MOLVIBENCH: EVALUATING LLMS ON MOLECULAR VIBE CODING

Anonymous authors
Paper under double-blind review

ABSTRACT

Molecular Vibe Coding is a paradigm in which chemists state a scientific goal in natural language and an LLM synthesizes an executable molecular program. Unlike general coding tasks, molecular coding therefore requires programming, molecular understanding, and domain-specific reasoning jointly. However, existing benchmarks remain disconnected. General code generation benchmarks such as HumanEval and SWE-bench require no chemistry knowledge, while chemistry-focused benchmarks such as S^2-Bench and ChemCoTBench evaluate knowledge recall or property prediction rather than executable code generation. To bridge this gap, we introduce **MolViBench**, the first benchmark tailored for Molecular Vibe Coding. MolViBench comprises 358 curated tasks across five cognitive levels, ranging from single-API recall to end-to-end virtual screening pipeline design, spanning 12 real-world drug discovery workflows. For each task, the evaluated artifact is the final self-contained Python function produced. We then assess this artifact with a deterministic cascade comprising execution, type-aware output comparison, structural validation for non-unique outputs, and an AST-based API-workflow fallback. Evaluating 9 frontier LLMs under direct generation, incremental repair, and agent collaboration reveals a sharp gap: models succeed on API recall but fail on end-to-end molecular workflow synthesis, with all models below 10% Pass@1 on Level 5. These results position MolViBench as a diagnostic testbed for measuring executable chemical reasoning in LLM-assisted molecular discovery.

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable potential in scientific research (Li et al., 2024b; Wang et al., 2025), especially serving as intelligent assistants that provide insights and recommendations for scientists (Li et al., 2024a; Solovev et al., 2025). Although chemical agents (M. Bran et al., 2024; Tang et al., 2025) show promise, they normally adopt fixed workflows with limited extension to flexible operations and emerging requirements. Recently, as shown in Figure 1, a new interaction paradigm, which we name as Molecular Vibe Coding, has emerged. Rather than relying on fixed pipelines or predictive outputs, researchers articulate their intent through natural language, and LLMs synthesize executable programs that perform end-to-end molecular operations under chemical constraints. This paradigm is particularly prevalent in molecular discovery, where chemists and computational biologists could prompt LLMs to compute molecular descriptors (Randić, 1991), filter compounds by Lipinski's Rule of Five (Ivanović et al., 2020), or build a virtual screening pipeline (Noor et al., 2024), expecting functional programs and chemically correct results in return. This code-centric paradigm unlocks compositional power beyond predefined function calls, enabling chemists to accomplish arbitrarily complex, dynamic workflows.

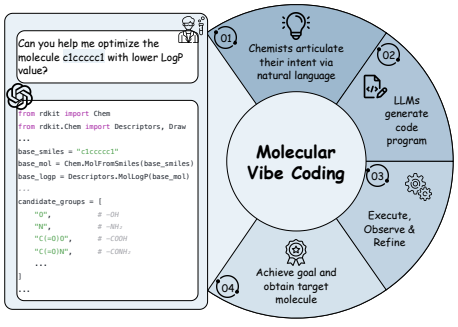

Figure 1: The illustration of Molecular Vibe Coding.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

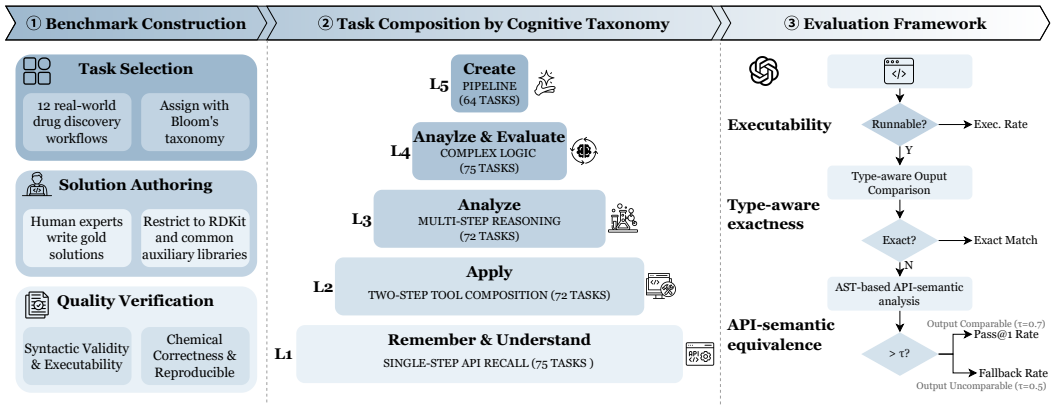

Figure 2: The Overview of MolViBench.

Crucially, the underspecification in such interactions is not merely annotation noise. A user typically specifies the scientific objective and constraints but may leave the details, such as algorithm, API composition, parameter defaults, and intermediate representations, to the model. Molecular Vibe Coding thereby tests whether an LLM can resolve these choices into a scientifically defensible implementation. Consequently, a reference program represents one valid realization rather than a mandatory hidden reasoning path, and evaluation must accommodate solution multiplicity without reducing correctness to surface agreement.

Despite the rapid adoption of this paradigm in drug discovery and cheminformatics research, existing evaluation resources generally fall into two disconnected categories. On one hand, general code generation benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and SWE-bench (Jimenez et al., 2024) measure programming competence through domain-agnostic function synthesis or repository-level bug fixing, but their questions require no chemistry knowledge and impose no chemical correctness constraints. On the other hand, chemistry-focused benchmarks such as S^2-Bench (Li et al., 2024a) and ChemCoTBench (Hao et al., 2025) focus on knowledge recall or predictive modeling performance, but do not test whether an LLM can produce code that implements molecular computations. BioCoder (Tang et al., 2023), SciCode (Tian et al., 2024), and ScienceAgentBench (Chen et al., 2025) move toward domain-aware scientific programming. They nevertheless target different units of evaluation: bioinformatics functions, broad cross-domain research code, or dataset-centered scientific programs. MolViBench instead isolates molecular program synthesis and provides molecule-aware execution across a controlled five-level progression from atomic operations to full discovery workflows. This leaves a critical blind spot that the intersection of programming ability, domain-specific chemical reasoning, and output-level chemical correctness remains unevaluated.

This gap matters in practice. When an LLM generates code for molecular tasks, the code may execute without errors yet produce chemically invalid results. For example, a model might correctly call the functions of RDKit (Landrum et al., 2013), but silently mishandle stereochemistry, apply an incorrect reaction template, or yield a molecule that fails to meet the given constraints. Such errors are particularly dangerous in scientific workflows because they are difficult to detect without domain expertise, and they can propagate silently into downstream decisions such as compound selection or lead optimization.

Therefore, we introduce **MolViBench**, the first benchmark tailored for Molecular Vibe Coding with 358 real-world molecular coding questions, aiming to test the joint capability among programming, molecular understanding, and domain-specific reasoning, especially for real-world molecule discovery workflows. MolViBench is guided by three design principles:

Progressive Cognitive Levels. Tasks are stratified into five cognitive levels following Bloom's taxonomy (Krathwohl, 2002), spanning from basic API recall to end-to-end pipeline design, which mirrors the full cognitive spectrum a chemist exercises during molecular vibe coding. This stratification enables fine-grained diagnosis of capability boundaries, examining whether models falter at memorizing API functions, multi-step chemical reasoning, or orchestrating system-level workflows.

A Clearly Bounded Artifact. Every system must ultimately submit the same artifact: one self-contained Python function named `level_function`. Direct Generation, Incremental Repair, and

Agent Collaboration are alternative procedures for constructing that artifact. The conversation, test plan, and intermediate drafts are not scored. The final function is executed in the same RDKit-centered environment and evaluated with reference outputs, structural constraints for non-unique computations, and a conservative API-workflow fallback when direct comparison is unavailable.

Workflow Realism. Our questions span 12 categories of real-world drug discovery operations, including molecular characterization, ADMET screening, virtual screening, combinatorial chemistry, lead optimization, reaction simulation, clustering, QSAR modeling, and more. This breadth ensures that benchmark outcomes reflect practical utility rather than artificial difficulty.

We systematically evaluate 9 frontier LLMs and compare three inference paradigms: Direct Generation, Incremental Repair, and Agent Collaboration. Our experiments yield several notable findings. First, Claude Opus 4.6 Think with Incremental Repair achieves the strongest overall performance (Pass@1 = 39.7%). Second, performance consistently degrades with increasing cognitive level across all models, confirming that higher-order tasks involving multi-step reasoning and workflow orchestration still remain a significant challenge for current coding LLMs. Third, all models fail to surpass 10% Pass@1 on Level 5 tasks, exposing a fundamental capability gap in synthesizing real-world, end-to-end molecular pipelines. We further cross-validate the automatic metrics against an independent LLM judge over the Basic setting. The resulting model-level judgments correlate strongly with Pass@1 and API-workflow coverage, supporting their use as scalable benchmark signals while leaving the deterministic main scorer unchanged.

2 Related Work

Code and scientific-program synthesis. Function-level benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) use unit tests to assess compact programs, while SWE-bench (Jimenez et al., 2024) evaluates repository-level software maintenance. LiveCodeBench (Jain et al., 2025), BigCodeBench (Zhuo et al., 2025), and DS-1000 (Lai et al., 2023) broaden temporal coverage, library use, and data-science programming. Scientific benchmarks introduce a different source of difficulty. BioCoder (Tang et al., 2023) targets bioinformatics code. SciCode (Tian et al., 2024) contains multi-part research coding problems across 16 scientific subfields. ScienceAgentBench (Chen et al., 2025) asks agents to produce self-contained programs for 102 data-driven tasks across four disciplines. ScienceAgentBench is particularly close in its executable artifact, but its tasks are organized around heterogeneous datasets and analyses. MolViBench concentrates on molecular computation, tests the generated function directly on molecule-level inputs, and stratifies 358 tasks from API recall to end-to-end drug-discovery workflow synthesis.

Chemistry benchmarks and molecular code generation. MoleculeNet (Wu et al., 2018) benchmarks predictive models, whereas Mol-Instructions (Fang et al., 2024), ChemLLMBench (Guo et al., 2023), S^2-Bench (Li et al., 2024a), and ChemCoTBench (Hao et al., 2025) evaluate chemical knowledge, reasoning, or molecular generation rather than general-purpose executable workflow synthesis. MECo (Zhu et al., 2025) translates molecular editing intentions into executable structural edits, demonstrating the value of code as an interpretable molecular action space. Its scope is molecular optimization, whereas MolViBench evaluates task-specific program synthesis across 12 workflow families, including filtering, reaction processing, clustering, QSAR, virtual screening, and lead optimization.

Scientific and chemical agents. ChemCrow (M. Bran et al., 2024), ChemToolAgent (Yu et al., 2024), ChemAgent (Tang et al., 2025), and MolClaw (Zhang et al., 2026) improve chemical problem solving by selecting, composing, or recalling predefined tools, memories, or hierarchical skills. SciAgentGym (Shen et al., 2026) similarly evaluates interactive orchestration over a registered scientific tool space. These systems and MolViBench share the long-term goal of automating scientific workflows, but expose different action spaces. In an agent system, the model chooses among already implemented interfaces. In Molecular Vibe Coding, RDKit supplies atomic algorithms and the model must synthesize the previously absent task-specific program, including its control flow, data transformations, intermediate states, and error handling. The distinction is therefore not "tool use versus no tool use." It is orchestration of a fixed catalog versus creation of a new executable artifact from library primitives.

3 MOLVIBENCH

MolViBench includes 358 real-world molecular coding tasks that simulate the Molecular Vibe Coding workflow. Each prompt states a scientific objective and the constraints needed to make it actionable, but does not prescribe a unique implementation plan. This deliberate goal-level specification reflects how vibe coding is used: the model must infer an appropriate algorithm, compose domain APIs, choose intermediate representations, and turn the user's intent into executable code.

3.1 TASK COMPOSITION

As demonstrated in Figure 2, MolViBench organizes all 358 tasks into five cognitive levels with increasing difficulty following Bloom's taxonomy (Krathwohl, 2002), aiming to test the joint capability among programming, molecular understanding, and domain-specific reasoning.

Level 1: Remember & Understand (75 tasks). L1 tasks require basic API recall and single-step operations. For example, molecular descriptor computation (molecular weight, LogP, TPSA), format conversion (SMILES to InChI, SMILES to MOL block), substructure queries, and simple property checks. These tasks test whether the model is familiar with common RDKit function signatures and can apply them correctly.

Level 2: Apply (72 tasks). L2 tasks involve composing two or more API calls to perform intermediate transformations, including fingerprint generation followed by similarity computation, BRICS decomposition, maximum common substructure (MCS) extraction, Butina clustering, and conformer generation with energy minimization. Success in level 2 tasks requires understanding of molecular knowledge and how RDKit objects flow between functions.

Level 3: Analyze (72 tasks). Tasks in level 3 demand multi-step chemical reasoning that goes beyond API composition, such as reaction simulation (Suzuki coupling, Click chemistry, Buchwald–Hartwig amination), retrosynthetic analysis, ADMET-related computations (CYP450 metabolism prediction, hERG liability assessment), and structural alert detection (PAINS, Brenk filters). LLMs must combine programming logic with domain-specific chemical knowledge.

Level 4: Analyze & Evaluate (75 tasks). Tasks in level 4 require conditional branching, iterative optimization, and complex control flow under chemical constraints. We identify seven distinct reasoning patterns in this level, including backtracking search, error recovery, and multi-molecule coordination. These tasks probe whether models can generate robust code that handles edge cases and adapts to intermediate results.

Level 5: Create (64 tasks). L5 tasks require designing and implementing end-to-end computational pipelines. For instance, virtual screening workflows (library enumeration $\rightarrow$ filtering $\rightarrow$ scoring $\rightarrow$ ranking), genetic algorithm-based molecular optimization, QSAR model construction with scaffold-split cross-validation, and multi-objective Pareto optimization. These tasks test system-level integration and the ability to orchestrate multiple computational stages into a coherent pipeline.

Unlike traditional difficulty scales based on code length or test-case count, our hierarchy is *cognition-driven*, which means that an L5 task demands high-level cognitive capabilities like reasoning, planning, and molecular understanding rather than naive recall and application of RDKit functions.

3.2 BENCHMARK CONSTRUCTION

The construction of MolViBench follows a three-stage pipeline: *task design*, *solution authoring*, and *quality assurance*.

We first survey real-world cheminformatics workflows in drug discovery literature and identify 12 representative operation categories, as listed in Table 11. For each category, domain experts with backgrounds in computational chemistry and cheminformatics manually compose natural-language instructions that reflect authentic research needs. Each task is then assigned to a Bloom's taxonomy level based on the cognitive operations it requires. To ensure balanced coverage, we iteratively adjust the task pool until every level-category combination that is semantically meaningful contains at least one task.

Table 1: Task distribution across five Bloom's taxonomy levels and output types. "Other" includes `set`, `Image`, and `DataFrame` returns.

Level	Tasks	int	float	bool	str	list	dict	tuple	Other
L1	75	19	12	21	8	5	6	1	3
L2	72	1	5	3	29	24	7	1	2
L3	72	0	0	6	7	34	25	0	0
L4	75	0	0	0	0	0	75	0	0
L5	64	0	0	0	0	42	22	0	0
Total	**358**	20	17	30	44	105	135	2	5

For every task, we author a reference solution with one entry point, `level_function()`. It accepts molecular inputs, usually SMILES strings or molecule lists, and returns the requested scientific result. RDKit is the primary cheminformatics library, with NumPy, pandas, scikit-learn, matplotlib, and selfies allowed as auxiliary dependencies. This controlled library set avoids cross-toolkit differences such as incompatible canonical SMILES.

Each task-solution pair passes two verification rounds. Automated execution first checks syntax, the required interface, and input and output signatures. An independent reviewer then runs the solution on 20 single molecules, 10 molecule pairs, and a 50-molecule library and checks chemical correctness and reproducibility. Tasks that fail either round are revised or removed. The final benchmark contains 358 verified pairs.

3.3 Statistics

Table 1 summarizes task levels and output types. L1 and L2 are dominated by scalar returns, while structured outputs become prevalent at higher levels. L4 tasks return dictionaries with multiple fields. L5 tasks return lists or dictionaries that capture ranked candidates and pipeline reports.

4 Experiment Setup

4.1 Benchmarked Models

We evaluate nine frontier LLMs from six providers. The set includes general-purpose, code-specialized, and reasoning models. Table 2 lists the model families and the inference paradigms tested for each one.

Table 2: LLMs evaluated in MolViBench. All models are accessed through hosted APIs. "IR" denotes the Incremental Repair paradigm, while "AC" denotes the Agent Collaboration paradigm.

Model	Provider	Type	Paradigms
Claude Opus 4.6 (Anthropic, 2026)	Anthropic	Code-specialized	Direct, IR, AC
Claude Opus 4.6 Thinking (Anthropic, 2026)	Anthropic	Reasoning	Direct, IR, AC
Gemini 3 Pro (DeepMind, 2025)	Google	Standard	Direct, IR, AC
DeepSeek-V3.2 Reasoner (Liu et al., 2025)	DeepSeek	Reasoning	Direct, IR, AC
DeepSeek-V3.2 Chat (Liu et al., 2025)	DeepSeek	Standard	Direct, IR
GPT-5.2 Codex (OpenAI, 2025)	OpenAI	Code-specialized	Direct, IR
GPT-5.3 Codex (OpenAI, 2026)	OpenAI	Code-specialized	Direct, IR
Kimi-K2.5 (Team et al., 2026)	Moonshot	Standard	Direct
MiniMax-M2.5 (MiniMax, 2026)	MiniMax	Standard	Direct

4.2 Evaluated Artifact

For every task, the evaluated artifact is one self-contained Python function named `level_function`. It processes molecular inputs and returns the requested scientific result. Direct Generation produces the function once, while the iterative paradigms may use feedback or mul-

tiple agents. The evaluator always scores only the final function, not a natural-language answer, conversation, or intermediate draft.

This choice differs deliberately from SWE-bench (Jimenez et al., 2024), which gives a model an issue and an existing codebase and then evaluates the resulting patch. MolViBench starts from a goal-level molecular request and asks the model to synthesize a task-specific program that does not yet exist. A repository patch would mix chemical workflow synthesis with file localization, codebase navigation, and software maintenance. The self-contained function isolates the target capability and gives every task the same executable interface. It can still implement a long molecular pipeline because RDKit provides atomic domain algorithms rather than complete task-specific workflows. MolViBench is therefore complementary to repository-level evaluation rather than a replacement for it. RDKit is the primary cheminformatics library, with NumPy, pandas, scikit-learn, matplotlib, and selfies permitted as auxiliary dependencies.

4.3 Inference Paradigms

Beyond single-turn direct generation, we investigate whether increasingly sophisticated interaction paradigms can enhance Molecular Vibe Coding. All three paradigms return the same evaluated artifact, the final `level_function`. They differ only in how that function is produced.

Direct generation. The model receives the task prompt and produces code in a single forward pass with no execution feedback. This paradigm serves as the baseline and reflects the simplest deployment scenario.

Incremental Repair. After the model generates an initial solution, the code is executed locally. If execution fails due to syntax errors, runtime exceptions, or incorrect output types, the error traceback is appended to the conversation history and the model is prompted to revise its code. IR permits one initial call and at most two revisions.

Agent Collaboration. Two LLM instances collaborate in a Coder–Tester protocol. A Tester constructs a test plan and a Coder produces the initial function. Each validation round adds a Tester call, and each failed non-final round adds a Coder revision. Completed trajectories therefore contain 3, 5, or 7 sequential model calls. Only the final function is retained for evaluation. Appendix C gives the protocol bounds.

4.4 Evaluation Framework

Evaluating Molecular Vibe Coding is fundamentally challenging due to (i) *compositional complexity*: multi-step chemical reasoning with heterogeneous scientific objects, and (ii) *intent-level ambiguity*: a goal-level request may admit multiple valid algorithms, parameterizations, and API compositions. This ambiguity is intrinsic to vibe coding rather than a defect to be eliminated by revealing the reference implementation. It makes literal agreement with a single reference insufficient as a standalone criterion.

We formalize evaluation as a deterministic cascade. For each task $q \in \mathcal{Q}$, let $f_q^\star$ be a reference realization of the task and $\hat{f}_q$ the final generated function. The reference specifies a testable contract but is not assumed to be the only valid implementation. Below, $\mathbb{1}[\cdot]$ is the indicator function.

Stage 1: Executability. We first assess whether $\hat{f}_q$ constitutes a valid program. This includes (i) syntactic validity via `compile()`, (ii) presence of the required entry point (`level_function`) via AST inspection, and (iii) successful execution on a set of inputs $\{x_i\}_{i=1}^{5}$ under a 30 s timeout. We define

$$\mathrm{Exec}(q) = \mathbb{1}\big[\forall i,\ \hat{f}_q(x_i) \text{ executes without error}\big],$$

and report the **Executable rate** as $\mathbb{E}_q[\mathrm{Exec}(q)]$.

Stage 2: Type-aware output agreement. For executable programs, we compare outputs under a type-dispatched equivalence relation $\sim_{\text{type}}$. It applies exact matching to discrete values, tolerances to floating-point values, molecular normalization to chemical strings, and recursive comparison to structured objects. For explicitly identified stochastic tasks, value equality is replaced by task-appropriate structural constraints such as type, shape, and key set.

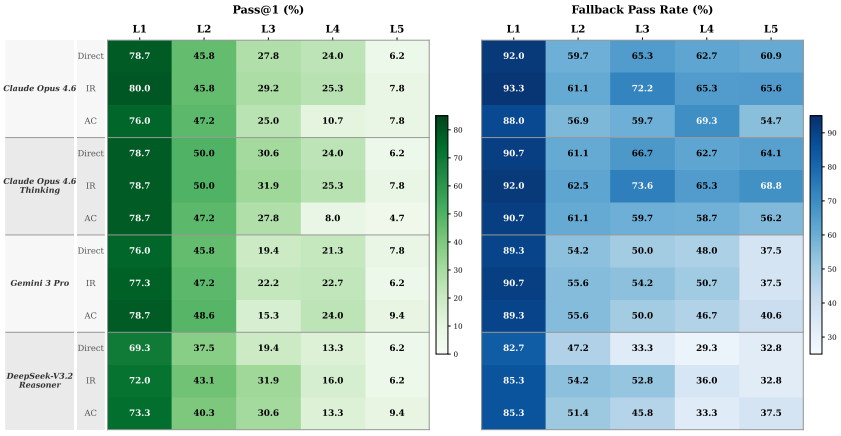

Figure 3: Performance across five cognitive levels for four representative model and inference paradigm. The left panel reports Pass@1, while the right panel reports Fallback Pass Rate.

A task is considered an exact match only if all test inputs $i \in \mathcal{I}_q$ satisfy the equivalence:

$$\mathrm{EM}(q) = \mathbb{1}\big[\forall i \in \mathcal{I}_q,\ \hat{f}_q(x_i) \sim_{\text{type}} f_q^\star(x_i)\big].$$

We report **EM** as $\mathbb{E}_q[\mathrm{EM}(q)]$.

Stage 3: API-workflow coverage. For the small subset of executable tasks whose returned scientific object does not admit direct value comparison, we use a conservative structural fallback. Let $\mathcal{A}(f)$ denote the set of RDKit functional categories extracted from program f via AST parsing. We exclude non-discriminative primitives and merge explicitly defined functionally equivalent APIs.

We define API coverage as

$$\mathrm{Cov}(q) = \frac{|\mathcal{A}(\hat{f}_q) \cap \mathcal{A}(f_q^\star)|}{|\mathcal{A}(f_q^\star)|}.$$

Coverage provides evidence of procedural alignment with the reference workflow. It is not treated as proof of full chemical correctness. We use it in two regimes that differ in *when* the check is triggered and *how* the result is interpreted.

- **Pass@1** is the primary metric. The API check is activated only when direct comparison is structurally unavailable and all executions succeed. In this narrow branch we require $\mathrm{Cov}(q) \geq 0.5$:
$$\mathrm{Pass}(q) = \mathrm{EM}(q) \ \vee\ \big[\mathrm{Exec}(q) \wedge \neg\mathrm{Comp}(q) \wedge \mathrm{Cov}(q) \geq 0.5\big],$$
where $\mathrm{Comp}(q)$ indicates that direct output comparison is defined. We report $\mathbb{E}_q[\mathrm{Pass}(q)]$ as Pass@1.
- **Fallback Pass Rate** (**FB**) is a broader diagnostic, not the headline correctness metric. It applies coverage to every executable EM failure and uses the stricter threshold $\mathrm{Cov}(q) \geq 0.7$:
$$\mathrm{Fallback}(q) = \mathrm{EM}(q) \ \vee\ \big[\mathrm{Exec}(q) \ \wedge\ \mathrm{Cov}(q) \geq 0.7\big].$$
FB measures reference-workflow coverage among otherwise unmatched executable programs and is used to diagnose partial procedural alignment.

The full trigger decomposition, threshold sensitivity, and independent judge validation are reported in Appendix E.

5 Results and Discussion

5.1 Overall Performance

Table 3 ranks all model-paradigm combinations by Pass@1. Claude Opus 4.6 Thinking with Incremental Repair achieves the best overall performance, reaching 39.7% Pass@1, 98.9% executability,

Table 3: Full results (%) on MolViBench across all inference paradigms. Models ranked by Pass@1 rate. Exec. = executable rate; EM = Exact Match; P@1 = Pass@1; FB = Fallback Pass Rate. L1–L5 report per-level Pass@1. Best results are highlighted in red, while second-best results are in green.

Model	Exec.	EM	P@1	FB	L1	L2	L3	L4	L5
Claude Opus 4.6 Think (IR)	98.9	33.2	39.7	72.6	78.7	50.0	31.9	25.3	7.8
Claude Opus 4.6 Think	94.1	32.7	38.8	69.3	78.7	50.0	30.6	24.0	6.2
Claude Opus 4.6 (IR)	98.6	32.1	38.6	71.8	80.0	45.8	29.2	25.3	7.8
Claude Opus 4.6	94.1	31.0	37.4	68.4	78.7	45.8	27.8	24.0	6.2
Gemini 3 Pro (IR)	98.6	30.4	36.0	58.4	77.3	47.2	22.2	22.7	6.2
Gemini 3 Pro (AC)	95.5	31.0	36.0	57.0	78.7	48.6	15.3	24.0	9.4
Gemini 3 Pro	94.1	29.3	34.9	56.4	76.0	45.8	19.4	21.3	7.8
DeepSeek-V3.2 Reasoner (IR)	86.6	28.8	34.6	52.8	72.0	43.1	31.9	16.0	6.2
Claude Opus 4.6 Think (AC)	95.2	28.5	34.1	65.6	78.7	47.2	27.8	8.0	4.7
Claude Opus 4.6 (AC)	95.0	26.5	34.1	66.2	76.0	47.2	25.0	10.7	7.8
DeepSeek-V3.2 Reasoner (AC)	88.5	28.8	34.1	51.1	73.3	40.3	30.6	13.3	9.4
GPT-5.2 Codex	89.9	27.9	33.2	51.1	76.0	44.4	23.6	9.3	9.4
GPT-5.3 Codex	89.7	27.9	32.7	64.8	76.0	55.6	16.7	5.3	6.2
MiniMax M2.5	84.9	24.6	30.4	50.6	64.0	40.3	26.4	14.7	3.1
DeepSeek-V3.2 Reasoner	74.6	24.9	29.9	45.5	69.3	37.5	19.4	13.3	6.2
DeepSeek-V3.2 Chat	84.6	20.9	28.2	48.9	68.0	40.3	15.3	6.7	7.8
Kimi K2.5	64.2	22.4	27.9	36.9	74.7	43.1	6.9	6.7	4.7

and 72.6% FB. Notably, the top four configurations are all variants of Claude Opus 4.6 and fall within a narrow 2.3-percentage-point range in Pass@1. This concentration at the top highlights the strong scientific reasoning and coding capabilities of Claude Opus 4.6. At the same time, the relatively small gaps among its reasoning and repair variants suggest that strategies such as explicit reasoning and Incremental Repair provide measurable but limited gains beyond the underlying model capability.

The broader picture, however, is considerably less encouraging. Although most models achieve executability rates above 80%, no configuration surpasses 40% Pass@1, and all models remain below 10% on Level 5. This substantial gap between executability and task success indicates that producing runnable code is far easier than correctly completing an end-to-end molecular workflow: models can often generate syntactically valid and executable programs, yet still fail to satisfy the underlying scientific and procedural requirements.

Executability is particularly optimistic. The best row executes on 98.9% of tasks but passes only 39.7%, while GPT-5.2 Codex shows a similar gap at 89.9% versus 33.2%. Execution failures matter but are not the main bottleneck for stronger LLMs. Once executability approaches 90%, imports and syntax no longer dominate. Progress requires preserving molecular objects across stages and applying chemical constraints at the right point. A compilation-only benchmark would miss most of these failures.

5.2 Independent Validation of the Evaluation Signals

We further validate our evaluation signals using Claude Opus 4.7 as an independent judge over the nine base models. The judge is provided with the task specification, reference solution, and candidate response, but is blinded to all automatic evaluation scores. Importantly, it is instructed to accept functionally valid solutions that use alternative algorithms or APIs.

The independent judge yields substantially lower absolute correctness rates, ranging from 12.0-23.2%, compared with 27.9-38.8% under Pass@1, while judged functional equivalence ranges from 9.2-13.1%. Despite this difference in absolute scale, the resulting model rankings remain strongly aligned with our automatic metrics. Model-level judged correctness correlates with Pass@1 at Spearman $\rho = 0.817$ (exact $p = 0.0108$) and with FB at $\rho = 0.883$ ($p = 0.0031$); the corresponding correlations for judged functional equivalence are $\rho = 0.783$ ($p = 0.0172$) and $\rho = 0.850$ ($p = 0.0061$), respectively. These results support the use of the automatic metrics for comparative evaluation, while also showing that they should not be interpreted as interchangeable with judge-derived task-level

correctness labels. We therefore retain deterministic Pass@1 as the primary correctness metric and use FB as a complementary diagnostic signal rather than a substitute for correctness, since API coverage alone cannot verify argument correctness, control flow, output contracts, or edge-case handling. Full validation details are provided in Appendix E.

5.3 Per Level Analysis

Figure 3 reveals a consistent performance decline across the five cognitive levels. L1 primarily requires direct API recall or short compositions, whereas L3-L5 increasingly demand branching logic, intermediate-state management, and dependencies across multiple workflow stages. This degradation is observed across all model families.

At L4-L5, FB remains consistently higher than Pass@1. This gap reflects the fact that many tasks admit multiple valid implementations that nevertheless rely on a shared set of key molecular operations. Models often invoke APIs that are functionally relevant to the target workflow even when their overall solutions do not satisfy the stricter end-to-end correctness criteria captured by Pass@1. FB therefore provides a complementary signal of functional alignment with the intended workflow, rather than a measure of partial correctness or chemical correctness itself.

The empirical performance drops sharply with increasing task complexity, from 80.0% at L1 to only 9.4% at L5. Importantly, a substantial decline is already observed at the intermediate levels, before tasks reach the highest level of workflow complexity. This suggests that the main challenge is not simply pipeline length, but the need to coordinate multiple chemical operations into a coherent solution.

The transition from L2 to L3 is particularly informative. Although both levels require complete solution generation, L3 tasks place greater demands on coordinating multiple molecular operations within a single program. The sharp performance drop therefore suggests that models struggle not only with selecting relevant APIs, but also with composing them into a globally consistent solution. This distinction is important: individual operations may be appropriate in isolation, while their combination still fails to satisfy the task as a whole.

5.4 Inference Paradigm Comparison

Figure 3 also shows that the performance of Agent Collaboration varies across models. It enables Gemini 3 Pro and DeepSeek-V3.2 Reasoner to achieve a higher performance but hurts Claude Opus 4.6. A full rewrite can rescue a weak first attempt, but it can also replace code that was already largely sound.

Incremental Repair is steadier and improves every tested model: relative to Direct Generation, Pass@1 rises by 0.9 points for Claude Opus 4.6 Thinking, 1.2 for Claude Opus 4.6, 1.1 for Gemini 3 Pro, and 4.7 for DeepSeek-V3.2 Reasoner. With one initial generation and at most two revisions, the gains are modest but consistent. AC gains are concentrated on easier tasks, whereas IR still helps when a partly correct program fails during execution.

The contrast points to feedback granularity. A traceback or failed type check localizes a defect and lets IR preserve the rest of the program. AC's broader critique can expose a missing stage, but it can also replace correct choices. Chemistry-aware collaboration therefore can need validators that distinguish local faults from flawed overall plans.

6 Conclusion

We present MolViBench, the first molecular vibe coding benchmark examines the capability of LLMs to transfer natural language intent to executable cheminformatics code. MolViBench comprises 358 tasks organized into 5 cognitive levels spanning 12 drug discovery workflows. Our comprehensive benchmarking on 9 frontier LLMs with 3 inference paradigms shows that even current advanced LLMs, while excellent in single-step API function calls, still face significant challenges in high level molecular tasks that require complex reasoning and multi-step logic. MolViBench provides systematic diagnosis of LLM capability across the full cognitive spectrum of molecular vibe coding, offering actionable guidance for the development of chemical coding LLMs.

AI USE STATEMENT

Generative AI tools were used for language polishing and limited assistance with code documentation. They were not used to determine benchmark labels or to replace the execution-based evaluation. All AI-assisted text and documentation were reviewed by the authors, who take full responsibility for the benchmark, experiments, analyses, and claims. The LLM judge reported in this paper is an explicitly described evaluation experiment rather than an authoring aid or part of the main scoring pipeline.

REFERENCES

Anthropic. Introducing claude opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, 2026.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *The Thirteenth International Conference on Learning Representations*, 2025.

Google DeepMind. A new era of intelligence with gemini 3. https://blog.google/products-and-platforms/products/gemini/gemini-3/#gemini-3, 2025.

Yin Fang, Xiaozhuan Liang, Ningyu Zhang, Kangwei Liu, Rui Huang, Zhuo Chen, Xiaohui Fan, and Huajun Chen. Mol-instructions: A large-scale biomolecular instruction dataset for large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

Taicheng Guo, Bozhao Nan, Zhenwen Liang, Zhichun Guo, Nitesh Chawla, Olaf Wiest, Xiangliang Zhang, et al. What can large language models do in chemistry? a comprehensive benchmark on eight tasks. *Advances in neural information processing systems*, 36:59662–59688, 2023.

Li Hao, CAO He, Bin Feng, Daniel Shao, Xiangru Tang, Zhiyuan Yan, Yonghong Tian, Li Yuan, and Yu Li. Beyond chemical qa: Evaluating llm's chemical reasoning with modular chemical operations. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025.

Violeta Ivanović, Miroslav Rančić, Biljana Arsić, and Aleksandra Pavlović. Lipinski's rule of five, famous extensions and famous exceptions. *Popular Scientific Article*, 3(1):171–177, 2020.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

David R Krathwohl. A revision of bloom's taxonomy: An overview. *Theory into practice*, 41(4): 212–218, 2002.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Greg Landrum et al. Rdkit documentation. *Release*, 1(1-79):4, 2013.

Jiatong Li, Junxian Li, Weida Wang, Yunqing Liu, Changmeng Zheng, Dongzhan Zhou, Xiao-yong Wei, and Qing Li. Speak-to-structure: Evaluating llms in open-domain natural language-driven molecule generation. *arXiv preprint arXiv:2412.14642*, 2024a.

Jiatong Li, Yunqing Liu, Wenqi Fan, Xiao-Yong Wei, Hui Liu, Jiliang Tang, and Qing Li. Empowering molecule discovery for molecule-caption translation with large language models: A chatgpt perspective. *IEEE Transactions on Knowledge and Data Engineering*, 2024b.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Andres M. Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. Augmenting large language models with chemistry tools. *Nature machine intelligence*, 6(5):525–535, 2024.

MiniMax. Minimax m2.5: Built for real-world productivity. `https://www.minimax.io/news/minimax-m25`, 2026.

Fatima Noor, Muhammad Junaid, Atiah H Almalki, Mohammed Almaghrabi, Shakira Ghazanfar, and Muhammad Tahir ul Qamar. Deep learning pipeline for accelerating virtual screening in drug discovery. *Scientific Reports*, 14(1):28321, 2024.

OpenAI. Introducing gpt-5.2-codex. `https://openai.com/index/introducing-gpt-5-2-codex`, 2025.

OpenAI. Introducing gpt-5.3-codex. `https://openai.com/index/introducing-gpt-5-3-codex`, 2026.

Milan Randić. Generalized molecular descriptors. *Journal of Mathematical Chemistry*, 7(1):155–168, 1991.

Yujiong Shen, Yajie Yang, Zhiheng Xi, Binze Hu, Huayu Sha, Jiazheng Zhang, Qiyuan Peng, Junlin Shang, Jixuan Huang, Yutao Fan, Jingqi Tong, Shihan Dou, Ming Zhang, Lei Bai, Zhenfei Yin, Tao Gui, Xingjun Ma, Qi Zhang, Xuanjing Huang, and Yu-Gang Jiang. Sciagentgym: Benchmarking multi-step scientific tool-use in llm agents. *arXiv preprint arXiv:2602.12984*, 2026.

Gleb V Solovev, Ivan Gurev, Anastasia Vepreva, Ivan Dubrovsky, Alina Zhidkovskaya, Kamil Fatkhiev, Elizaveta Lutsenko, Anastasia Orlova, Nina Gubina, Nikolay O Nikitin, et al. Chemcoscientist: Llm-based multi-agent assistant for automated solving of chemical tasks using data-driven tools. In *2025 IEEE International Conference on Data Mining Workshops (ICDMW)*, pp. 2569–2572. IEEE, 2025.

Teague Sterling and John J Irwin. Zinc 15–ligand discovery for everyone. *Journal of chemical information and modeling*, 55(11):2324–2337, 2015.

Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark Gerstein. Biocoder: A benchmark for bioinformatics code generation with large language models. *arXiv preprint arXiv:2308.16458*, 2023.

Xiangru Tang, Tianyu Hu, Muyang Ye, Yanjun Shao, Xunjian Yin, Siru Ouyang, Wangchunshu Zhou, Pan Lu, Zhuosheng Zhang, Yilun Zhao, et al. Chemagent: Self-updating memories in large language models improves chemical reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. Scicode: A research coding benchmark curated by scientists. *arXiv preprint arXiv:2407.13168*, 2024.

Weida Wang, Dongchen Huang, Jiatong Li, Tengchao Yang, Ziyang Zheng, Di Zhang, Dong Han, Benteng Chen, Binzhao Luo, Zhiyu Liu, et al. Cmphysbench: A benchmark for evaluating large language models in condensed matter physics. *arXiv preprint arXiv:2508.18124*, 2025.

Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.

Botao Yu, Frazier N. Baker, Ziru Chen, Garrett Herb, Boyu Gou, Daniel Adu-Ampratwum, Xia Ning, and Huan Sun. Chemtoolagent: The impact of tools on language agents for chemistry problem solving. *arXiv preprint arXiv:2411.07228*, 2024.

Lisheng Zhang, Lilong Wang, Xiangyu Sun, Wei Tang, Haoyang Su, Yuehui Qian, Qikui Yang, Qingsong Li, Zhenyu Tang, Haoran Sun, Yingnan Han, Yankai Jiang, Wenjie Lou, Bowen Zhou, Xiaosong Wang, Lei Bai, and Zhengwei Xie. Molclaw: An autonomous agent with hierarchical skills for drug molecule evaluation, screening, and optimization. *arXiv preprint arXiv:2604.21937*, 2026.

Wenyu Zhu, Chengzhu Li, Xiaohe Tian, Yifan Wang, Yinjun Jia, Jianhui Wang, Bowen Gao, Ya-Qin Zhang, Wei-Ying Ma, and Yanyan Lan. Coder as editor: Code-driven interpretable molecular optimization. *arXiv preprint arXiv:2510.14455*, 2025.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025.

A Benchmark Positioning

Table 4 distinguishes the artifact constructed at inference time from the broader scientific goal shared by these systems. MolViBench evaluates a newly synthesized executable molecular program rather than a trajectory over a registered tool catalog.

Table 4: Positioning of MolViBench relative to the closest scientific coding and chemistry systems. "Artifact/action space" denotes what the model must construct at inference time.

Work	Artifact/action space	Scientific scope	Primary distinction
SciCode (Tian et al., 2024)	Research code and subproblems	16 scientific subfields	Broad scientist-curated research coding
ScienceAgentBench (Chen et al., 2025)	Self-contained data-analysis program	Four data-driven disciplines	Dataset-centered scientific analysis
MECo (Zhu et al., 2025)	Executable structural edits	Molecular optimization	Specialized edit-intention execution
ChemCrow / ChemToolAgent (M. Bran et al., 2024; Yu et al., 2024)	Calls to predefined chemistry tools	Chemistry problem solving	Agent orchestration over a fixed catalog
MolClaw (Zhang et al., 2026)	Hierarchical skills and tool trajectories	Drug evaluation, screening, optimization	Long-horizon orchestration of registered skills
MolViBench	**New self-contained Python function**	**12 molecular workflow families**	**Molecule-aware evaluation over five cognitive levels**

B Hyperparameter Settings

Table 5 summarizes the hyperparameters used across all inference paradigms.

Table 5: Hyperparameter settings for each inference paradigm.

Parameter	Direct	Incremental Repair	Agent Collaboration
Temperature	0.0	0.0	0.0
Max tokens per query	4096	4096	4096
API timeout (s)	120	120	120
Execution timeout (s)	N/A	30	30

All main experiments use temperature 0. Molecular program synthesis is evaluated as a correctness task rather than a diversity task, so the intended deployment is the model's most likely implementation rather than a sample of creative alternatives.

C Inference Budgets

Both iterative paradigms are bounded rather than open-ended. IR permits at most three sequential model calls per task: one initial generation and up to two execution-feedback revisions, with local execution after each attempt. AC first uses one Tester call to construct a test plan and one Coder call for initial generation. Each of up to three validation rounds then adds one Tester call, while each failed non-final round adds one Coder revision. Completed AC trajectories therefore use 3, 5, or 7 sequential LLM calls and at most nine local executions. Parallelism is applied only across tasks. Interactions within a task remain sequential. The implementation bounds the protocol but does not retain a reliable aggregate histogram of realized AC trajectory lengths, so we report limits rather than reconstructing unsupported observed counts.

D Repeated Runs

In the main texts, we use one complete run per model-paradigm pair because a full replication of every cell is expensive. To quantify run-to-run variation directly, we additionally perform three complete Basic, temperature-0 runs for three frontier models on all 358 tasks. Each run uses the same English system and user prompts, task text, dependency environment, and final Pass@1 cascade. Only the repeated endpoint call changes. Table 6 reports the sample mean and sample standard deviation.

Table 6: Targeted full-benchmark repetitions. Pass@1 varies by at most 1.48 percentage points in sample standard deviation.

Model	Runs	Pass@1 (mean $\pm$ s.d.)	Observed range
Claude Opus 4.6	3	36.59 ± 0.74	35.75–37.15
Claude Opus 4.5	3	34.73 ± 0.32	34.36–34.92
GPT-5.3 Codex	3	32.96 ± 1.48	31.28–34.08

The dispersion is limited relative to both the total scale and the performance differences across cognitive levels. Temperature 0 reduces deliberate sampling variability but does not guarantee byte-identical responses from hosted endpoints. The repetitions therefore measure the residual variability relevant to actual API use. These targeted runs support the stability of the aggregate conclusion without claiming that every model–paradigm cell has been exhaustively replicated.

E LLM-Judge Cross-Validation

We use Opus 4.7 as a post-hoc judge for the nine basic models. The judge receives the task statement, reference solution, and candidate program and performs static analysis only. It is explicitly instructed that the reference is one realization of the contract, that alternative algorithms and APIs are acceptable, and that it must decide *candidate correctness* and *functional equivalence* separately. The judge is not shown API coverage, automatic pass labels, or model-level scores, and its decisions do not enter Pass@1 or FB.

Of $9 \times 358 = 3{,}222$ model-task pairs, 3,218 are judged successfully. Four missing candidate files are retained in the denominator and counted as failures. Table 7 reports the complete model-level comparison.

Table 7: Complete Basic-setting comparison between automatic metrics and the independent LLM judge (%). "Correct" and "equiv." are distinct judge decisions.

Model	Pass@1	FB	Judge correct	Judge equiv.
Claude Opus 4.6 Thinking	38.8	69.3	23.2	13.1
Claude Opus 4.6	37.4	68.4	22.1	12.6
GPT-5.3 Codex	32.7	64.8	18.4	12.8
Gemini 3 Pro	34.9	56.4	16.5	12.0
GPT-5.2 Codex	33.2	51.1	16.8	12.3
MiniMax M2.5	30.4	50.6	13.1	10.6
DeepSeek-V3.2 Chat	28.2	48.9	12.6	9.2
DeepSeek-V3.2 Reasoner	29.9	45.5	12.0	10.1
Kimi K2.5	27.9	36.9	14.2	11.7

At the model level, judge correctness correlates with Pass@1 at Spearman $\rho = 0.817$ (exact $p = 0.0108$) and with FB at $\rho = 0.883$ ($p = 0.0031$). Judge functional equivalence correlates with Pass@1 at $\rho = 0.783$ ($p = 0.0172$) and with FB at $\rho = 0.850$ ($p = 0.0061$). Pass@1 and FB themselves correlate at $\rho = 0.933$ ($p = 0.0007$). The judge is substantially stricter in absolute calibration because it considers API arguments, control flow, output contracts, and edge behavior. The correlations should therefore be interpreted as convergent evidence about model ranking rather than interchangeable task-level labels.

F Documentation-Retrieval Condition

The main benchmark is closed-book. As a complementary experiment, we implement a ReAct agent with BM25 retrieval over chunks of official RDKit documentation. Both rows in Table 8 use the same Opus 4.7 backbone, endpoint, temperature, and output budget, so the additional documentation-search and execution loop is the experimental intervention.

Retrieval does not improve this implementation. L1-L2 remain comparatively close, while the gap widens at L3-L4, where success depends on composing several operations under chemical constraints. Retrieved fragments do not consistently surface the task-relevant context, and errors introduced during

Table 8: Direct generation versus ReAct with RDKit documentation retrieval (%).

Setting	Searches	Exec. calls	Exec.	Pass@1	L1	L2	L3	L4	L5
Direct generation	0	0	93.3	36.3	77.3	45.8	23.6	22.7	7.8
ReAct + documentation retrieval	821	1,689	83.5	27.4	73.3	40.3	4.2	10.7	4.7

intermediate search or execution propagate. We report this as a result for this retrieval implementation, not as a general claim that access to documentation is harmful.

G Inference-Paradigm Visualization

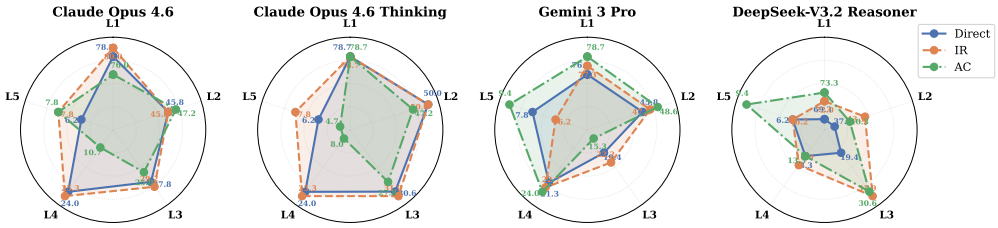

Figure 4: Comparison of the three inference paradigms. Direct denotes Direct Generation, IR denotes Incremental Repair, and AC denotes Agent Collaboration.

H Sensitivity Analysis of Fallback Pass Rate

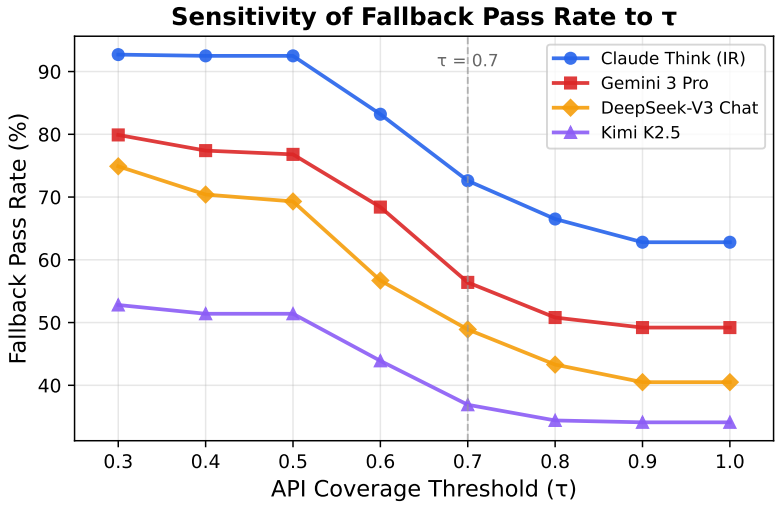

Figure 5: Fallback Pass Rate as a function of the API-coverage threshold τ. The dashed vertical line marks $\tau = 0.7$ adopted in the main evaluation. All models exhibit a clear inflection between $\tau = 0.6$ and $\tau = 0.8$.

Figure 5 and Table 9 report how Fallback Pass Rate (FB) varies with the API-coverage threshold τ across four representative models.

Threshold selection. All models exhibit a monotonic decline in FB as τ increases. The steepest drop occurs between $\tau = 0.6$ and $\tau = 0.8$. The largest single-step decrease is concentrated at $\tau = 0.6 \rightarrow 0.7$, with drops of 10.6 points for Claude Think IR, 12.0 for Gemini 3 Pro, 7.8 for DeepSeek-V3, and 7.0 for Kimi K2.5. This region forms a natural decision boundary. We therefore

select $\tau = 0.7$, which (i) lies within the transition region, (ii) provides meaningful model separation with a spread of approximately 36 points, and (iii) maintains non-trivial selectivity without being overly permissive.

Boundary behaviour. At $\tau = 0.3$, thresholds are permissive to the point of being uninformative: even the weakest model (Kimi K2.5) reaches 52.8%. Conversely, at $\tau \geq 0.9$ the curves flatten, indicating that requiring near-perfect API overlap excludes very few additional predictions and adds little discriminative power.

FB vs. EM at strict overlap. At $\tau = 1.0$, FB remains substantially above Exact Match for all models (e.g., Claude Think IR: 62.8% vs. 33.2% EM). Thus, even identical functional-category sets do not imply output agreement. This is why FB is reported as workflow-coverage diagnostics rather than substituted for the primary output-level metric.

Table 9: Fallback Pass Rate (%) at varying τ thresholds. Underlined values correspond to $\tau = 0.7$ used in the main paper.

Model	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Claude Think (IR)	92.7	92.5	92.5	83.2	<u>72.6</u>	66.5	62.8	62.8
Gemini 3 Pro	79.9	77.4	76.8	68.4	<u>56.4</u>	50.8	49.2	49.2
DeepSeek-V3 Chat	74.9	70.4	69.3	56.7	<u>48.9</u>	43.3	40.5	40.5
Kimi K2.5	52.8	51.4	51.4	43.9	<u>36.9</u>	34.4	34.1	34.1

I Chinese Experiment Results

Table 10: Cross-lingual comparison (English vs. Chinese prompts). L1–L5 report per-level Pass@1_{soft} (%). Δ is CN$-$EN.

Model	Lang	Exec.	EM	P@1	FB	L1	L2	L3	L4	L5
Claude Opus 4.6	EN	94.1	31.0	37.4	68.4	78.7	45.8	27.8	24.0	6.2
	CN	92.5	30.7	38.3	71.8	80.0	54.2	31.9	14.7	6.2
Gemini 3 Pro	EN	94.1	29.3	34.9	56.4	76.0	45.8	19.4	21.3	7.8
	CN	94.4	32.7	37.7	61.5	80.0	48.6	23.6	24.0	7.8
GPT-5.3 Codex	EN	89.7	27.9	32.7	64.8	76.0	55.6	16.7	5.3	6.2
	CN	94.1	29.6	36.9	71.5	78.7	62.5	27.8	4.0	7.8
DeepSeek-V3.2 Chat	EN	84.6	20.9	28.2	48.9	68.0	40.3	15.3	6.7	7.8
	CN	87.2	24.0	33.2	50.6	73.3	44.4	27.8	9.3	7.8
DeepSeek-V3.2 Reasoner	EN	74.6	24.9	29.9	45.5	69.3	37.5	19.4	13.3	6.2
	CN	76.8	22.9	29.3	45.5	69.3	40.3	18.1	10.7	4.7

Because frontier models are increasingly developed and deployed across both English- and Chinese-speaking ecosystems, prompt language may become a non-trivial source of benchmark variation. Evaluating the same tasks in both languages allows us to assess whether measured capability reflects underlying molecular coding competence or is partly conditioned on linguistic formulation. Table 10 compares English and Chinese prompts across five representative models to quantify this effect.

Contrary to the expectation that English-centric API documentation would favor English prompts, four of the five models achieved equal or higher Pass@1 scores with Chinese prompts. GPT-5.3 Codex exhibited the most significant improvement (+4.2 pp in Pass@1, +6.7 pp in Fallback), primarily driven by gains in L2 (+6.9 pp) and L3 (+11.1 pp). Similarly, DeepSeek-V3.2 Chat gained +5.0 pp in Pass@1, aligning with its Chinese-heavy training data. In contrast, DeepSeek-V3.2 Reasoner was the only model to experience a slight performance decline with Chinese prompts (-0.6 pp Pass@1), characterized by a drop in L4 accuracy from 13.3% to 10.7%. This suggests that its reasoning chain may be less effective when the initial problem statement is in Chinese, despite the language-agnostic

nature of the generated code. Overall, performance disparities for commonly used languages are minimal, and a Chinese version is provided to support diverse research groups and objectives.

J PROMPT TEMPLATES

We present the complete prompt templates used in each inference paradigm. All prompts enforce a unified function interface (`level_function`) and restrict library usage to RDKit, NumPy, pandas, scikit-learn, matplotlib, and SELFIES.

J.1 DIRECT GENERATION

System prompt.

```
You are an expert cheminformatics programmer.
Write Python code using RDKit to solve molecular computing tasks.
Your code must define a function called `level_function` that takes
the specified input and returns the result.
Only use these libraries: rdkit, numpy, pandas, scikit-learn,
matplotlib, selfies.
Include error handling with try/except blocks.
Do NOT include any import statements for libraries other than the
ones listed above.
Return only the Python code, no explanations.
```

User prompt.

```
Write a Python function `level_function` to accomplish the
following task:

Task: {question}

Requirements:
1. Function must be named `level_function`
2. Use the RDKit library
3. Include try/except error handling
4. Return None for invalid input

Provide the complete Python code.
```

J.2 INCREMENTAL REPAIR

The Incremental Repair uses the same system prompt and initial user prompt as direct generation. When the generated code fails execution, the following repair prompt is appended to the conversation history:

Repair prompt.

````
The code you provided failed during execution. Here is the error:

```
{error}
```

The test was run as:
```python
result = level_function({test_input})
```

Please fix the code and provide the corrected complete Python code.
The function must still be named `level_function`.
Return only the Python code, no explanations.
````

J.3 AGENT COLLABORATION

The Agent Collaboration paradigm separates code generation (R1, Coder) from validation (R2, Tester).

R1 (Coder) system prompt.

```
You are an expert cheminformatics programmer. You write Python
functions using RDKit to solve molecular science tasks.

Rules:
- The function MUST be named `level_function`
- Include all necessary imports inside or above the function
- Handle invalid inputs gracefully (return None)
- Only use: rdkit, numpy, pandas, scikit-learn, matplotlib, selfies
- Return ONLY the Python code, no explanations
```

R2 (Tester) test plan design prompt.

```
Given a molecular computing task, design a test plan BEFORE seeing
any code.

Task: {question}

Available test molecules (SMILES): {molecules}

Design your test plan as JSON with these fields:
{"test_smiles": [...], "expected_type": "int|float|str|...",
 "value_description": "...", "reasonable_range": "...",
 "failure_modes": [...]}
```

R2 (Tester) verification prompt.

```
You designed this test plan for the task:
Task: {question}
Test plan: {test_plan}

The code was executed with these results: {exec_results}

Evaluate:
1. Does the code run without errors on all test molecules?
2. Is the output type correct?
3. Is the output chemically reasonable?
4. Are there any obvious chemical logic errors?

Respond with JSON:
{"verdict": "PASS"} or {"verdict": "FAIL", "reason": "..."}
```

R1 (Coder) fix prompt. When A2 returns a FAIL verdict, A1 receives the following prompt with A2's feedback:

```
Your previous code for this task was tested and FAILED.

Task: {question}
Your previous code: {code}
Test results: {exec_results}
Tester feedback: {feedback}

Fix the code. IMPORTANT:
- Focus on fixing the SPECIFIC issue reported by the tester
- Do NOT change the overall chemical logic unless the tester
  explicitly says it is wrong
- Return the complete fixed Python code
```

K Workflow Coverage for Molecular Discovery

To ensure practical relevance, we also present the distribution of questions across 12 categories of real-world molecular discovery operations, as detailed in Table 11.

Table 11: Coverage of drug discovery workflow categories in MolViBench.

Category	Tasks	Representative Operations
Molecular Characterization	∼75	MW, LogP, TPSA, fingerprints, scaffolds
ADMET Screening	∼30	BBB permeability, CYP450, hERG, Ames, PAINS
Virtual Screening	∼25	Tanimoto search, cascade filtering, scoring
Combinatorial Chemistry	∼20	R-group enumeration, BRICS recombination
Lead Optimization	∼30	LogP/TPSA tuning, QED optimization, SA score
Reaction Chemistry	∼25	Suzuki, Click, Buchwald–Hartwig, retrosynthesis
Clustering & Selection	∼15	Butina, K-means, MaxMin, representative picking
QSAR / ML	∼15	Descriptor selection, RF regression, scaffold split
Molecular Generation	∼15	Derivative enumeration, genetic algorithm, quality metrics
Cheminformatics I/O	∼20	SDF/PDB parsing, CSV pipelines, batch processing
Stereochemistry	∼15	R/S assignment, E/Z detection, isomer enumeration
Rule-Based Filtering	∼25	Lipinski, Veber, Ghose, Egan, multi-rule cascades

L Complete Task List

Tables 12–16 list all 358 tasks in MolViBench, organized by Bloom's taxonomy level.

Table 12: Level 1: Remember & Understand (75 tasks). Basic API recall, format conversion, and single-step descriptor extraction.

#	Task Description
1	Calculate the molecular weight of a given molecule.
2	Calculate the molecular formula of a given molecule.
3	Calculate the LogP value of a given molecule.
4	Calculate the number of hydrogen bond donors of a molecule.
5	Calculate the number of hydrogen bond acceptors of a molecule.
6	Determine whether a molecule contains an aromatic ring.
7	Determine whether a molecule contains a hydroxyl group (-OH).
8	Determine whether a molecule contains a carboxyl group (-COOH).
9	Determine whether a molecule contains an amino group (-NH2).
10	Determine whether a molecule contains halogen atoms.
11	Extract the topological polar surface area (TPSA) of a molecule.
12	Output the canonical SMILES of a molecule.
13	Output the InChI representation of a molecule.
14	Visualize a molecule as a 2D SVG.
15	Get the number of atoms in a molecule.
16	Get the number of bonds in a molecule.
17	Get the number of rings in a molecule.
18	Determine whether a molecule contains a six-membered ring.
19	Determine whether a molecule contains a five-membered ring.
20	Determine whether a molecule contains a heterocyclic ring.
21	Output all atom symbols in a molecule.
22	Output all bond types in a molecule.
23	Determine whether a molecule is charged.
24	Calculate the number of rotatable bonds in a molecule.
25	Calculate the number of possible stereocenters in a molecule.

Continued on next page

#	Task Description
26	Determine whether a molecule satisfies Lipinski's Rule of Five.
27	Calculate the QED value (drug-likeness score) of a molecule.
28	Output the molar refractivity of a molecule.
29	Get the distribution of valence electrons for atoms in a molecule.
30	Determine whether a molecule is chiral.
31	Convert a molecule to an adjacency matrix.
32	Convert a molecule to an atom feature matrix.
33	Convert a molecule to molblock format.
34	Determine whether two SMILES are equivalent.
35	Check whether a molecular SMILES is valid.
36	Determine whether two molecules are constitutional isomers (ignoring stereoisomerism).
37	Extract substructure match coordinates from a molecule.
38	Given a SMARTS pattern, determine whether it matches.
39	Determine whether a molecule contains an aromatic nitrogen.
40	Determine whether a molecule contains a sulfur atom.
41	Determine whether a molecule contains a phosphorus atom.
42	Determine whether a molecule contains a metal element.
43	Get the list of functional groups in a molecule.
44	Count the number of oxygen atoms in a molecule.
45	Count the number of nitrogen atoms in a molecule.
46	Count the number of carbon atoms in a molecule.
47	Count the number of sulfur atoms in a molecule.
48	Count the number of halogen atoms in a molecule.
49	Count the number of hydrogen atoms in a molecule.
50	Determine whether two molecules are constitutional isomers.
51	Convert a molecule's SMILES to its SELFIES representation.
52	Extract the largest organic fragment from a salt-containing SMILES.
53	Output the R/S absolute configuration of all chiral centers.
54	Output the E/Z configuration of all double bonds.
55	Calculate the exact molecular weight of a deuterium-labeled molecule.
56	Extract the Murcko scaffold of a molecule.
57	Calculate the Gasteiger partial charge for each atom.
58	Calculate the BertzCT complexity index.
59	Calculate the Balaban J index.
60	Determine whether a molecule is a macrocycle ($\geq$12-membered ring).
61	Validate an invalid SMILES input and return a meaningful error message.
62	Output the SMARTS pattern representation of a molecule.
63	Get the hybridization type (sp/sp2/sp3) of each atom.
64	Calculate the Labute approximate surface area (ASA).
65	Calculate atom-wise Crippen contribution decomposition for MR.
66	Calculate the synthetic accessibility score (SA Score).
67	Calculate the Fsp3 value (fraction of sp3-hybridized carbons).
68	Calculate the heavy atom count.
69	Calculate the fraction of aromatic atoms relative to total heavy atoms.
70	Calculate the number of aliphatic rings.
71	Calculate the number of aromatic rings.
72	Determine whether a molecule satisfies Veber's rules.
73	Determine whether a molecule satisfies the Ghose filter.
74	Calculate the total count of NH and OH groups (NHOHCount).
75	Calculate the total count of nitrogen and oxygen atoms (NOCount).

Table 13: Level 2: Apply (72 tasks). Multi-tool composition with two-step transformations.

#	Task Description
1	Calculate the Tanimoto similarity of two molecules.
2	Calculate the Dice similarity of two molecules.
3	Calculate the Cosine similarity of two molecules.
4	Generate the Morgan fingerprint of a molecule (radius=2).
5	Generate the Morgan fingerprint of a molecule (radius=3).
6	Compare the Hamming distance of two molecular fingerprints.
7	Given benzene, generate all mono-substituted methyl derivatives.
8	Given benzene, generate all di-substituted methyl derivatives.
9	Given benzene, generate the para-substituted product.
10	Given benzene, generate the ortho-substituted product.
11	Given benzene, generate the meta-substituted product.
12	Replace a hydrogen in benzene with chlorine to generate chlorobenzene.
13	Replace the methyl group in toluene with a hydroxyl group.
14	Replace the hydroxyl group in phenol with a methoxy group.
15	Randomly replace one hydrogen in a molecule with fluorine.
16	Randomly replace one hydrogen in a molecule with chlorine.
17	Randomly replace one hydrogen in a molecule with bromine.
18	Randomly replace one hydrogen in a molecule with a nitro group.
19	Generate a 3D conformer of a molecule.
20	Generate multiple 3D conformers and return the lowest-energy one.
21	Optimize a molecule's 3D geometry using the MMFF94 force field.
22	Optimize a molecule's 3D geometry using the UFF force field.
23	Calculate the 3D molecular volume.
24	Calculate the 3D solvent-accessible surface area.
25	Calculate the radius of gyration of a molecule.
26	Calculate the 3D autocorrelation descriptors.
27	Calculate the WHIM descriptors of a molecule.
28	Calculate the RDF descriptors of a molecule.
29	Read molecules from an SDF file and return their SMILES.
30	Write a list of molecules to an SDF file.
31	Read molecules from a CSV file (SMILES column) and return Mol objects.
32	Write molecules with properties to a CSV file.
33	Convert a molecule to PDB format.
34	Read a molecule from a MOL2 file.
35	Remove all stereo information from a molecule.
36	Enumerate all possible stereoisomers of a molecule.
37	Neutralize charged groups in a molecule.
38	Standardize a molecule (remove fragments, neutralize, canonical tautomer).
39	Add hydrogens to a molecule.
40	Remove hydrogens from a molecule.
41	Generate the canonical tautomer of a molecule.
42	Fragment a molecule using BRICS decomposition.
43	Perform Murcko scaffold decomposition and return the generic scaffold.
44	Compute the shape similarity of two molecules (3D).
45	Align two molecules by their maximum common substructure.
46	Compute the USRCAT descriptor of a molecule.
47	Compute the Coulomb matrix of a molecule.
48	Cluster a set of molecules using the Butina algorithm.
49	Perform PCA on molecular fingerprints and return 2D coordinates.
50	Perform t-SNE on molecular fingerprints and return 2D coordinates.
51	Perform UMAP on molecular fingerprints and return 2D coordinates.
52	Generate the MACCS keys fingerprint of a molecule.
53	Generate the Atom Pair fingerprint of a molecule.
54	Generate the Topological Torsion fingerprint of a molecule.

Continued on next page

#	Task Description
55	Generate the RDKit fingerprint of a molecule.
56	Compute the maximum common substructure (MCS) of two molecules.
57	Perform BRICS decomposition and recombine fragments.
58	Perform R-group decomposition given a core scaffold.
59	Enumerate molecules from a SMILES with enumerable stereo centers.
60	Generate a combinatorial library from a scaffold with R-groups.
61	Compute the shape-based overlay of two molecules.
62	Compute the electrostatic similarity of two molecules.
63	Compute the pharmacophore fingerprint of a molecule.
64	Compute the 2D pharmacophore fingerprint of a molecule.
65	Compute the atom-pair fingerprint similarity of two molecules.
66	Compute the topological torsion fingerprint similarity of two molecules.
67	Perform MaxMin diversity picking from a molecule set.
68	Compute the molecular complexity using the Bertz index.
69	Compute the synthetic accessibility score and rank molecules.
70	Perform matched molecular pair (MMP) analysis on two molecules.
71	Cluster a set of molecules using Butina and return cluster IDs.
72	Compute the 2D pharmacophore fingerprint (Pharm2D) of a molecule.
73	Given a molecule and a substructure SMARTS, return all matching atom index tuples.

Table 14: Level 3: Analyze (72 tasks). Multi-step chemical reasoning requiring domain knowledge.

#	Task Description
1	Define an alcohol to alkyl halide reaction using SMARTS.
2	Define an esterification reaction using SMARTS.
3	Define an amidation reaction using SMARTS.
4	Simulate the nitration of benzene.
5	Simulate the sulfonation of benzene.
6	Simulate the nucleophilic substitution of an alkyl halide.
7	Simulate the oxidation of an alcohol to an aldehyde.
8	Simulate the oxidation of an aldehyde to a carboxylic acid.
9	Simulate the reduction of a carboxylic acid to an alcohol.
10	Simulate the reduction of a ketone to an alcohol.
11	Simulate the dehydration of an alcohol to form an alkene.
12	Simulate the hydrogenation of an alkene to form an alkane.
13	Simulate the halogenation of an alkene.
14	Simulate the hydrogenation of an alkyne to form an alkene.
15	Simulate the complete hydrogenation of an alkyne to form an alkane.
16	Simulate the Friedel-Crafts alkylation of an aromatic compound.
17	Simulate the Friedel-Crafts acylation of an aromatic compound.
18	Simulate the Diels-Alder reaction.
19	Simulate the Wittig reaction.
20	Simulate the Grignard reaction.
21	Simulate the aldol condensation reaction.
22	Simulate the Claisen rearrangement.
23	Simulate the Cope rearrangement.
24	Simulate the Beckmann rearrangement.
25	Simulate the Baeyer-Villiger oxidation.
26	Simulate the Swern oxidation.
27	Simulate the Birch reduction.
28	Simulate the Michael addition.
29	Simulate the Mannich reaction.
30	Simulate the Heck reaction.
31	Simulate the Sonogashira coupling.

Continued on next page

#	Task Description
32	Enumerate all possible E/Z isomers of a molecule.
33	Enumerate all possible R/S stereoisomers of a molecule.
34	Determine if two molecules are enantiomers.
35	Determine if two molecules are diastereomers.
36	Determine if a molecule is a meso compound.
37	Identify all stereocenters and assign R/S configuration.
38	Convert between different stereochemical representations.
39	Determine the optical activity relationship between two molecules.
40	Generate all possible tautomers and rank by stability.
41	Perform a random single-point mutation on a molecule.
42	Perform a random single-bond deletion on a molecule.
43	Perform a random atom insertion on a molecule.
44	Perform a random ring opening on a molecule.
45	Predict CYP450 metabolism sites of a molecule.
46	Calculate the drug-likeness score using multiple criteria.
47	Identify potential reactive metabolites.
48	Calculate the polar surface area contribution of each atom.
49	Predict blood-brain barrier permeability.
50	Calculate the number of Lipinski violations and identify which rules are violated.
51	Perform retrosynthetic analysis to identify possible precursors.
52	Identify potential toxicophores in a molecule.
53	Calculate the ligand efficiency metrics.
54	Compute the shape complementarity of two molecules.
55	Identify bioisosteric replacements for a functional group.
56	Calculate the free energy of solvation estimate.
57	Predict aqueous solubility using ESOL model.
58	Extract all unique Murcko scaffolds from a molecule set.
59	Compute the scaffold diversity of a molecule set.
60	Simulate the Suzuki coupling reaction.
61	Simulate the Click Chemistry (azide-alkyne cycloaddition) reaction.
62	Simulate the Buchwald-Hartwig amination.
63	Analyze macrocycle conformer distribution.
64	Simulate covalent inhibitor warhead attachment.
65	Detect structural alerts (Brenk filter).
66	Detect PAINS substructures.
67	Detect reactive functional groups.
68	Detect aggregator-like substructures.
69	Apply multi-rule filtering (Lipinski + Veber + PAINS).
70	Compute pharmacophore fingerprint similarity between two molecules.
71	Perform scaffold-based train/test split.
72	Compute validity, uniqueness, and novelty metrics for generated SMILES.

Table 15: Level 4: Analyze & Evaluate (75 tasks). Branching, iteration, and multi-step decision-making.

#	Task Description
1	Aromatic ring detection → if yes, replace H with OH → calculate LogP.
2	Carboxyl group detection → if yes, convert to ester → calculate MW.
3	Amino group detection → if yes, acetylate → calculate TPSA.
4	Halogen detection → if yes, replace with H → calculate LogP.
5	Benzene ring detection → if yes, nitrate → calculate MR.
6	Double bond detection → if yes, hydrogenate → calculate MW.
7	Hydroxyl group detection → if yes, chlorinate → calculate formula.
8	Ketone group detection → if yes, reduce to alcohol → calculate QED.

Continued on next page

#	Task Description
9	Ester group detection → if yes, hydrolyze → calculate LogP.
10	Amide group detection → if yes, hydrolyze → calculate TPSA.
11	Nitro group detection → if yes, reduce to amine → calculate pKa estimate.
12	Sulfide detection → if yes, oxidize to sulfoxide → calculate TPSA.
13	Aldehyde detection → if yes, oxidize to acid → calculate MW.
14	Alkene detection → if yes, epoxidize → calculate SA Score.
15	Aromatic amine detection → if yes, diazotize → calculate LogP.
16	Phenol detection → if yes, methylate → calculate QED.
17	Thiol detection → if yes, oxidize to disulfide → calculate MW.
18	Nitrile detection → if yes, hydrolyze to amide → calculate TPSA.
19	Carbamate detection → if yes, hydrolyze → calculate LogP.
20	Lactone detection → if yes, ring-open → calculate MW.
21	Epoxide detection → if yes, ring-open with water → calculate LogP.
22	Anhydride detection → if yes, hydrolyze → calculate TPSA.
23	Acyl chloride detection → if yes, hydrolyze → calculate MW.
24	Imine detection → if yes, reduce to amine → calculate pKa estimate.
25	Vinyl halide detection → if yes, Heck couple → calculate MW.
26	Boronic acid detection → if yes, Suzuki couple → calculate LogP.
27	Terminal alkyne detection → if yes, Sonogashira couple → calculate MW.
28	Primary alcohol detection → if yes, Swern oxidize → calculate LogP.
29	Secondary amine detection → if yes, reductive aminate → calculate MW.
30	Azide detection → if yes, Click react → calculate TPSA.
31	Diene detection → if yes, Diels-Alder react → calculate MW.
32	Grignard reagent detection → if yes, add to carbonyl → calculate LogP.
33	Enol detection → if yes, tautomerize to keto → calculate stability.
34	Peroxide detection → if yes, reduce → calculate MW.
35	Phosphonate detection → if yes, Wittig react → calculate MW.
36	Sulfonamide detection → if yes, hydrolyze → calculate TPSA.
37	Isocyanate detection → if yes, react with amine → calculate MW.
38	Acid chloride detection → if yes, Friedel-Crafts acylate → calculate LogP.
39	Michael acceptor detection → if yes, Michael add → calculate MW.
40	Enamine detection → if yes, hydrolyze → calculate LogP.
41	Oxime detection → if yes, Beckmann rearrange → calculate MW.
42	Allyl group detection → if yes, Claisen rearrange → calculate LogP.
43	Diol detection → if yes, periodate cleave → calculate MW.
44	Hemiacetal detection → if yes, oxidize → calculate LogP.
45	Aromatic halide detection → if yes, Buchwald-Hartwig aminate → calculate TPSA.
46	Ketone alpha-H detection → if yes, aldol condense → calculate MW.
47	Conjugated diene detection → if yes, 1,4-add → calculate LogP.
48	Strained ring detection → if yes, ring-open → calculate MW.
49	Protecting group detection → if yes, deprotect → calculate MW.
50	Anomeric center detection → if yes, glycosylate → calculate MW.
51	Macrocycle detection → if yes, conformer sample → calculate radius of gyration.
52	MW-driven BRICS decomposition vs. fragment growing.
53	Iterative LogP optimization via functional group modification.
54	PAINS filtering → QED ranking → return top 3.
55	Iterative polar group addition targeting TPSA range.
56	Iterative side-chain replacement with QED convergence.
57	Lipinski-guided mutation with property monitoring.
58	Backtracking molecular weight targeting (increase path).
59	Backtracking molecular weight targeting (decrease path).
60	Error recovery for invalid SMILES in batch processing.
61	Sequential reaction application with fallback on failure.
62	Multi-molecule scaffold swapping with property comparison.
63	Scaffold hopping with similarity constraint.
64	CSV pipeline: read → filter → compute → write.

Continued on next page

#	Task Description
65	SDF pipeline: read → filter → compute → write.
66	Multi-round iterative generation with diversity constraint.
67	Multi-round iterative generation with property convergence.
68	Multi-condition Lipinski decision tree.
69	Isomer-conditional stereochemistry analysis.
70	Derivative generation with SA Score ranking.
71	Multi-rule classification (drug-like / lead-like / fragment-like).
72	Similarity search with structural alert filtering.
73	Combinatorial enumeration with QED ranking.
74	Cluster-wise MCS extraction.
75	Cascade filtering pipeline (Lipinski → Veber → PAINS → Brenk).
76	Substructure search with descriptor ranking (Top-5).

Table 16: Level 5: Create (64 tasks). End-to-end pipeline construction and system-level integration.

#	Task Description
1	Generate new molecule candidates with similarity >0.7 to known actives.
2	Generate all mono-substituted derivatives of a target molecule.
3	Generate 10 molecules with different side chain modifications from a scaffold.
4	Generate molecules meeting pharmacophore requirements.
5	Extract common substructure scaffold from a molecule set.
6	Generate fragment-growing derivatives from a binding pocket fragment.
7	Generate ring-opening/ring-closing isomers of a drug candidate.
8	Generate stereoisomers and retain all feasible conformations.
9	Enumerate all halogen-substituted variants via substitution.
10	Build a candidate library based on structural similarity.
11	Screen molecules satisfying Lipinski + Veber + QED > 0.5.
12	Multi-property filtering: MW, LogP, TPSA, rotatable bonds.
13	ADMET-like filtering pipeline with multiple criteria.
14	Diversity-based subset selection from a large library.
15	Similarity-based virtual screening against a query molecule.
16	Substructure-based filtering with property ranking.
17	Pharmacophore-based virtual screening.
18	Shape-based virtual screening using 3D conformers.
19	Multi-objective Pareto ranking of molecules.
20	Consensus scoring combining multiple similarity metrics.
21	Lead optimization: iterative modification to improve QED.
22	Lead optimization: reduce LogP while maintaining activity.
23	Lead optimization: improve metabolic stability estimate.
24	Lead optimization: reduce molecular weight while keeping potency proxy.
25	Bioisosteric replacement optimization.
26	Fragment-based lead growing with property constraints.
27	Scaffold hopping with multi-property optimization.
28	R-group optimization with SAR analysis.
29	Multi-parameter optimization (MPO) scoring and ranking.
30	Matched molecular pair-guided optimization.
31	Multi-objective analysis: QED vs. SA Score trade-off.
32	Descriptor correlation analysis across a molecule set.
33	Structure-activity relationship (SAR) cliff detection.
34	Chemical space coverage analysis using fingerprint diversity.
35	Scaffold frequency analysis and distribution.
36	Property distribution analysis with statistical summary.
37	Functional group frequency analysis across a library.
38	Molecular complexity distribution analysis.

Continued on next page

#	Task Description
39	Selectivity analysis between two targets.
40	ADMET property radar chart generation.
41	Multi-endpoint activity prediction and ranking.
42	Generation → filtering → optimization pipeline.
43	BRICS recombination → filtering → ranking pipeline.
44	Scaffold decoration → ADMET filtering → diversity selection.
45	Fragment linking → property optimization → ranking.
46	Enumeration → clustering → representative selection.
47	Mutation → filtering → similarity-constrained selection.
48	Multi-step reaction → product filtering → property ranking.
49	Library enumeration → diversity analysis → subset selection.
50	Conformer generation → shape screening → ranking.
51	Retrosynthesis → feasibility scoring → route ranking.
52	MaxMin diversity selection from a fingerprint matrix.
53	BRICS fragment recombination with property constraints.
54	Scaffold morphing between two molecules.
55	QSAR regression model: fingerprint → Random Forest → R^2 and RMSE.
56	QSAR classification model: fingerprint → classifier → AUC and accuracy.
57	Matched molecular pair analysis with property delta.
58	Focused library design around a seed molecule.
59	Genetic algorithm-based molecular optimization.
60	Combinatorial library design: scaffold + R-groups → enumerate → filter → select.
61	Virtual screening pipeline: library → Lipinski → similarity → ADMET → rank.
62	De novo design: fragment → grow → optimize → validate.
63	SAR table generation: analogs → descriptors → activity cliff → report.
64	QSAR with featurization → Random Forest regression → R^2 and RMSE.
65	Applicability domain analysis: fingerprint space → boundary → decision.

M CASE STUDIES

We present one representative case study per Bloom level, comparing model outputs to illustrate recurring failure patterns. For each case, we show the task, the reference solution code, representative model-generated code, and analysis.

M.1 LEVEL 1: CALCULATE QED DRUG-LIKENESS SCORE (TASK 27)

Task. *"Calculate the QED value (drug-likeness score) of a molecule."*

Reference solution.

```
from rdkit import Chem
from rdkit.Chem import Descriptors

def level_function(mol):
    try:
        mol = Chem.MolFromSmiles(mol)
        if mol is None:
            return None
        return Descriptors.qed(mol)
    except Exception as e:
        print(e)
        return None
```

Claude Opus 4.6 (*Correct*). Uses `QED.qed(mol)` from `rdkit.Chem.QED`, an equivalent API that returns the same scalar value.

```
from rdkit import Chem
from rdkit.Chem import QED

def level_function(smiles: str):
    try:
```

```
        if smiles is None or not isinstance(smiles, str) or smiles.strip() == "":
            return None
        mol = Chem.MolFromSmiles(smiles)
        if mol is None:
            return None
        qed_value = QED.qed(mol)
        return qed_value
    except Exception:
        return None
```

DeepSeek-V3.2 Reasoner (*Partially Correct*). Uses `QED.default(mol)` which returns a named tuple of property contributions, not the scalar QED score.

```
from rdkit import Chem
from rdkit.Chem import QED

def level_function(molecule_input):
    try:
        if isinstance(molecule_input, str):
            mol = Chem.MolFromSmiles(molecule_input)
        elif isinstance(molecule_input, Chem.rdchem.Mol):
            mol = molecule_input
        else:
            return None
        if mol is None:
            return None
        qed_value = QED.default(mol)  # BUG: returns named tuple, not float
        return qed_value
    except Exception:
        return None
```

Key observation. Even at Level 1, subtle API confusion can cause failures. `QED.qed()` and `QED.default()` both exist in RDKit's `QED` module, but return fundamentally different objects (float vs. named tuple). This illustrates why our evaluation framework uses type-aware comparison rather than simple string matching, and our API Fallback could also help relieve the problem of the function selection.

M.2 Level 2: Maximum Common Substructure (Task 56)

Task. *"Compute the maximum common substructure (MCS) of two molecules."*

Reference solution.

```
from rdkit import Chem
from rdkit.Chem import rdFMCS

def level_function(smiles1, smiles2):
    try:
        mol1 = Chem.MolFromSmiles(smiles1)
        mol2 = Chem.MolFromSmiles(smiles2)
        if mol1 is None or mol2 is None:
            return None
        mcs_result = rdFMCS.FindMCS([mol1, mol2])
        if mcs_result.canceled or mcs_result.smartsString == "":
            return None
        return mcs_result.smartsString
    except Exception:
        return None
```

Claude Opus 4.6 (*Different returned contract*). Returns a dictionary with additional metadata instead of the requested primary result.

```
from rdkit import Chem
from rdkit.Chem import rdFMCS

def level_function(smiles1: str, smiles2: str):
    try:
        mol1 = Chem.MolFromSmiles(smiles1)
        mol2 = Chem.MolFromSmiles(smiles2)
        if mol1 is None or mol2 is None:
            return None
        mcs_result = rdFMCS.FindMCS([mol1, mol2])
        if mcs_result.canceled or not mcs_result.smartsString:
```

```
            return None
        return {
            "smarts": mcs_result.smartsString,
            "num_atoms": mcs_result.numAtoms,
            "num_bonds": mcs_result.numBonds,
        }
    except Exception:
        return None
```

DeepSeek-V3.2 Reasoner (*Different returned contract*). Also wraps the result in a dictionary with additional fields including `cancelled` status.

```
from rdkit import Chem
from rdkit.Chem import rdFMCS

def level_function(smiles1, smiles2):
    try:
        mol1 = Chem.MolFromSmiles(smiles1)
        mol2 = Chem.MolFromSmiles(smiles2)
        if mol1 is None or mol2 is None:
            return None
        result = rdFMCS.FindMCS([mol1, mol2])
        return {
            "smarts": result.smartsString,
            "numAtoms": result.numAtoms,
            "numBonds": result.numBonds,
            "cancelled": result.canceled,
        }
    except Exception:
        return None
```

Key observation. All models identify and use `rdFMCS.FindMCS()`, demonstrating API recall at Level 2, but several fail the requested return contract by adding unrequested fields. This example illustrates why API recall and end-to-end task satisfaction must be reported separately.

M.3 Level 3: Click Chemistry Reaction (Task 62)

Task. *"Simulate a Click Chemistry (azide-alkyne cycloaddition) reaction."*

Reference solution.

```
from rdkit import Chem
from rdkit.Chem import AllChem

def level_function(azide_smiles, alkyne_smiles):
    try:
        azide = Chem.MolFromSmiles(azide_smiles)
        alkyne = Chem.MolFromSmiles(alkyne_smiles)
        if azide is None or alkyne is None:
            return None
        rxn_smarts = ('[C:1]#[C:2].[N:3]=[N+:4]=[N-:5]'
                      '>>[N:3]1-[N:4]=[N:5]-[C:1]=[C:2]1')
        rxn = AllChem.ReactionFromSmarts(rxn_smarts)
        products = rxn.RunReactants((alkyne, azide))
        if not products:
            return None
        results = set()
        for prod_set in products:
            for prod in prod_set:
                for atom in prod.GetAtoms():
                    atom.SetFormalCharge(0)
                Chem.SanitizeMol(prod)
                results.add(Chem.MolToSmiles(prod))
        return sorted(results)
    except Exception:
        return None
```

Claude Opus 4.6 (*Partially correct*). Restricts alkyne to terminal alkynes via `[CH1]`, returns single SMILES instead of list.

```
from rdkit import Chem
from rdkit.Chem import AllChem

def level_function(azide_smiles, alkyne_smiles):
```

```
    try:
        azide = Chem.MolFromSmiles(azide_smiles)
        alkyne = Chem.MolFromSmiles(alkyne_smiles)
        if azide is None or alkyne is None:
            return None
        # BUG: [CH1] restricts to terminal alkynes only
        rxn = AllChem.ReactionFromSmarts(
            '[C:1]#[CH1:2].[N:3]=[N+:4]=[N-:5]'
            '>>[c:1]1[n:3][n:4][n:5][c:2]1')
        products = rxn.RunReactants((alkyne, azide))
        if not products:
            return None
        product = products[0][0]
        Chem.SanitizeMol(product)
        return Chem.MolToSmiles(product)  # returns str, not list
    except Exception:
        return None
```

GPT-5.2-Codex (*Incorrect*). Forces aromaticity on triazole ring with lowercase `[n]` atoms, producing wrong product structure.

```
from rdkit import Chem
from rdkit.Chem import rdChemReactions

def level_function(reactants):
    try:
        if isinstance(reactants, str):
            parts = reactants.split(".")
        else:
            parts = list(reactants)
        azide_smi, alkyne_smi = parts[0], parts[1]
        # BUG: aromatic [n] atoms force incorrect aromaticity
        rxn_smarts = ('[N-:1]=[N+:2]=[N:3].[C:4]#[C:5]'
                      '>>[n:1]1[n:2][c:4][c:5][n:3]1')
        rxn = rdChemReactions.ReactionFromSmarts(rxn_smarts)
        azide = Chem.MolFromSmiles(azide_smi)
        alkyne = Chem.MolFromSmiles(alkyne_smi)
        products = rxn.RunReactants((azide, alkyne))
        if not products:
            return None
        return Chem.MolToSmiles(products[0][0])
    except Exception:
        return None
```

Key observation. Level 3 tasks expose the gap between "knowing about" a reaction and correctly encoding it in SMARTS. Small atom-mapping differences (e.g., `[CH1]` vs. `[C]`, aromatic `[n]` vs. aliphatic `[N]`) produce fundamentally different chemical outcomes. Two models (not shown) also add irrelevant 3D computation before a purely 2D SMARTS-based transformation.

M.4 Level 4: PAINS Filtering with QED Ranking (Task 54)

Task. *"Given a list of molecules → check each for PAINS substructures → filter out PAINS-containing molecules → calculate QED for remaining → return the top 3 with highest QED."*

Reference solution.

```
from rdkit import Chem
from rdkit.Chem import QED
from rdkit.Chem.FilterCatalog import FilterCatalog, FilterCatalogParams

def level_function(smiles_list):
    try:
        params = FilterCatalogParams()
        params.AddCatalog(FilterCatalogParams.FilterCatalogs.PAINS)
        catalog = FilterCatalog(params)
        passed = []
        for smi in smiles_list:
            mol = Chem.MolFromSmiles(smi)
            if mol is None:
                continue
            if not catalog.HasMatch(mol):
                qed_val = QED.qed(mol)
                passed.append({"smiles": smi, "QED": round(qed_val, 4)})
        passed.sort(key=lambda x: x["QED"], reverse=True)
        return {
```

```
            "total_input": len(smiles_list),
            "pains_filtered": len(smiles_list) - len(passed),
            "remaining": len(passed),
            "top3": passed[:3],
        }
    except Exception:
        return None
```

Claude Opus 4.6 (*Output-contract failure*). Implements PAINS filtering but returns tuples instead of the specified records.

```
from rdkit import Chem
from rdkit.Chem import QED
from rdkit.Chem.FilterCatalog import FilterCatalog, FilterCatalogParams

def level_function(smiles_list):
    try:
        params = FilterCatalogParams()
        params.AddCatalog(FilterCatalogParams.FilterCatalogs.PAINS)
        catalog = FilterCatalog(params)
        clean_mols = []
        for smi in smiles_list:
            mol = Chem.MolFromSmiles(smi)
            if mol is None:
                continue
            if not catalog.HasMatch(mol):
                clean_mols.append((smi, QED.qed(mol)))
        clean_mols.sort(key=lambda x: x[1], reverse=True)
        return clean_mols[:3]  # returns list of tuples, not dict
    except Exception:
        return None
```

DeepSeek-Reasoner (*Correct logic, wrong output type*). Implements PAINS filtering correctly but returns raw Mol objects instead of SMILES strings.

```
from rdkit import Chem
from rdkit.Chem import QED
from rdkit.Chem.FilterCatalog import FilterCatalog, FilterCatalogParams

def level_function(smiles_list):
    try:
        params = FilterCatalogParams()
        params.AddCatalog(FilterCatalogParams.FilterCatalogs.PAINS)
        catalog = FilterCatalog(params)
        results = []
        for smi in smiles_list:
            mol = Chem.MolFromSmiles(smi)
            if mol and not catalog.HasMatch(mol):
                results.append((mol, QED.qed(mol)))  # BUG: Mol object
        results.sort(key=lambda x: x[1], reverse=True)
        return results[:3]
    except Exception:
        return None
```

Key observation. The models recall the PAINS filtering operation, yet several fail the complete task contract. API-workflow coverage captures this procedural alignment, whereas Pass@1 continues to require the applicable output-level decision rule.

M.5 Level 5: Combinatorial Library Design Pipeline (Task 60)

Task. *"Combinatorial library design: scaffold + R-groups → enumerate all combinations → filter by Lipinski + QED → diversity selection → output final library with properties."*

Reference solution (abbreviated, with the full version in the released code).

```
from rdkit import Chem, DataStructs
from rdkit.Chem import AllChem, Descriptors, QED, rdMolDescriptors
from rdkit.Chem.FilterCatalog import FilterCatalog, FilterCatalogParams
import numpy as np

def level_function(scaffold_smiles, rgroup_lists):
    try:
        scaffold = Chem.MolFromSmiles(scaffold_smiles)
```

```
        if scaffold is None:
            return None
        # Stage 1: Find dummy atom positions [*]
        dummy_indices = [a.GetIdx() for a in scaffold.GetAtoms()
                         if a.GetAtomicNum() == 0]
        # Stage 2: Enumerate all R-group combinations
        from itertools import product
        combos = list(product(*rgroup_lists))
        # Stage 3: Attach R-groups by replacing dummy atoms
        library = []
        for combo in combos:
            mol = Chem.RWMol(scaffold)
            # ... atom-level manipulation to attach R-groups ...
            library.append(Chem.MolToSmiles(mol))
        # Stage 4: Filter by Lipinski + QED > 0.5
        filtered = [smi for smi in library
                    if passes_lipinski(smi) and get_qed(smi) > 0.5]
        # Stage 5: MaxMin diversity picking
        selected = maxmin_pick(filtered, n=50)
        # Stage 6: Return with properties
        return {"total_enumerated": len(library),
                "after_filter": len(filtered),
                "selected": len(selected),
                "molecules": [get_props(s) for s in selected]}
    except Exception:
        return None
```

Claude Opus 4.6 (*Largely correct*). Implements all 6 stages. Uses `Chem.ReplaceSubstructs()` for R-group attachment, which is a valid alternative to manual atom editing. Correctly implements Lipinski filtering and QED thresholding. Uses fingerprint-based greedy diversity selection (slightly different from MaxMin but functionally similar).

```
def level_function(scaffold_smiles, rgroup_lists):
    try:
        scaffold = Chem.MolFromSmiles(scaffold_smiles)
        dummy = Chem.MolFromSmarts("[#0]")  # match dummy atoms
        # Enumerate combinations
        from itertools import product
        combos = list(product(*rgroup_lists))
        library = []
        for combo in combos:
            mol = Chem.RWMol(scaffold)
            for i, rg_smi in enumerate(combo):
                rg = Chem.MolFromSmiles(rg_smi)
                # ReplaceSubstructs for attachment
                mol = AllChem.ReplaceSubstructs(mol, dummy, rg)[0]
            Chem.SanitizeMol(mol)
            library.append(Chem.MolToSmiles(mol))
        # Filter + diversity select + return props
        # ... (correct implementation) ...
    except Exception:
        return None
```

DeepSeek-Reasoner (*Incorrect*). Attempts to use `AllChem.EnumerateLibraryFromReaction()` with a reaction SMARTS, which requires a different input format. The reaction SMARTS is incorrectly constructed, causing enumeration to fail silently and return an empty library.

```
def level_function(scaffold_smiles, rgroup_lists):
    try:
        # BUG: Constructs invalid reaction SMARTS
        rxn_smarts = "[*:1]>>[" + scaffold_smiles + ":1]"
        rxn = AllChem.ReactionFromSmarts(rxn_smarts)
        # EnumerateLibraryFromReaction expects different input format
        library = AllChem.EnumerateLibraryFromReaction(
            rxn, rgroup_lists)
        products = [Chem.MolToSmiles(m[0]) for m in library]
        # products is empty due to SMARTS error
        # ... rest of pipeline never executes meaningfully ...
    except Exception:
        return None
```

Key observation. Level 5 tasks are the most discriminating because they require correct implementation of *every* pipeline stage, and a single-stage failure cascades into overall failure. The most common

failure mode is not in filtering or ranking, which most models handle well. It appears in the *molecular construction* stage when the program must attach R-groups to scaffolds, handle dummy atoms, and manage combinatorial enumeration. This suggests that LLMs struggle most with low-level molecular manipulation operations that have no direct analogue in general-purpose programming.

N LIMITATIONS

MolViBench has several limitations. First, it focuses on RDKit as the primary cheminformatics toolkit, enabling deterministic and reproducible evaluation but leaving broader multi-toolkit workflows, such as those involving OpenBabel or DeepChem, for future work. Second, although the benchmark contains 358 tasks spanning 12 workflow categories and multiple cognitive levels, further expansion can be needed to cover emerging cheminformatics applications and a wider range of workflow patterns. Finally, no automatic metric can fully capture the diversity of valid implementations for open-ended molecular coding tasks. We therefore combine deterministic Pass@1 with FB and an independent LLM-judge analysis, but these signals remain complementary rather than interchangeable notions of correctness.

O STATEMENT OF ETHICS

All tasks in MolViBench were manually authored by domain experts and do not involve any human subjects or personal data. Molecular inputs used for evaluation are drawn from the publicly available ZINC250K dataset (Sterling & Irwin, 2015). All evaluated LLMs were accessed through hosted APIs in accordance with the applicable terms of service. The benchmark covers standard, publicly documented cheminformatics operations and does not introduce any capabilities beyond those already available through open-source toolkits and academic literature. This research conforms with the ICLR Code of Ethics.

P BROADER IMPACTS

MolViBench advances the responsible development of AI-assisted molecular discovery by providing a rigorous evaluation framework that exposes subtle failure modes, such as *runnable but chemically wrong* outputs. This work lowers barriers for researchers without deep computational chemistry expertise, enabling broader and more reliable use of LLMs across diverse research communities. We caution that model-generated cheminformatics code should always be validated by domain experts before use in high-stakes decision-making, and we encourage future work to incorporate chemical safety constraints as an explicit evaluation dimension.