qwen3-8B.
- **Our Models:** Qwen3-8B-SFT (original data), C-SFT, C-SFT+TIR, and ChemReason.

4.1 Implementation details

Our training pipeline consists of a supervised fine-tuning (SFT) stage followed by a reinforcement learning (RL) stage. Both stages are powered by code-reasoning data generated via the synthesis route. We sample 5,000 code-reasoning trajectories from the synthesized corpus to adapt the base model to the target `<think>`Reasoning`</think><code>`code func`</code><result>`code result`</result>`....`<think>`Final Reasoning`</think>` Final Answer. We fine-tune `Qwen3-8B` with LLaMA-Factory under supervised learning to quickly acquire the output style required by code-enhanced reasoning. Hyperparameters are set to: learning rate 1×10^{-4}, batch size 16, epochs 1, and maximum sequence length 15,000. During this stage, tokens within `<result>` are masked out from the loss, as they originate from external sandbox execution rather than model prediction.

We build upon the `Verl` framework with customized components: an isolated code sandbox, code-calling rewards (format/correctness/execution-success), and a direct-reasoning reward. The rollout loop orchestrates multi-round "code $\rightarrow$ verification $\rightarrow$ reflection" interactions with the sandbox to realize verifiable reasoning. RL hyperparameters are: batch size 128, learning rate 1×10^{-6}, maximum response length 25,000, and 8 rollouts per prompt. Training is conducted on 32$\times$ A100-80G GPUs. To stabilize training and avoid spurious supervision, `<result>` tokens are excluded from the RL objective as well.

Table 1: Open source, Closed source and the evaluation results of our model on TOMG

Method	MolCustom			MolEdit			MolOpt			Avg.
	Aut.	Bon.	Fun.	Add.	Del.	Sub.	LogP	MR	QED	
Closed-Source Models										
Claude-3.5	0.19	0.10	0.23	0.68	0.54	0.81	0.79	0.69	0.53	0.51
Gemini-1.5-pro	0.17	0.07	0.42	0.70	0.75	0.71	0.77	0.78	0.47	0.52
GPT-4-turbo	0.17	0.07	0.21	0.69	0.72	0.77	0.76	0.73	0.39	0.50
Doubao	0.48	**0.46**	0.22	0.44	0.74	0.51	0.68	0.80	0.55	0.54
Open-Source Models										
Qwen3-235B	0.54	0.44	0.21	0.67	0.80	0.70	0.85	0.80	0.49	0.61
Deepseek-R1	0.52	0.45	0.30	0.68	0.83	0.84	0.84	0.81	0.60	0.65
Qwen3-32B	0.27	0.36	0.22	0.49	0.57	0.64	0.58	0.59	0.41	0.46
QwQ-32B	0.00	0.00	0.00	0.51	0.71	0.53	0.46	0.49	0.29	0.33
DeepSeek-32B	0.00	0.00	0.00	0.27	0.56	0.33	0.30	0.30	0.21	0.22
ChemDFM	0.03	0.07	0.06	0.35	0.18	0.33	0.36	0.42	0.21	0.22
Qwen3-8B	0.29	0.15	0.33	0.29	0.72	0.35	0.37	0.29	0.19	0.33
Our Code-Reasoning Methods										
SFT(ori)	0.18	0.16	0.40	0.42	0.65	0.68	0.60	0.50	0.35	0.44
C-SFT	0.22	0.22	0.50	0.61	0.73	0.68	0.68	0.68	0.49	0.53
C-SFT-TIR	0.31	0.27	0.48	0.67	0.56	0.74	0.80	0.80	0.61	0.58
ChemReason	0.45	0.25	**0.60**	**0.93**	**0.90**	**0.87**	**0.94**	**0.96**	**0.82**	**0.75**

4.2 Main Result

From Table 1, we evaluate closed-source models, open-source models, and our two-stage trained models on the nine chemical sub-tasks. The results show that our proposed `ChemReason` consistently achieves state-of-the-art (SOTA) performance across all open-domain molecular benchmarks. After supervised fine-tuning (SFT), the model exhibits clear improvements over the base version. Furthermore, integrating the SFT model with the code-sandbox calling mechanism yields additional gains, demonstrating the effectiveness of the code reasoning paradigm. Building on this, reinforcement learning (RL) further enhances generalization ability. After the full two-stage training pipeline, the model shows substantial improvements on MolOpt and MolEdit.

Effect of code reasoning. The results highlight that code reasoning significantly boosts model performance. With SFT alone, the model occasionally hallucinates answers, as outputs within the `<result>` tags are generated tokens rather than verified outcomes. This often misleads the model into prematurely terminating the reasoning process and producing incorrect answers. In contrast, when the SFT model interacts with the external code sandbox, the `<result>` outputs are always reliable. This enables the model to accurately determine correctness, leverage error messages for debugging, and effectively perform self-reflection and verification.

Effect of reinforcement learning. While SFT combined with sandbox interaction improves reliability, its capability is bounded by the supervised data. RL provides the critical breakthrough: given a single prompt, the model explores multiple reasoning trajectories, each interacting with the sandbox to obtain verifiable feedback. Through multi-round verification and reflection, the model converges to a robust solution. Dense reward functions Xie et al. (2024) play a central role in this stage. Unlike sparse rewards that are strictly binary (0 or 1) and insufficient for stepwise learning, our dense design assigns graded scores to each trajectory. By contrasting successes and failures within a batch, the model progressively learns optimal solution paths. This reward shaping is essential for overcoming the limitations of sparse feedback and driving substantial performance gains during RL training.

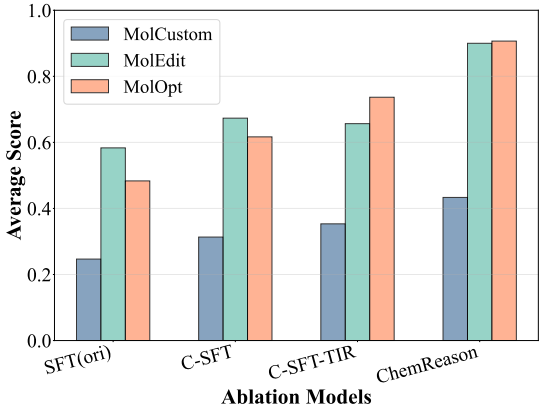

Figure 4: Ablation Study Performance Comparison

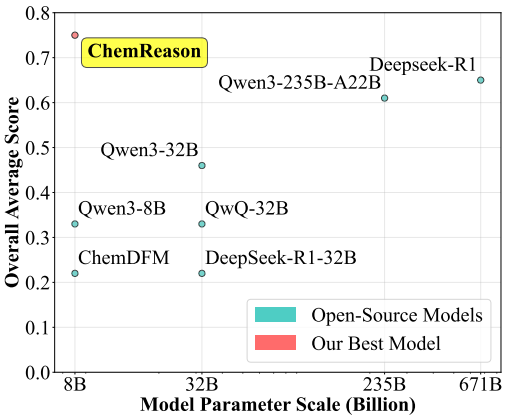

Figure 5: Model Scale vs Performance

4.3 Validity analysis

To verify that the proposed *code-enhanced reasoning paradigm* is superior to conventional reasoning, we design two controlled experiments. The first uses a Qwen3-235B distilled chemical dataset to obtain a training set with standard reasoning paths (text-only reasoning). The second leverages our synthesized code-reasoning trajectories as training data. Both variants are trained with identical hyperparameter settings under supervised fine-tuning (SFT), and then evaluated on all nine tasks of TOMG-Bench. As shown in Fig. 4, the code-enhanced paradigm consistently outperforms the standard reasoning approach, demonstrating the clear advantage of integrating tool calls and verifiable intermediate steps into the reasoning process.

Although the model after cold-start SFT can already follow the code-reasoning paradigm, its `<result>` outputs are still generated by the model itself and may suffer from hallucinations. To verify the benefit of introducing a real sandbox, we design a comparison between two variants: (i) SFT with model-generated `<result>` for reasoning, and (ii) SFT combined with Tool-Integrated Reasoning (SFT+TIR), where `<result>` is obtained from actual sandbox execution. As shown in Fig. 4, leveraging real execution results immediately improves accuracy. Moreover, if computational resources do not allow full GRPO training, performing extensive SFT under the code-reasoning paradigm and then incorporating a sandbox for real `<result>` feedback provides an economical yet effective route to boost accuracy.Finally, let's present the comparison of the parameter scale between the model we proposed and the open-source model. It achieves better performance with smaller parameters, as shown in Fig. 5

5 Conclusion

In this paper, we present `ChemReason`, a chemistry-oriented LLM enhanced with code reasoning. The trained model is able to autonomously generate chemical code during step-by-step reasoning, invoke an external sandbox for execution, and perform self-reflection on results to complete molecular validation. To address the scarcity of code-reasoning data, we propose a code-augmented data synthesis pipeline that integrates the logic of code, verification, and reflection. Furthermore, we introduce a two-stage training strategy: a cold-start phase for format alignment, followed by dense-reward reinforcement learning to accelerate the model' s mastery of chemical formulas and code generation. As a result, `ChemReason` achieves state-of-the-art performance across nine open-domain molecular tasks in TOMG-Bench. More broadly, the proposed code–verification–reflection paradigm provides an extensible pathway for AI for Science, offering a generally applicable framework for tackling complex scientific computing problems.

6 Ethics Statement

This work adheres to the ICLR Code of Ethics. In this study, no human subjects or animal experimentation was involved. All datasets used, including TOMG, were sourced in compliance with relevant usage guidelines, ensuring no violation of privacy. We have taken care to avoid any biases or discriminatory outcomes in our research process. No personally identifiable information was used, and no experiments were conducted that could raise privacy or security concerns. We are committed to maintaining transparency and integrity throughout the research process.

7 Reproducibility Statement

We have made every effort to ensure that the results presented in this paper are reproducible. All code and datasets have been made publicly available in an anonymous repository to facilitate replication and verification. The experimental setup, including training steps, model configurations, and hardware details, is described in detail in the paper.

Additionally, Chemical molecular formula dataset, such as TOMG, are publicly available, ensuring consistent and reproducible evaluation results.

We believe these measures will enable other researchers to reproduce our work and further advance the field.

References

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. Chemcrow: Augmenting large-language models with chemistry tools. *arXiv preprint arXiv:2304.05376*, 2023.

Aakriti Chauhan, C Manjunath, and Ginu Anie Joseph. A comprehensive analysis of autonomous medical assistants for improving healthcare outcomes. In *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pp. 1–6. IEEE, 2024.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025a.

Dong Guo, Faming Wu, Feida Zhu, Fuxing Leng, Guang Shi, Haobin Chen, Haoqi Fan, Jian Wang, Jianyu Jiang, Jiawei Wang, et al. Seed1. 5-vl technical report. *arXiv preprint arXiv:2505.07062*, 2025b.

Michael Hammer and Stanley B Zdonik. Knowledge-based query processing. In *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*, pp. 137–147, 1980.

Yuchen Jiang, Xiang Li, Hao Luo, Shen Yin, and Okyay Kaynak. Quo vadis artificial intelligence? *Discover Artificial Intelligence*, 2(1):4, 2022.

Beong-woo Kwak, Minju Kim, Dongha Lim, Hyungjoo Chae, Dongjin Kang, Sunghwan Kim, Dongil Yang, and Jinyoung Yeo. Toolhaystack: Stress-testing tool-augmented language models in realistic long-term interactions. *arXiv preprint arXiv:2505.23662*, 2025.

Jiatong Li, Junxian Li, Yunqing Liu, Dongzhan Zhou, and Qing Li. Tomg-bench: Evaluating llms on text-based open molecule generation. *arXiv preprint arXiv:2412.14642*, 2024.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*, 2025.

Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, Aixin Sun, Hany Awadalla, et al. Sciagent: Tool-augmented language models for scientific reasoning. *arXiv preprint arXiv:2402.11451*, 2024.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*, 2025.

Shanto Rahman and August Shi. Flakesync: Automatically repairing async flaky tests. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–12, 2024.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Derong Shen, Gaoshang Sun, Tiezheng Nie, and Yue Kou. Deepsearcher: A one-time searcher for deep web. In *2009 Ninth International Conference on Hybrid Intelligent Systems*, volume 3, pp. 273–277. IEEE, 2009.

Vasilis Vouvoutsis, Fran Casino, and Constantinos Patsakis. Beyond the sandbox: Leveraging symbolic execution for evasive malware classification. *Computers & Security*, 149: 104193, 2025.

Hongru Wang, Cheng Qian, Wanjun Zhong, Xiusi Chen, Jiahao Qiu, Shijue Huang, Bowen Jin, Mengdi Wang, Kam-Fai Wong, and Heng Ji. Otc: Optimal tool calls via reinforcement learning. *arXiv e-prints*, pp. arXiv–2504, 2025.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Automated dense reward function generation for reinforcement learning. In *International Conference on Learning Representations (ICLR), 2024 (07/05/2024-11/05/2024, Vienna, Austria)*, 2024.

Youjun Xu, Kangjie Lin, Shiwei Wang, Lei Wang, Chenjing Cai, Chen Song, Luhua Lai, and Jianfeng Pei. Deep learning for molecular generation. *Future medicinal chemistry*, 11(6):567–597, 2019.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):481–503, 2025.

Appendices

A Multi-dimensional Capability Comparison

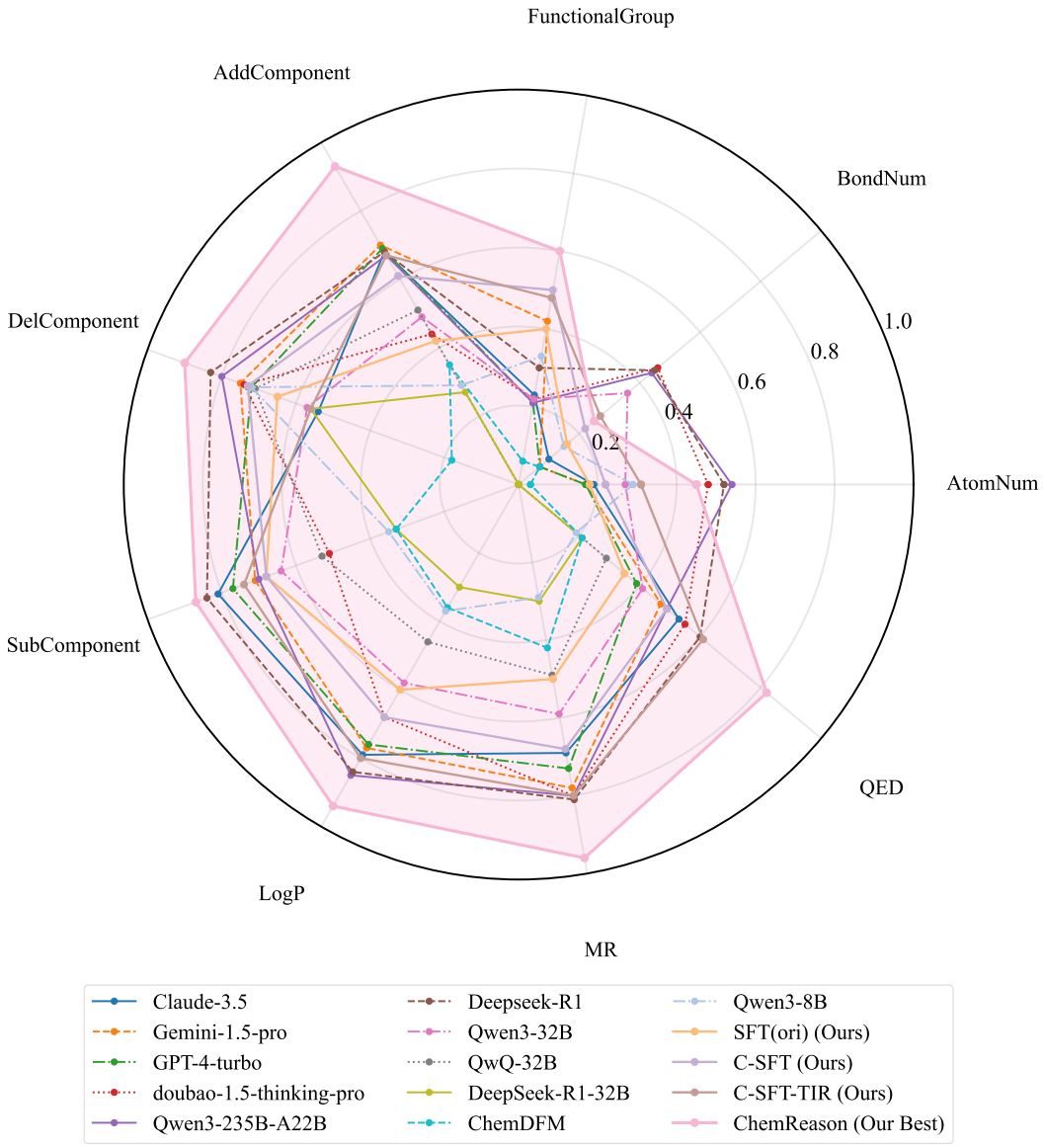

Figure 6: Multi-dimensional Capability Comparison Across Nine Molecular Generation Tasks

B Data and Round Distribution Overview

Figure 7: Overview of Data and Round Distribution

C Prompts

C.1 Data generation prompt

Data generation prompt

system prompt:

You are an AI research assistant with unlimited capacity for deep thinking. First, perform in−depth reasoning and then answer the user's question.

Current date: {datatime}

Please first clarify the core requirements of the question, then plan your reasoning before answering. During the thinking process, you may use code tools multiple times.

Thinking and Output Format Requirements

1. Your thought process must be enclosed within `<think>` and `</think>` tags, for example:

   <think>
   Analyze the user's question, break down the reasoning tasks, evaluate boundary conditions, and choose the most suitable algorithm or method.
   </think>

   − It can contain line breaks inside. Ensure logical clarity and a well−structured flow;
   − Avoid using bold, italics, headings, or other Markdown formatting. Keep it concise and clear.

2. If you need to utilize Python code to assist in solving the problem, place it immediately after `</think>` in `<code>` and `</code>` tags:

   <code>
   ```python
   // Here is the generated code snippet
   ```
   </code>

   − The code must correspond closely to the reasoning;
   − You can output code in separate steps. After addressing one subtask, output code once;
   − Make sure each code snippet can run independently or won' t fail when finally integrated;
   − After outputting `</code>`, stop generating further code immediately.

3. When users see the code you generate, they will help you execute it and send you the execution results between `<result>` and `</result>` tags; when users provide the code execution results, please continue thinking, and generate new code if necessary.

4. When you believe your reasoning is complete and you are ready to answer the user' s question and provide a comprehensive final solution, use `<answer>` and `</answer>` tags to present the fully integrated result. You should provide a structurally clear, accurate, and informative description, making appropriate use of Markdown style and formatting (though normally do not start with a Markdown heading). No reference numbering is required.

Summary of Work Method

− You must **perform your reasoning while writing code** , demonstrating a clear programming thought process;
− Avoid generating a large amount of code at once or completing tasks without analysis;
− Always focus on the user' s question by breaking it down from multiple perspectives, clarifying each design decision;
− If there is a more optimal design pattern or structural suggestion, point it out and adopt it in your reasoning.

Please respond strictly following the above approach from now on.

C.2 LLM Refactor Prompt

LLM Refactor Prompt

system prompt:

你是一名专业的Python程序员，擅长重构代码。你的任务是对用户提供的Python代码进行重构，使其结构不同但功能完全一致。

请严格遵守以下要求：

1. 保留原代码中的所有边界条件判断和异常处理逻辑。
2. 确保所有错误检查、异常捕获和边界情况处理与原代码逻辑完全一致。
3. 不得删减或简化任何安全检查、验证逻辑或条件分支。
4. 对于科学计算或专业领域代码，必须保留所有特定领域的逻辑细节。
5. 程序在所有可能的输入条件下的行为必须与原代码完全一致。
6. 仅输出重构后的代码，不添加任何解释或额外内容。

你可以调整代码风格、优化算法实现或重组代码结构，但绝不能改变代码的功能和行为。

user prompt:

{code}

C.3 Code Consistency Evaluation Prompt

Code Consistency Evaluation Prompt

system prompt:

你是一位顶级的Python代码逻辑分析专家。你的核心任务是深入比较两段Python代码及其执行结果，以判断它们在功能和核心逻辑上是否完全等价，即使它们的实现方式或代码结构有所不同。

判断标准:
1. 关注代码的核心功能和处理逻辑，而非输出格式或提示文本的表述差异
2. 如果两段代码对相同输入产生实质上相同的结果(即使输出文本表述不同)，应判断为逻辑相同
3. 重点分析数据处理流程、计算过程、条件判断和最终决策
4. 以下不影响逻辑等价性判断:
   – 变量名称差异
   – 输出文本的措辞或格式差异
   – 代码结构优化或重构
   – 注释或文档字符串的差异
5. 对于科学计算代码，需确认计算公式和方法的一致性，而非输出描述的一致性

请仔细分析提供的代码（python code1 和 python code2）及其对应的执行结果（result and report 1 和 result and report 2）。你的目标是判断 code2 是否是 code1 在保持核心逻辑完全一致前提下的有效重构或变体。

最终仅输出 True (表示两段代码逻辑完全等价) 或 False (表示逻辑不等价)，不要添加任何解释。

user prompt:

```python code1
{Original Code}
```
```result and report 1
{Execution Results of Original Code}
```

```python code2
{Refactored Code}
```
```result and report 2
{Execution Results of Refactored Code}
```

D Data Example

D.1 Data Example

Data Example: Question

Please optimize the molecule Cc1nc(NC(=O)COC(=O)C(NC(=O)c2ccco2)C(C)C)c(Cl)cc1Cl to have a higher MR value. I require its SMILES representation.

Data Example: Answer-Think

Okay, let's tackle this problem. The user wants to optimize the given molecule to have a higher molar refractivity (MR) value. First, I need to understand what MR depends on. MR is a measure of the

total polarizability of a molecule and is calculated using the formula MR = (4 /3) ∗ N ∗ , where N is Avogadro's number and is the polarizability. Each atom contributes to the MR value based on its type and bonds. So, substituents with higher MR contributions will increase the overall MR.
Looking at the original molecule: Cc1nc(NC(=O)COC(=O)C(NC(=O)c2ccco2)C(C)C)c(Cl)cc1Cl. Let's parse the SMILES to identify substituents. The core structure is a benzene ring (the 'c1...cc1' part) with two chlorine atoms attached. There's a methyl group (C), a nitrogen−containing group with amide linkages, and an oxygen−containing ester or carbonyl groups. The substituents on the benzene ring are Cl, Cl, methyl, and a complex substituent.
To increase MR, I should replace atoms or groups with those that have higher MR contributions. Chlorine has a higher MR than, say, hydrogen, but the molecule already has two Cl atoms. Maybe adding more halogens with higher MR? Bromine has a higher MR than chlorine (Br's MR contribution is around 8.7, Cl is around 6.0). So replacing Cl with Br would help.
Looking at the substituents on the benzene ring: the positions with Cl are probably at positions 2 and 4 (assuming the numbering starts at the methyl). Wait, the SMILES is Cc1nc(...)c(Cl)cc1Cl. So the benzene ring has substituents: methyl (attached to carbon 1?), then the nitrogen group, then a Cl in position 4 (since the ring is written as c1nc...c(Cl)cc1Cl). Let me draw this mentally. The ring is six−membered. The first carbon (position 1) is connected to a methyl (C), then position 2 is N connected to the other groups. The c(Cl) is position 4, and the last Cl is position 6? Maybe. Regardless, the current substituents are two Cl atoms, a methyl, and the complex substituent.
Replacing the Cl atoms with Br would definitely increase MR. Let's try that. Also, the methyl group (C) has an MR contribution of around 5.65, but if we replace it with something like iodine (but iodine is not common in organic molecules) or maybe a larger substituent. Alternatively, adding more substituents. However, adding substituents might require considering the feasibility of substitution on the ring.
Wait, the benzene ring already has four substituents: methyl (C), the NC(=O)... group, Cl, and another Cl. Wait, the benzene ring is six−membered. Let me count: the SMILES starts with Cc1nc(...)c(Cl)cc1Cl. So positions 1 (methyl), 2 (n, which is part of the ring with a nitrogen?), then position 3 (connected to NC(=O)...), then position 4 (Cl), and position 6 (Cl). Wait, maybe the ring is numbered such that the substituents are at positions 2, 4, and 6? Hmm, maybe the SMILES notation is a bit confusing. Regardless, the two Cl atoms are present. If we can replace both Cl atoms with Br, that would add more MR. Alternatively, maybe add another Br in a position that's currently a hydrogen.
But the benzene ring in the original SMILES has four substituents? Let me parse it again. The core is c1nc(...)c(Cl)cc1Cl. Breaking down the SMILES:
− Cc1: the first atom of the ring is a carbon connected to a methyl (C), so position 1.
− Then n: next is a nitrogen in the ring (position 2), which is connected to the NC(=O)... group.
− Then c(...) which is position 3, connected to the Cl at position 4 (c(Cl)cc1Cl). Wait, maybe the positions are 1 (methyl), 2 (n), 3 (c connected to NC(=O)...?), position 4 (Cl), and position 6 (Cl). So the benzene ring has substituents at positions 1, 2, 4, and 6? If that's the case, then there are four substituents. But benzene can't have more than six substituents, but four is possible.
Alternatively, perhaps the ring has three substituents: methyl at position 1, the nitrogen−containing group at position 2, and two Cl atoms at positions 4 and 6. So replacing those Cl with Br would increase MR. Let's proceed with that.
So modifying both Cl atoms to Br would give higher MR. Also, looking at other parts of the molecule, maybe replacing oxygen atoms with sulfur? Wait, but oxygen's MR contribution is lower than sulfur. For example, a single−bonded sulfur (like in thioether) has a higher MR than oxygen. So replacing an ether oxygen (O) with S could increase MR. Let's check where there are oxygen atoms in the molecule.
The original SMILES has O in several places: OC(=O) and O=C(O)... So there are ester groups, amide carbonyls. For example, the part COC(=O)... is an ester (O linking the C and the C(=O)). Replacing that oxygen (the one in the ester) with S would make a thioester, which has a higher MR. Each substitution of O with S would add more MR.
Alternatively, adding more bulky groups. For example, the amide groups could have their substituents replaced with larger alkyl chains. For instance, the C(C)C group (which is an isopropyl) could be replaced with a tert−butyl (C(C)(C)C), which has a higher MR. Also, the methyl group (C) attached to the benzene could be replaced with a larger group like ethyl or isopropyl, but that might require checking if the position allows such substitution.
Putting this together, the optimized molecule would involve:
1. Replace both Cl atoms on the benzene ring with Br.
2. Replace the ester oxygen (O in COC(=O)) with S.
3. Replace the isopropyl group (C(C)C) with a tert−butyl group (C(C)(C)C).
4. Possibly replace other oxygen atoms if feasible.
But let's check the original structure step by step.
Original SMILES: Cc1nc(NC(=O)COC(=O)C(NC(=O)c2ccco2)C(C)C)c(Cl)cc1Cl
Breaking it down:
− The benzene ring is: c1nc(...)c(Cl)cc1Cl
  − Positions 1: methyl (C)
  − Position 2: nitrogen (n) connected to a carbon (c), which is part of the NC(=O)... group.
  − Position 4: Cl (from c(Cl))
  − Position 6: Cl (from the ending cc1Cl)
So replacing the Cl at positions 4 and 6 with Br would change those to Br.
Then, looking at the ester group: COC(=O)... The O in the ester is the oxygen in COC(=O). Replacing that O with S would make it CSC(=O)...
Also, the C(C)C group (isopropyl) in the part C(NC(=O)c2ccco2)C(C)C can be replaced with a tert−butyl group: C(C)(C)C.

...

Wait, maybe these numbers are not accurate. It's better to refer to a table of MR increments for common groups.
Alternatively, considering that replacing a methyl (5.65) with a tert−butyl (C(C)(C)C) which is four carbons. If the original is isopropyl (C(C)C), which is three carbons, replacing with tert−butyl (four carbons) would add more MR. The MR for tert−butyl is higher than isopropyl.
But perhaps the largest gains would come from replacing Cl with Br, O with S in multiple positions, and increasing alkyl chain length where possible.
Let's try to modify the original SMILES step by step.
Original: Cc1nc(NC(=O)COC(=O)C(NC(=O)c2ccco2)C(C)C)c(Cl)cc1Cl
First, replace the two Cl atoms with Br:
Change c(Cl) to c(Br), and the ending Cl to Br. So the benzene ring part becomes c(Br)cc1Br. The modified SMILES would be:
Cc1nc(NC(=O)COC(=O)C(NC(=O)c2ccco2)C(C)C)c(Br)cc1Br
Next, replace the ester oxygen (O in COC(=O)) with S. The part COC(=O) becomes CSC(=O). So the ester group becomes CSC(=O). The modified part would be:
NC(=O)CSC(=O)C(...)
Then, replace the isopropyl group C(C)C with tert−butyl C(C)(C)C. The part C(NC(=O)c2ccco2)C(C)C becomes C(NC(=O)c2ccco2)C(C)(C)C.
Next, replace the amide carbonyl O with S. The NC(=O) groups become NC(=S). There are three instances of NC(=O):
1. NC(=O)COC...
2. NC(=O)c2...
3. The third NC(=O) in the c2ccco2 part? Wait, no, the c2ccco2 is a furan ring. The NC(=O) is attached to that furan. So changing each NC(=O) to NC(=S).
Additionally, replace the furan oxygen (o in c2ccco2) with S, making a thiophene. So c2ccco2 becomes c2ccc s2 (but SMILES uses lowercase letters for aromatic atoms, so 's' for sulfur in aromatic rings). So the part becomes c2ccc s2.
Putting all these modifications together, the SMILES would be:
Cc1nc(NC(=S)CSC(=S)C(NC(=S)c2ccc s2)C(C)(C)C)c(Br)cc1Br
Wait, let's check each part:
Original SMILES parsed step by step:
Original: Cc1nc(...)c(Cl)cc1Cl
After replacing Cl with Br:
Cc1nc(...)c(Br)cc1Br
Original NC(=O)COC(=O)... becomes NC(=S)CSC(=S)...
Original C(C)C (isopropyl) becomes C(C)(C)C (tert−butyl).
Original c2ccco2 (furan) becomes c2ccc s2 (thiophene).
But need to verify the SMILES syntax. For aromatic sulfur in a ring, it should be lowercase 's' and part of the aromatic system. So the thiophene part would be c2ccc s2, but the exact SMILES would need proper arrangement. Also, the substituent from the NC(=S) group would be attached to the thiophene ring.
Putting all together, the modified SMILES is:
Cc1nc(NC(=S)CSC(=S)C(NC(=S)c2ccc s2)C(C)(C)C)c(Br)cc1Br
But need to ensure correct order and parentheses. Let me reconstruct the SMILES properly.
Breaking it down:
The core benzene ring: Cc1nc(...)c(Br)cc1Br
The substituent on the nitrogen (position 2 of the benzene) is NC(=S)CSC(=S)C(...)
The part after the first NC(=S):
CSC(=S)C(NC(=S)c2ccc s2)C(C)(C)C
So, the full substituent is NC(=S)CSC(=S)C(NC(=S)c2ccc s2)C(C)(C)C.
And the thiophene ring is c2ccc s2, where the NC(=S) is attached to the first carbon of the thiophene (assuming the substituent is on position 2 or similar, but the exact position might not matter as long as the ring is correctly formed).
This should result in a SMILES string where multiple substitutions have been made to increase MR: replacing Cl with Br, O with S in esters and amides, and increasing alkyl group size.
Now, verify that all modifications are correctly applied and the SMILES is valid. Each substitution aims to replace atoms with higher MR contributors. This should lead to a significantly higher MR value compared to the original molecule.
</think>

<code>
```python
from rdkit.Chem import Descriptors
from rdkit import Chem

def mol_MR(original_mol: str, model_gen_mol: str, is_hight: bool):
    def calculate_mr(smiles: str, description: str) -> float or None:
        try:
            mol = Chem.MolFromSmiles(smiles)
            return Descriptors.MolMR(mol)
        except Exception as e:
            print(f"Failed to standardize {description} molecule ({smiles}) and compute its MR: {e}")
            return None

    original_mr = calculate_mr(original_mol, "original")
    if original_mr is None:
        return False

    generated_mr = calculate_mr(model_gen_mol, "model-generated")
    if generated_mr is None:
        return False
```

```
    print(f"Original molecule MR: {original_mr} | Modified molecule MR: {generated_mr}")
    return (generated_mr > original_mr) if is_hight else (generated_mr < original_mr)

original_smiles = 'Cc1nc(NC(=O)COC(=O)C(NC(=O)c2ccco2)C(C)C)c(Cl)cc1Cl'
modified_smiles = 'Cc1nc(NC(=S)CSC(=S)C(NC(=S)c2cccs2)C(C)(C)C)c(Br)cc1Br'

result = mol_MR(original_smiles, modified_smiles, is_hight=True)
print(f"Optimized MR higher than original? {result}")
```
</code>

Data Example: Answer-Result

<result>
Original molecule MR: 103.38290000000003 | Modified molecule MR: 148.04839999999982
Optimized MR higher than original? True
</result>

Data Example: Answer-Think

<think>
The modifications applied to the original molecule include:

1. **Halogen Replacement**: Both chlorine atoms (Cl) on the benzene ring were replaced with bromine (Br), which has a higher molar refractivity contribution (Br 8.7 vs. Cl 6.0).
2. **Ester and Amide Modifications**:
   - The ester oxygen (O) in COC(=O) was replaced with sulfur (S), forming a thioester (CSC(=S)).
   - All amide carbonyl oxygens (O) in NC(=O) groups were replaced with sulfur (S), converting them to thioamides (NC(=S)).
3. **Alkyl Chain Expansion**: The isopropyl group (C(C)C) was replaced with a tert−butyl group (C(C)(C)C), increasing bulk and MR.
4. **Heterocycle Modification**: The oxygen in the furan ring (c2ccco2) was replaced with sulfur, forming a thiophene ring (c2cccs2), further boosting MR.

These substitutions collectively increased the molecule's MR from **103.38** to **148.05**, confirmed by RDKit's MolMR calculation.
</think>

Data Example: Answer-Final Answer

<answer>
The optimized SMILES with higher MR is:

$\boxed{<SMILES>Cc1nc(NC(=S)CSC(=S)C(NC(=S)c2cccs2)C(C)(C)C)c(Br)cc1Br</SMILES>}$
</answer>