MDGYM: BENCHMARKING AI AGENTS ON MOLECULAR SIMULATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

The promise of AI-driven scientific discovery hinges on whether AI agents can autonomously design and execute the computational workflows that underpin modern science. Molecular dynamics (MD) simulation provides a demanding test of this capability: it requires translating a physical problem into syntactically and semantically correct input scripts, choosing force fields, ensembles, and initial and boundary conditions, diagnosing engine errors and unstable trajectories, and interpreting outputs against known physical behavior. We introduce MDGYM, a benchmark of 200 expert-curated MD simulation problems in LAMMPS and GROMACS, two widely used MD engines, organized into three difficulty levels and comprising 366 target quantities scored against reference values from expert simulations. We evaluate three general-purpose command-line agent frameworks, Claude Code, Codex, and OpenHands, each with four LLMs. The best configuration solves 68% of easy problems, 30% of medium problems, and 21% of hard problems, and the choice of model changes performance far more than the choice of framework. Trajectory and engine-error analyses show that most engine errors arise while agents assemble the simulation input, that weaker models fail early through syntax errors and premature termination, and that stronger models complete more simulations but report incorrect values from error-free runs. Current agents therefore lack three capabilities required for autonomous simulation: coherent specification of simulation inputs, domain-aware recovery from engine errors, and physical and methodological validation of results.

1 INTRODUCTION

Autonomous scientific discovery, in which AI systems formulate hypotheses, design experiments, execute computational workflows, and interpret results against physical theory, has become a central goal of AI research (Wang et al., 2023; Lu et al., 2024; Boiko et al., 2023). Agents operating in scientific domains must choose an appropriate methodology, design experiments grounded in the relevant physical concepts, and interpret numerical outputs against known physical laws to arrive at meaningful conclusions. Correctness in such tasks is defined by physical validity, which compilers and test suites for software do not verify (Bisht et al., 2026). Determining whether current AI agents possess these capabilities, and where they fall short, is therefore a prerequisite for a credible path toward autonomous scientific discovery – we test this using our novel molecular dynamics benchmark

Molecular dynamics (MD) simulation, a method for simulating the behavior of materials and molecules at the atomic scale, is central to drug discovery (Ahmed et al., 2023), materials design (Choi, 2023), and biophysics (Sinha et al., 2022). A successful MD simulation requires converting a scientific problem stated in natural language into a physically grounded model, justifying each modeling choice against the real system, and analyzing the results with methods appropriate to the relevant length and time scales. For systems of realistic complexity, the outcome of the simulation, such as the equilibrium density of a polymer melt or the diffusion coefficient of ions in solution, is unknown until the simulation is run. MD simulations therefore resemble laboratory experiments more closely than standard coding tasks and are often referred to as *in silico* experiments.

An agent performing MD must translate a physical description of a system into syntactically and semantically correct input scripts, reason about initial and boundary conditions, select appropriate force fields and integration parameters, monitor trajectories for numerical instability, and interpret

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.

output quantities against known physical behavior. These competencies must be composed across a multi-stage pipeline of system setup, initialization, energy minimization, equilibration, production, and post-processing. A physically incoherent choice at any stage can yield a simulation that runs to completion and produces output without raising any error, yet whose result is physically meaningless. Since the correct outcome is unknown *a priori*, the agent must detect and recover from such errors through scientific reasoning alone. The combination of long-horizon, multi-stage execution with domain knowledge makes MD simulation a demanding setting in which to examine the question at the heart of AI for science: can AI agents autonomously design and execute scientific experiments?

Existing benchmarks for MD agents address parts of this question. They evaluate small task sets, target a single engine or a single class of systems, or evaluate whether codes are executed without comparing the simulated quantities against reference values (Section 2). In response, we introduce **MDGYM**, an expert-curated benchmark of 200 MD simulation problems in two widely used engines, LAMMPS (Thompson et al., 2022) and GROMACS (Hess et al., 2008), organized into three levels of increasing difficulty. The problems cover diverse types of materials such as biomolecules, metals and alloys, polymers, and carbon-based nanomaterials in crystalline and amorphous solid, liquid, and gaseous states. Each problem reflects a scientific question encountered in a research setting, is developed and verified by domain experts, and is scored automatically against physically meaningful reference quantities. Using MDGYM, we evaluate three general-purpose command-line agent frameworks, Claude Code, Codex, and OpenHands, each instantiated with four LLMs spanning frontier proprietary and open-weight models. Our contributions are as follows.

- **Benchmark:** We release MDGYM, comprising 200 simulation problems with 366 target quantities across 76 physical properties and 160 knowledge questions, together with a modular evaluation harness that extends to additional agents and MD engines, for tracking progress toward autonomous scientific computing.
- **Evaluation:** We evaluate 12 agent–model configurations on MDGYM. The best configuration solves 68% of easy problems and 30% of medium and 21% of hard problems; the choice of model changes performance far more than the choice of agent framework.
- **Diagnosis:** We present a taxonomy of failure modes, derived from trajectory analysis and engine-level error classification, that localizes where in the simulation pipeline current agents fail. Our taxonomy identifies the capabilities that agents must acquire before they can serve as reliable partners in computational science.

2 RELATED WORKS

MD simulation integrates Hamilton's equations of motion for an atomistic system, with interatomic forces derived from an empirical force field that approximates the quantum mechanical potential energy surface. Thermostats and barostats extend the microcanonical (NVE) ensemble to constant-temperature and constant-pressure conditions, and non-equilibrium protocols such as uniaxial loading, planar shear, or imposed thermal gradients yield mechanical and transport properties. Physical validity requires a timestep that resolves the fastest vibrational mode, a force field applied within its domain of parameterization, ensemble and coupling constants that match the target conditions, and an analysis procedure appropriate to the measured quantity. Refer App. A for a discussion on MD simulations.

Benchmarks for coding agents. Since MD simulation scripts are written in specialized programming languages, our task is related to other coding tasks. Popular enchmarks for code generation and repository-level software engineering, such as HumanEval-V (Zhang et al., 2024) and SWE-bench (Jimenez et al., 2023), have largely been saturated. Terminal-Bench (Merrill et al., 2026) evaluates agents on 89 realistic terminal tasks spanning scientific computing, software engineering, and system administration. SUPER (Bogin et al., 2024) evaluates the setup and execution of 45 end-to-end problems from ML and NLP research repositories, and SciReplicate-Bench (Xiang et al., 2025) the reproduction of algorithms from research papers. AInsteinBench (Duston et al., 2025) extends this line to scientific repositories and finds that agents struggle to preserve scientific invariants and maintain correctness in complex scientific algorithms, and DevOps-Gym (Tang et al., 2026) covers the software DevOps cycle. In these benchmarks, correctness is syntactic or behavioral, and is verified by test suites, stack traces, or output matching, so errors surface as signals on which an agent can act; correctness in MD simulation is physical and lacks such a verifier.

Table 1: Benchmarks (excluding QA) for scientific AI agents that require code generation, simulation, or experimentation. Counts follow each benchmark's own definition of a problem.

Benchmark	Task type	Problems
QUASAR (Yang & Evans, 2026)	Atomistic simulation (DFT, MLP, MD, MC)	9
DynaMate (Guilbert et al., 2025)	Protein and protein–ligand MD workflows	12
MDCrow (Campbell et al., 2025)	Protein MD through predefined tools	25
MD-EvalBench (Shi et al., 2026)	LAMMPS code generation	566
SUPER (Bogin et al., 2024)	Executing ML and NLP research repositories	45
BixBench (Mitchener et al., 2025)	Bioinformatics data analysis	53
SciCode (Tian et al., 2024)	Research coding in the natural sciences	80
AILA (Mandal et al., 2025)	Atomic force microscopy experiments	100
ScienceAgentBench (Chen et al., 2024)	Data-driven scientific discovery	102
MDGYM (ours)	**End-to-end MD in LAMMPS and GROMACS**	**200**

Benchmarks for scientific AI agents. A growing body of benchmarks evaluates LLM agents on scientific tasks (Table 1); App. B reviews AI scientist systems. ScienceAgentBench (Chen et al., 2024) draws its tasks from peer-reviewed publications in four disciplines. SciCode (Tian et al., 2024) decomposes research-level problems from 16 natural-science subfields into subproblems and evaluates LLMs on these problems. BixBench (Mitchener et al., 2025) analyzes frontier models on bioinformatics analysis scenarios in the open-answer regime. AILA (Mandal et al., 2025) evaluates LLM agents that operate an atomic force microscope in live laboratory experiments, and LAB-Bench (Laurent et al., 2024) tests practical biology research through over 2,400 multiple-choice questions on literature reasoning, database navigation, and sequence manipulation. More recently, Ríos-García et al. (2026) evaluated AI agents across eight environments and more than 25,000 runs. While earlier works focused on QA abilities of LLMs, recent works evaluate the agentic their abilities to solve scientific questions or a part thereof. MDGYM contributes to this family of work by framing MD simulation as an end-to-end problem to be solved by agents.

Agents and benchmarks for MD simulation. Existing agents for MD target specific systems or workflows. MDCrow (Campbell et al., 2025) automates protein simulation tasks through a predefined set of tools; DynaMate (Guilbert et al., 2025) is a multi-agent framework for end-to-end workflows on protein and protein–ligand systems; PolyJarvis (Zhao et al., 2026) couples an LLM with the RadonPy platform to predict polymer properties; and NAMD-Agent (Chandrasekhar & Farimani, 2025) automates protein simulations in NAMD. These agents are evaluated on small task sets defined by their authors, ranging from 4 to 40 problems for the MD benchmarks in Table 1.The closest prior work, MDAgent2 (Shi et al., 2026), extends MDAgent (Shi et al., 2025) with a multi-agent system for LAMMPS code generation, execution, and self-correction, and introduces MD-EvalBench for LAMMPS code generation and question answering. MD-EvalBench covers a single engine and scores each generated script by whether it executes and by expert ratings of the script, without comparing the simulated quantities against reference values. MDGYM addresses these gaps with 200 expert-curated problems across LAMMPS and GROMACS at three difficulty levels, each scored against physically meaningful reference quantities (Table 1), together with a modular harness that extends to additional engines such as AMBER, NAMD, and OpenMM. Since every problem requires executing simulations to completion, this scale entails a substantial computational cost (Sec. 3).

3 MDGYM DATASET & EVALUATION HARNESS

MDGYM comprises 200 simulation problems and 160 knowledge QA tasks, developed and curated by researchers with expertise in MD simulation. The problems are drawn from research publications and tutorials or constructed from scratch. GROMACS problems adapt input files from the GROMACS repository,[1] published studies (Rajput et al., 2023; Koch et al., 2025), and tutorials (Lemkul, 2018; Gravelle et al., 2025); LAMMPS problems adapt examples from the official LAMMPS repository (Thompson et al., 2022) and tutorials (Gravelle et al., 2025). Every sourced file is modified, for instance by substituting a different material, ensemble, or loading condition, so that no problem reduces to running an existing example verbatim, and each problem poses a research-level question. The knowledge

[1] https://www.gromacs.org/

questions are built on the problem statements and test syntax, such as correct command usage, and concepts, such as physical reasoning about ensembles and force fields (App. C.5).

Most problems follow a common protocol of energy minimization, NVT and/or NPT equilibration, and a production run from which the observables are extracted by post-processing; the harder problems additionally apply non-equilibrium protocols such as deformation, shear, or tensile loading. Reference values are obtained by executing expert-written simulation scripts in GROMACS and LAMMPS and computing the observables with Python scripts or with the analysis tools native to each engine. Each problem is stored as a JSON specification (App. C.2) that holds the problem description, difficulty level, engine, required metrics, reference values, and time limit; the orchestrator generates the agent prompt from this specification and passes the reference values to the validator.

Quality control. Problem curation and difficulty labelling were performed by a postdoctoral researcher with expertise in MD simulation and verified by a senior investigator; difficulty labels were assigned before any agent evaluation. Each draft problem was further reviewed by an ensemble of LLMs for ambiguous wording, incomplete specifications, and errors, and refined manually. This cycle was repeated until every problem satisfied three criteria: *correctness*, providing sufficient and accurate detail for the simulation to be reproduced; *credibility*, reflecting a realistic molecular simulation research scenario of appropriate complexity; and *clarity*, defining the expected output format and evaluation criteria unambiguously (App. C.4 gives an example). Finally, the trajectory of every reference simulation was inspected visually in VMD or OVITO to confirm physically realistic structural evolution and equilibration.

Validation by human experts. To test whether the reference values can be reproduced independently from the problem descriptions, five evaluators working in MD simulation (four PhD students and one Master's student) each simulated 10 systems, five in GROMACS and five in LAMMPS, balanced across difficulty levels, using the same structure and topology files supplied to the agents. Of the 95 physical quantities evaluated across these 50 systems, 87 (91.6%) fall within the 5% tolerance of the reference values. The remaining eight deviate are property types, e.g., radius of gyration of polymers, whose values vary between independent simulations by more than 5% owing to finite sampling. For these properties, MDGYM therefore adopts higher tolerances, determined from the observed variance across independent reference runs, applied identically to agents and human evaluators. Under this adjusted tolerance, all 95 quantities agree with the reference values. This reproduction rate also provides a human reference for the agent results in Section 5.

Dataset statistics. MDGYM contains 200 simulation problems with 366 target quantities over 76 distinct properties, together with 160 knowledge questions (Fig. 1). GROMACS is generally preferred for biomolecular simulations of proteins, lipids, and nucleic acids, whereas LAMMPS offers the flexibility suited to materials, polymers, and coarse-grained systems (App. C.3); the two engines together cover biomolecules, metals and alloys, polymers, and carbon-based nanomaterials in crystalline and amorphous solid, liquid, and gaseous states. The properties fall into six classes, with the number of distinct properties in parentheses: thermodynamic (24), mechanical (24), structural (13), transport (6), magnetic (2), and other or numerical (7). Thermodynamic properties account for 58% of target quantities, reflecting their importance in standard MD workflows (App. C.6).

The problems are divided into three levels of difficulty according to the properties they require. *Level 1* (Easy) problems require thermodynamic properties, such as temperature, pressure, and total energy per particle, that need minimal post-processing of the native engine output. *Level 2* (Medium) problems require properties that need substantial post-processing, such as coordination numbers, the average radius of gyration, hydrogen-bond statistics, and energy components other than the potential or total energy; the level is assigned by the property, irrespective of whether the agent computes it during the run or afterwards. *Level 3* (Hard) problems require specialized simulations beyond equilibration runs, such as non-equilibrium deformation, shear, or tensile protocols and multi-stage runs, or post-processing by curve fitting, such as extracting a diffusion coefficient from the slope of the mean-squared displacement; examples include Young's modulus, shear viscosity, diffusion coefficients, thermal conductivity, and magnetic properties.

Since every problem requires running simulations, MDGYM is considerably more expensive per problem than language-only or code-generation benchmarks: under the time budget of Sec. 4, a single evaluation pass of one agent–model configuration requires about 82 hours of wall-clock time. The size of 200 problems balances coverage of MD workflows against the cost of evaluating many

configurations, and Table 1 places it in the context of existing benchmarks. The problem set can be expanded systematically by varying the material, force field, or cutoff within existing simulation templates, without redesigning the underlying problems.

Evaluation harness. MDGYM is implemented as a modular evaluation harness with three independently extensible layers (App. C). The *agent layer* provides a common interface to any command-line agent: each implementation runs the agent as a subprocess, streams its actions and outputs to a structured trajectory log, and returns control upon completion or timeout. The *orchestration layer* coordinates a single evaluation episode by building the task prompt, invoking the agent, and passing its reported output to the validator, independently of the agent and engine in use. The *validator layer* compares the reported values with the reference values of each engine-specific task and assigns the score defined in Section 4. Each layer is defined by an abstract interface, so that a new agent, a new MD engine such as AMBER, NAMD, or OpenMM, or a new scoring metric is added by implementing a single class without modifying the other components.

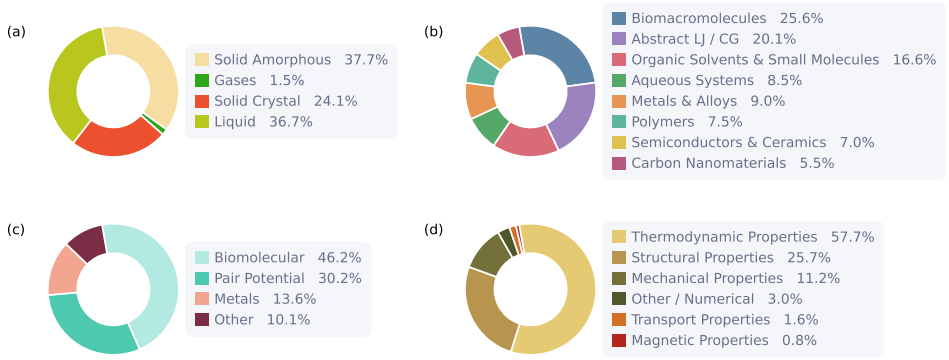


Figure 1: Statistical overview of the molecular dynamics simulation dataset across four classification dimensions. Donut charts showing the distribution of simulations by (a) physical state, (b) material/system type, (c) force field category, and (d) computed property type.

4 EVALUATION SETUP

Agents and models. We evaluate three general-purpose command-line agent frameworks, Claude Code, Codex, and OpenHands (Wang et al., 2024), each with four LLMs: Claude Opus 4.8, GPT 5.5, DeepSeek V4 Pro (Xu et al., 2026), and Qwen 3.8 27BQwen Team (2026), giving twelve agent–model configurations. All models are accessed through the OpenRouter API[2] , and models with configurable reasoning effort run at the provider's default. Every agent runs headlessly with permission prompts disabled, so that it executes shell commands and edits files autonomously. The system prompts, tool definitions, and configuration of each framework are reported in App. D. The execution environment provides LAMMPS (22 Jul 2025) and GROMACS (v2026.0) together with the required Python packages, and each problem supplies its starting files, such as structure, topology, and potential files. To evaluate the contributions of model and framework further, App. H reports additional combinations of eight LLMs and the three frameworks (24 combinations) on a subset of 50 Medium and Hard problems.

Task prompt. Each problem is presented as a single structured prompt generated from its JSON specification (Section 3; example in App. C.4). The prompt states the problem identifier, the target engine, a natural-language description of the simulation, the required output metrics, and the paths to the supplied input files. Agents write all scripts in a dedicated working directory and report their results in a `final_answer.json` file that maps each metric name to a unitless numerical value; a problem whose file is missing or cannot be parsed receives a score of zero.

Time budget. Each problem has a time budget of $300 + 3\,t_{\text{sim}}$ seconds, where t_{sim} is the wall-clock runtime of the reference simulation measured during ground-truth generation on 24-core AMD CPUs. The fixed 300 seconds allows the agent to read the task and plan, and the factor of 3 allows about three complete simulation attempts; an agent that exceeds the budget is terminated and the problem is

[2] `https://openrouter.ai/`

scored as failed. We express the budget in wall-clock time since simulation execution dominates the cost of each problem, and token budgets are not comparable across models with different tokenizers.

Evaluation metrics. For a problem with required metric set M, the score is $s = \frac{1}{|M|}\sum_{k\in M} \mathbf{1}\,[\text{passed}(k)]$, where $\text{passed}(k) = 1$ if the reported value of metric k lies within a relative tolerance of the reference value, and 0 otherwise. A problem is solved only if $s = 1$. For each configuration, we report the *full-success rate*, the fraction of problems solved, and the *total partial-credit score*, the sum of s over problems, each by difficulty level and engine together with a 95% Wilson confidence interval (App. I) over the same set of problems for every configuration. In addition, we inspect trajectories manually for answers that were guessed or recalled without running the simulation, and score such problems as failed.

Failure analysis. Three complementary analyses characterize how agents fail (App. E). A rule-based classifier assigns each failed trajectory to categories of an expert-curated taxonomy. An LLM judge (Claude Sonnet 5, majority vote over three classifications) assigns each engine error to predefined LAMMPS and GROMACS categories. A second LLM judge (Claude Sonnet 5) determines the last pipeline stage that each simulation completed. To isolate post-processing, we additionally provide agents with correct simulation outputs for 50 problems and ask them only to compute the target observables. To determine whether failures arise from missing domain knowledge, we evaluate the same models on MD-QA (Ríos-García et al., 2026) and on the MDGYM knowledge questions, curated set of questions which require expert level knowledge in MD simulations to answer.

5 RESULTS AND DISCUSSION

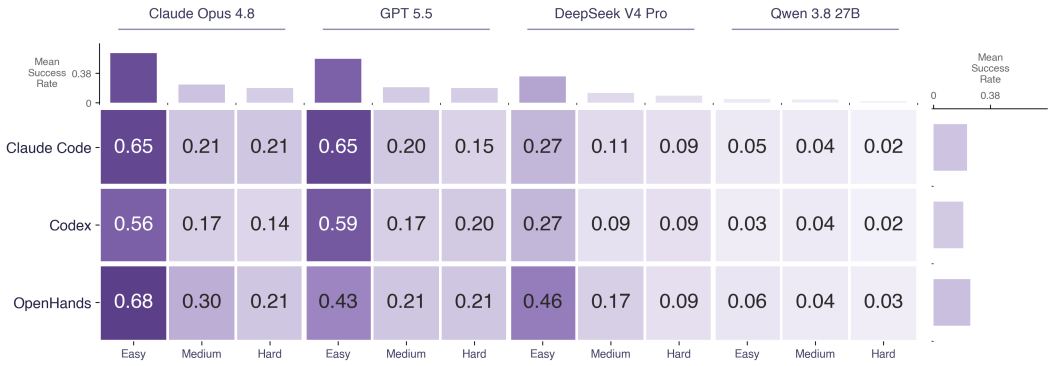


Figure 2: Full-success rate of each agent–model configuration by difficulty level on all MDGYM tasks. Bars above and to the right give mean values over models and frameworks, respectively.

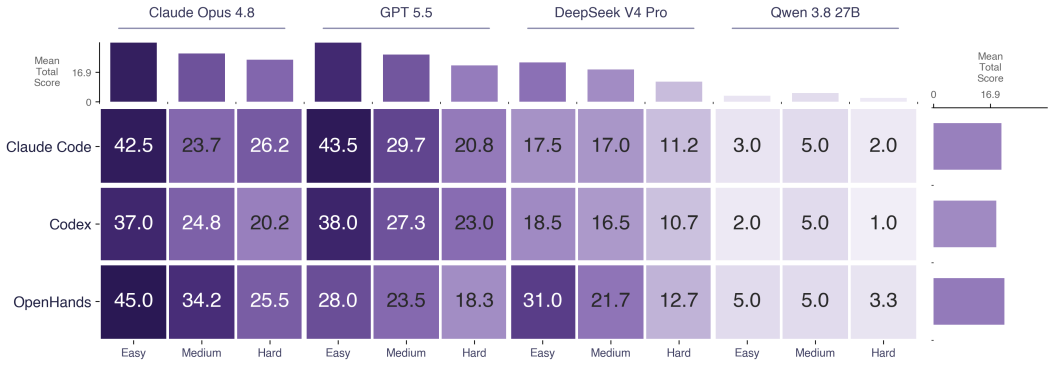


Figure 3: Partial-credit score, the sum of per-problem scores s, of each agent–model configuration by difficulty level. Bars above and to the right give means over models and frameworks, respectively.

Overall Performance. Figures 2 and 3 report the full-success rate and the total partial-credit score of the twelve agent–model configurations. On Easy problems, the best configuration of each

framework solves 65% (Claude Code with Claude Opus 4.8 or GPT 5.5), 59% (Codex with GPT 5.5), and 68% (OpenHands with Claude Opus 4.8) of the problems. These rates fall to 21%, 17%, and 30% on Medium problems and to 21%, 20%, and 21% on Hard problems. OpenHands with Claude Opus 4.8 attains the highest rate at every difficulty level, with a 95% Wilson interval of 56.0–78.4% on Easy problems (App. I). The two proprietary models outperform DeepSeek V4 Pro and Qwen 3.8 27B under every framework and at every difficulty level, with one exception: GPT 5.5 under OpenHands on Easy problems (43% against 46% for DeepSeek V4 Pro). Pooled over frameworks, the mean per-problem scores are 0.472, 0.426, 0.265, and 0.053 for Claude Opus 4.8, GPT 5.5, DeepSeek V4 Pro, and Qwen 3.8 27B. All pairwise differences between models are significant (Mann–Whitney U, $p < 0.001$) except that between Claude Opus 4.8 and GPT 5.5 ($p = 0.074$). App. H reports the remaining model–framework combinations. Doubling the time budget from 300 to 600 seconds on a subset of 50 problems improves the performance of Claude Sonnet 4.6 by only 4%.

Disaggregated by engine (Figs. G.3 and G.5), ten of the twelve configurations achieve a higher Easy success rate on GROMACS than on LAMMPS. The best Easy rate reaches 90% on GROMACS (GPT 5.5 under Claude Code and Codex) against 52% on LAMMPS (Claude Opus 4.8 under OpenHands). This gap narrows with difficulty. On Hard problems, Claude Opus 4.8 solves a higher fraction of LAMMPS problems than of GROMACS problems under all three frameworks: 28% against 13% under Claude Code, 14% against 13% under Codex, and 25% against 17% under OpenHands. From Easy to Hard, the success rates of Claude Opus 4.8 and GPT 5.5 decrease by 60 to 74 percentage points on GROMACS and by 11 to 31 percentage points on LAMMPS. The one exception on GROMACS is GPT 5.5 under OpenHands, which decreases by 30 points. The decrease in per-problem score with difficulty is significant for Claude Opus 4.8, GPT 5.5, and DeepSeek V4 Pro (Kruskal–Wallis, $p \leq 0.002$) and across all twelve configurations ($H = 51.29$, $p < 0.001$). Qwen 3.8 27B, whose scores are already close to zero on Easy problems, shows a trend in the same direction that is not significant ($p = 0.064$).

Varying the framework at fixed model changes the success rate far less than varying the model at fixed framework. On Easy problems, Claude Opus 4.8 solves 65.1%, 55.6%, and 68.3% of the problems under Claude Code, Codex, and OpenHands, and the three 95% Wilson intervals overlap. Under OpenHands, the Easy success rate ranges from 6% for Qwen 3.8 27B to 68% for Claude Opus 4.8. A two-way analysis of variance on per-problem scores confirms this asymmetry. Model identity explains 15.8% of the variance ($F = 149.37$, $p < 0.001$), framework identity explains 0.14% and is not significant ($F = 1.92$, $p = 0.147$), and their interaction explains 0.94% ($F = 4.42$, $p < 0.001$). The controlled comparison of eight LLMs under all three frameworks on 50 Medium and Hard problems gives the same result (Fig. H.8). For a fixed proprietary model, the frameworks differ by at most three problems, while across models the success rate ranges from 0% to 16%. The base model thus determines performance on MDGYM far more than the agent framework does.

Knowledge and execution. The same models answer MD knowledge questions accurately. On MD-QA, Claude Opus 4.8, GPT 5.5, DeepSeek V4 Pro, and Qwen 3.8 27B score 90%, 95%, 97.5%, and 88.75%, and on the 160 MDGYM knowledge questions they score 90%, 90%, 88%, and 80%. The contrast with the end-to-end success rates places the difficulty in applying this knowledge to configure, execute, and analyze a simulation, which the following research questions examine.

RQ1: Where in the simulation pipeline do agents fail? Figure 4 reports the percentage of runs that complete each stage of the pipeline. Attrition is far steeper on LAMMPS than on GROMACS. Claude Opus 4.8 declines from 92% of runs at initialization to 46% at production on LAMMPS, and from 99% to 87% on GROMACS, consistent with the engine-level gap in success rates. On LAMMPS,

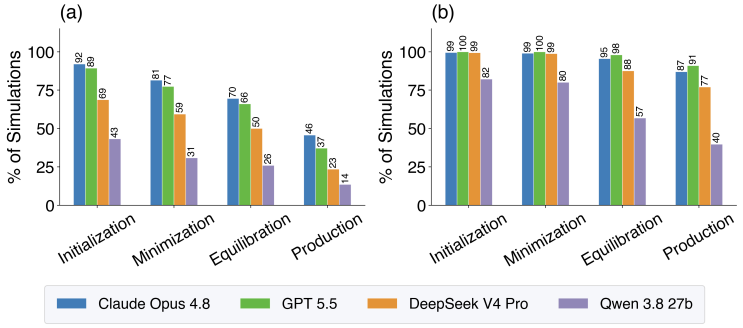


Figure 4: Percentage of runs completing each stage of the MD pipeline for each model, sum pooled over all harnesses, for (a) LAMMPS and (b) GROMACS problems. Stages are assigned by the classifier of App. E.

Claude Opus 4.8, GPT 5.5,
and DeepSeek V4 Pro lose the largest share of runs, 24 to 29%, between equilibration and production. Model capability shapes both the level and the shape of the curve. Claude Opus 4.8 and GPT 5.5 track each other closely on both engines, DeepSeek V4 Pro trails by a margin that widens at each stage. Qwen 3.8 27B fails earliest; 57% of its LAMMPS runs and 18% of its GROMACS runs do not complete initialization, and only 14% and 40% reach production. That said, completing production does not ensure a correct result. Claude Opus 4.8 and GPT 5.5 complete production in 87% and 91% of their GROMACS runs, and the remaining failures lie in the reported values, which RQ2 examines.

RQ2: What are the most common failure modes? Figure 5 reports the prevalence of five failure modes among failed trajectories (App. E). Syntax errors are the most prevalent mode for every model (30–59%), whereas fabricated answers (at most 6%) and environment misunderstanding (at most 5%) are rare. When agents fabricate, they either recall a plausible value from parametric memory or approximate it using standalone Python code instead of the simulation. Interestingly, Claude Opus 4.8 and DeepSeek V4 Pro both encounter syntax errors in 30% of their failed trajectories, yet Claude Opus 4.8 achieves a higher success rate across all frameworks and at every difficulty level. Claude Opus 4.8 and DeepSeek V4 Pro recover 18% and 14.3% of the time after encountering a syntax error. The remaining modes separate the models. Premature termination is most frequent for DeepSeek V4 Pro (25%) and Qwen 3.8 27B (18%), and least frequent for GPT 5.5 (4%). Incorrect post-processing, which comprises every failed trajectory whose answer came from an error-free run, accounts for 24% and 39% of the failures of Claude Opus 4.8 and GPT 5.5, against 7% and 2% for DeepSeek V4 Pro and Qwen 3.8 27B. Weaker models therefore fail early in the pipeline, and stronger models, having cleared the early stages, fail at obtaining a correct value from a completed simulation. The drop from produced to correct answers at every difficulty level shows the same pattern (Fig. G.7).

Incorrect post-processing combines two sources of error; the analysis of a correct simulation, and simulations that completed without error but were physically incorrect. To isolate the first, we perform a targeted ablation. Given correct simulation outputs for 50 problems and asked only to compute the target observables, Claude Sonnet 4.6 and DeepSeek V4 Pro each succeed on 72% of the problems, against 20% and 44% end-to-end on the same problems. Post-processing is therefore an independent source of error, and the remaining failures in this category reflect simulations that were configured incorrectly without raising any error.

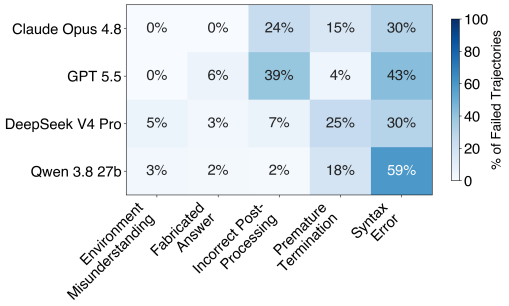


Figure 5: Prevalence of failure modes, as the percentage of failed trajectories exhibiting each mode. Modes are non-exclusive.

RQ3: Do agents fail at syntax, specification, or physics? We group the engine errors of Figure F.2 into four classes: *syntax* errors, where the engine cannot parse a command; *specification* errors, where individually valid commands define an incomplete or inconsistent simulation through its force-field assignments, data and topology files, atom groups, or command order; *execution-environment* errors, where the input requests parallel or accelerator features that the CPU-only environment does not provide; and *runtime instabilities*, where a running simulation becomes numerically or physically unstable. In LAMMPS, syntax errors (Command Syntax) account for 28.2% of all errors and specification errors for 47.0%, led by Pair/Force Field mis-specification (13.1%) and Data File and Read errors (10.6%). Execution-environment errors (Package/Partition and MPI/Parallel) account for 14.0%, and runtime instabilities, led by Energy Divergence (3.9%) and Lost Atoms (3.5%), for 10.7%. In GROMACS, 74.3% of errors arise during preprocessing by `grompp`, before the first MD step, and are dominated by Index Group (47.2%) and Topology (16.6%) errors. Of the remaining 25.7%, runtime instabilities (Pressure Coupling, Energy Blow-up etc.) account for 10.0%, and errors in electrostatics, box, and output settings (PME/Electrostatics, PBC/Box, and Output/File Write) for 15.0%.

In both engines, most engine errors therefore arise while agents assemble the simulation input, and a minority arise from the instability of a running simulation. The same models answer 88–90% of the MDGYM knowledge questions correctly, and 80% in the case of Qwen 3.8 27B. They know

the individual commands and concepts but fail to combine interdependent choices of force field, topology, atom groups, and execution settings into a specification consistent with the target engine and hardware. Physically incorrect simulations that run to completion raise no engine error; they appear in RQ2 among the incorrect answers from error-free runs.

RQ4: What do these failures collectively reveal? The analyses identify three capabilities that current agents lack for MD simulation. The first is *coherent specification*: combining the structure, force field, topology, atom groups, and execution settings into an input consistent with the target engine, the capability whose absence produces most engine errors in RQ3. The second is *domain-aware error recovery*: interpreting error messages and correcting the input accordingly, whose absence appears as premature termination in 4–25% of failed trajectories and as the stage attrition in RQ1. The third is *physical and methodological validation*: recognizing when a completed simulation or its analysis yields a physically incorrect value, and selecting the analysis procedure appropriate to the measured quantity; its absence accounts for 24% and 39% of the failures of Claude Opus 4.8 and GPT 5.5, and for the 28% of problems that remain unsolved when agents receive correct simulation outputs.

Each of these gaps guide future intervention strategies. The engine error logs collected in MDGYM provide pairs of error messages and corrective actions for fine-tuning recovery behavior. LAMMPS input scripts and GROMACS parameter and topology files are low-resource languages, underrepresented in pretraining corpora relative to Python or C++, which motivates targeted data collection and continued pretraining on simulation inputs. Physics-based validators and engine-specific error parsers can be added to the agent loop through the modular harness of MDGYM. The same requirements arise in other scientific computing domains with multi-stage pipelines and physically defined correctness criteria, such as density functional theory, finite element analysis, and computational fluid dynamics, where software engineering benchmarks likewise do not measure them.

6 CONCLUSION

We introduce MDGYM, a benchmark of 200 expert-curated MD simulation problems in LAMMPS and GROMACS at three difficulty levels, with 366 target quantities scored against reference values from expert simulations. Across twelve configurations of three general-purpose command-line agent frameworks and four LLMs, the best configuration solves 68% of Easy problems, 35% of Medium problems, and 33% of Hard problems. The base model determines performance far more than the framework. At fixed model, the Easy success rate varies by 6.4 percentage points across frameworks, whereas at fixed framework it varies by up to 50 percentage points across models. The same models answer 80–90% of the MDGYM knowledge questions correctly, which locates the difficulty in applying domain knowledge to configure, execute, and analyze a simulation. Error analysis reveals that most engine errors arise while agents assemble the simulation input, and few from the instability of a running simulation. Weaker models fail early in the pipeline, through syntax errors and premature termination, whereas stronger models complete more simulations but report incorrect values from error-free runs in 24–39% of their failed trajectories. These results identify three capabilities that agents must acquire to serve as reliable partners in computational science: coherent identification and specification of simulation inputs, domain-aware recovery from engine errors, and physical and methodological validation of results. We release the benchmark, the evaluation harness, and the agent trajectories and will maintain a public leaderboard to track progress on these capabilities. MDGYM code and dataset are available at code and dataset.

Limitations and future work. MDGYM covers two MD engines and does not yet include AMBER, NAMD, or OpenMM, which the modular harness supports. All problems use classical force fields and exclude ab initio MD and enhanced sampling methods such as metadynamics and replica exchange, which require additional methodological choices and form a natural extension to harder problems. We evaluate general-purpose command-line frameworks, some of whose system prompts carry software-engineering framing (App. C); development of agents designed for simulation, with physics-based validators or engine-specific repair loops, could further improve the performance. Each configuration is evaluated in a single run due to the high cost associated with the runs ($\sim$4000 USD for the entire project), and repeated runs would quantify run-to-run variance beyond the Wilson intervals of App. I. Finally, extending the benchmark to open-ended problems, in which agents must choose the method, judge convergence, and report the uncertainty of their results, would evaluate the scientific judgment that autonomous discovery requires.

AI USE STATEMENT

In this work, we used generative AI tools for code generation and text improvement. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artefacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The complete MDGYM benchmark (https://anonymous.4open.science/r/MDGym-Dataset-CFE8), including all 200 tasks, their JSON specifications (problem descriptions, required metrics, ground-truth values, and starting files), and the exact task prompts sent to agents, is available at the anonymized repository. Our evaluation harness code (https://anonymous.4open.science/r/MDBench-C1F5), including the agent, orchestration, and validator layers described in Section C, is publicly released. The exact prompts and classification rules used for our automated trajectory-level and simulation-level error analyses are provided in full in Appendix E.3, including the complete system prompts given to the LLM-based error classifier. Our statistical methodology, including the exact tests and software packages used to compute all reported significance results, is described in Appendix I, and the raw per-problem scores underlying these analyses are released alongside the benchmark to enable independent verification. Details of our human expert validation study, including the number of evaluators, their qualifications, and the exact validation protocol, are provided in Section 3.

REFERENCES

Mayar Ahmed, Alex M Maldonado, and Jacob D Durrant. From byte to bench to bedside: molecular dynamics simulations and drug discovery. *BMC biology*, 21(1):299, 2023.

Harshit Bisht, Vinay Kumar, Kevin Maik Jablonka, NM Krishnan, et al. Agentic ai scientists are not built for autonomous scientific discovery. *arXiv preprint arXiv:2605.08956*, 2026.

Ben Bogin, Kejuan Yang, Shashank Gupta, Kyle Richardson, Erin Bransom, Peter Clark, Ashish Sabharwal, and Tushar Khot. SUPER: evaluating agents on setting up and executing tasks from research repositories. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pp. 12622–12645. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.EMNLP-MAIN.702. URL https://doi.org/10.18653/v1/2024.emnlp-main.702.

Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with large language models. *Nat.*, 624(7992):570–578, 2023. doi: 10.1038/S41586-023-06792-0. URL https://doi.org/10.1038/s41586-023-06792-0.

Andres M. Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D. White, and Philippe Schwaller. Augmenting large language models with chemistry tools. *Nat. Mac. Intell.*, 6(5): 525–535, 2024. doi: 10.1038/S42256-024-00832-8. URL https://doi.org/10.1038/s42256-024-00832-8.

Quintina L. Campbell, Sam Cox, Jorge Medina, Brittany Watterson, and Andrew D. White. Mdcrow: Automating molecular dynamics workflows with large language models. *CoRR*, abs/2502.09565, 2025. doi: 10.48550/ARXIV.2502.09565. URL https://doi.org/10.48550/arXiv.2502.09565.

Achuth Chandrasekhar and Amir Barati Farimani. Automating MD simulations for proteins using large language models: Namd-agent. *CoRR*, abs/2507.07887, 2025. doi: 10.48550/ARXIV.2507.07887. URL https://doi.org/10.48550/arXiv.2507.07887.

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. *CoRR*, abs/2410.05080, 2024. doi: 10.48550/ARXIV.2410.05080. URL https://doi.org/10.48550/arXiv.2410.05080.

Joonmyung Choi. A review of the mechanical design of materials based on molecular dynamics simulations. *Multiscale Science and Engineering*, 5(3):86–103, 2023.

Keyan Ding, Jing Yu, Junjie Huang, Yuchen Yang, Qiang Zhang, and Huajun Chen. Scitoolagent: a knowledge-graph-driven scientific agent for multitool integration. *Nat. Comput. Sci.*, 5(10): 962–972, 2025. doi: 10.1038/S43588-025-00849-Y. URL https://doi.org/10.1038/s43588-025-00849-y.

Titouan Duston, Shuo Xin, Yang Sun, Daoguang Zan, Aoyan Li, Shulin Xin, Kai Shen, Yixiao Chen, Qiming Sun, Ge Zhang, Jiashuo Liu, Huan Zhou, Jingkai Liu, Zhichen Pu, Yuanheng Wang, Bo-Xuan Ge, Xin Tong, Fei Ye, Zhi-Chao Zhao, Wen-Biao Han, Zhoujian Cao, Yueran Zhao, Weiluo Ren, Qingshen Long, Yuxiao Liu, Anni Huang, Yidi Du, Yuanyuan Rong, and Jiahao Peng. Ainsteinbench: Benchmarking coding agents on scientific repositories. *CoRR*, abs/2512.21373, 2025. doi: 10.48550/ARXIV.2512.21373. URL https://doi.org/10.48550/arXiv.2512.21373.

Simon Gravelle, Cecilia M. S. Alvares, Jacob R. Gissinger, and Axel Kohlmeyer. A set of tutorials for the LAMMPS simulation package [article v1.0]. *Living Journal of Computational Molecular Science (LiveCoMS)*, 6(1):3037, 2025. doi: 10.33011/livecoms.6.1.3037.

Salomé Guilbert, Cassandra Masschelein, Jeremy Goumaz, Bohdan Naida, and Philippe Schwaller. Dynamate: An autonomous agent for protein-ligand molecular dynamics simulations. *CoRR*, abs/2512.10034, 2025. doi: 10.48550/ARXIV.2512.10034. URL https://doi.org/10.48550/arXiv.2512.10034.

Berk Hess, Carsten Kutzner, David Van Der Spoel, and Erik Lindahl. Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation*, 4(3):435–447, 2008.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *CoRR*, abs/2310.06770, 2023. doi: 10.48550/ARXIV.2310.06770. URL https://doi.org/10.48550/arXiv.2310.06770.

Leon Koch, Dominik Baier, Satyendra Rajput, Benedikt König, Michael Tiemann, Simon Ebbinghaus, Divya Nayar, and Klaus Huber. Disaggregation at high volume exclusion: An "overcrowding" effect. *The Journal of Physical Chemistry B*, 129(40):10213–10228, 2025. doi: 10.1021/acs.jpcb.5c01245. URL https://doi.org/10.1021/acs.jpcb.5c01245. PMID: 41026846.

Jon M. Laurent, Joseph D. Janizek, Michael Ruzo, Michaela M. Hinks, Michael J. Hammerling, Siddharth Narayanan, Manvitha Ponnapati, Andrew D. White, and Samuel G. Rodriques. Lab-bench: Measuring capabilities of language models for biology research. *CoRR*, abs/2407.10362, 2024. doi: 10.48550/ARXIV.2407.10362. URL https://doi.org/10.48550/arXiv.2407.10362.

Justin A. Lemkul. From proteins to perturbed hamiltonians: A suite of tutorials for the gromacs-2018 molecular simulation package [article v1.0]. *Living Journal of Computational Molecular Science*, pp. 5068, Oct. 2018. URL https://livecomsjournal.org/index.php/livecoms/article/view/v1i1e5068.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Indrajeet Mandal, Jitendra Soni, Mohd Zaki, Morten M. Smedskjaer, Katrin Wondraczek, Lothar Wondraczek, Nitya Nand Gosvami, and N. M. Anoop Krishnan. Evaluating large language model agents for automation of atomic force microscopy. *Nature Communications*, 16(1):9104, October 2025. ISSN 2041-1723. doi: 10.1038/s41467-025-64105-7. URL https://www.nature.com/articles/s41467-025-64105-7.

Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, Estefany Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas

Menis-Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *CoRR*, abs/2601.11868, 2026. doi: 10.48550/ARXIV.2601.11868. URL https://doi.org/10.48550/arXiv.2601.11868.

Ludovico Mitchener, Jon M Laurent, Alex Andonian, Benjamin Tenmann, Siddharth Narayanan, Geemi P Wellawatte, Andrew White, Lorenzo Sani, and Samuel G Rodriques. Bixbench: a comprehensive benchmark for llm-based agents in computational biology. *arXiv preprint arXiv:2503.00096*, 2025.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Satyendra Rajput, Roland Pollak, Klaus Huber, Simon Ebbinghaus, and Divya Nayar. Ethylene glycol energetically disfavours oligomerization of pseudoisocyanine dyestuffs at crowded concentrations. *Soft Matter*, 19(33):6399–6413, 2023.

Martiño Ríos-García, Nawaf Alampara, Chandan Gupta, Indrajeet Mandal, Sajid Mannan, Ali Asghar Aghajani, NM Krishnan, and Kevin Maik Jablonka. Ai scientists produce results without reasoning scientifically. *arXiv preprint arXiv:2604.18805*, 2026.

Zhuofan Shi, Chunxiao Xin, Tong Huo, Yuntao Jiang, Bowen Wu, Xingyue Chen, Wei Qin, Xinjian Ma, Gang Huang, Zhenyu Wang, et al. A fine-tuned large language model based molecular dynamics agent for code generation to obtain material thermodynamic parameters. *Scientific Reports*, 15(1):10295, 2025.

Zhuofan Shi, Yufei Shao, Mengyan Dai, Yadong Yu, Pan Xiang, Dongliang Huang, Hongxu An, Chunxiao Xin, Haiyang Shen, Zhenyu Wang, et al. Mdagent2: Large language model for code generation and knowledge q&a in molecular dynamics. *arXiv preprint arXiv:2601.02075*, 2026.

Siddharth Sinha, Benjamin Tam, and San Ming Wang. Applications of molecular dynamics simulation in protein study. *Membranes*, 12(9):844, 2022.

Yuheng Tang, Kaijie Zhu, Bonan Ruan, Chuqi Zhang, Michael Yang, Hongwei Li, Suyue Guo, Tianneng Shi, Zekun Li, Christopher Kruegel, Giovanni Vigna, Dawn Song, William Yang Wang, Lun Wang, Yangruibo Ding, Zhenkai Liang, and Wenbo Guo. Devops-gym: Benchmarking AI agents in software devops cycle. *CoRR*, abs/2601.20882, 2026. doi: 10.48550/ARXIV.2601.20882. URL https://doi.org/10.48550/arXiv.2601.20882.

Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271, February 2022. doi: 10.1016/j.cpc.2021.108171. URL https://www.sciencedirect.com/science/article/pii/S0010465521002836.

Minyang Tian, Luyu Gao, Shizhuo D Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, et al. Scicode: A research coding benchmark curated by scientists. *Advances in Neural Information Processing Systems*, 37:30624–30650, 2024.

Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, Anima Anandkumar, Karianne Bergen, Carla P. Gomes, Shirley Ho, Pushmeet Kohli, Joan Lasenby, Jure Leskovec, Tie-Yan Liu, Arjun K. Manrai, Debora S. Marks, Bharath Ramsundar, Le Song, Jimeng Sun, Jian Tang, Petar Velickovic, Max Welling, Linfeng Zhang, Connor W. Coley, Yoshua Bengio, and Marinka Zitnik. Scientific discovery in the age of artificial intelligence. *Nat.*, 620(7972):47–60, 2023. doi: 10.1038/S41586-023-06221-2. URL https://doi.org/10.1038/s41586-023-06221-2.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Yanzheng Xiang, Hanqi Yan, Shuyin Ouyang, Lin Gui, and Yulan He. Scireplicate-bench: Benchmarking llms in agent-driven algorithmic reproduction from research papers. *arXiv preprint arXiv:2504.00255*, 2025.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*, 2025.

Fengxu Yang and Jack D Evans. Quasar: A universal autonomous system for atomistic simulation and a benchmark of its capabilities. *Journal of Chemical Information and Modeling*, 66(10): 5911–5918, 2026.

Fengji Zhang, Linquan Wu, Huiyu Bai, Guancheng Lin, Xiao Li, Xiao Yu, Yue Wang, Bei Chen, and Jacky Keung. Humaneval-v: benchmarking high-level visual reasoning with complex diagrams in coding tasks. *arXiv preprint arXiv:2410.12381*, 2024.

Alexander Zhao, Achuth Chandrasekhar, and Amir Barati Farimani. Polyjarvis: Llm agent for autonomous polymer md simulations. *arXiv preprint arXiv:2604.02537*, 2026.

A MOLECULAR DYNAMICS SIMULATION

This appendix summarizes the physical and numerical requirements of MD simulation that MDGYM tests, describes the stages of the simulation pipeline used, and relates each requirement to the engine error categories of App. E.

A.1 EQUATIONS OF MOTION AND NUMERICAL INTEGRATION

MD simulation computes the time evolution of a system of N interacting atoms by numerically integrating the classical equations of motion. Given positions $\{\mathbf{q}_i\}_{i=1}^N$ and momenta $\{\mathbf{p}_i\}_{i=1}^N$, the dynamics follow Hamilton's equations,

$$\frac{d\mathbf{q}_i}{dt} = \frac{\partial H}{\partial \mathbf{p}_i}, \qquad \frac{d\mathbf{p}_i}{dt} = -\frac{\partial H}{\partial \mathbf{q}_i}, \tag{1}$$

where $H = \sum_i \mathbf{p}_i^2/2m_i + V(\{\mathbf{q}_i\})$ is the Hamiltonian, m_i are the atomic masses, and V is the potential energy surface. These equations are equivalent to Newton's equations of motion, with the force on each atom given by $\mathbf{F}_i = -\partial V/\partial \mathbf{q}_i$. The forces are evaluated at every step, and their evaluation dominates the computational cost of a simulation.

Integration is almost universally performed with the velocity Verlet algorithm (Verlet, 1967), which advances the system by a timestep Δt as

$$\begin{aligned} \mathbf{p}_i\big(t + \tfrac{\Delta t}{2}\big) &= \mathbf{p}_i(t) + \tfrac{\Delta t}{2}\,\mathbf{F}_i(t), \\ \mathbf{q}_i(t + \Delta t) &= \mathbf{q}_i(t) + \tfrac{\Delta t}{m_i}\,\mathbf{p}_i\big(t + \tfrac{\Delta t}{2}\big), \\ \mathbf{p}_i(t + \Delta t) &= \mathbf{p}_i\big(t + \tfrac{\Delta t}{2}\big) + \tfrac{\Delta t}{2}\,\mathbf{F}_i(t + \Delta t), \end{aligned} \tag{2}$$

where the second half-step in momentum uses the forces evaluated at the updated positions. The algorithm is symplectic and time-reversible. It therefore conserves a shadow Hamiltonian close to the true Hamiltonian and exhibits no long-term energy drift, provided Δt is sufficiently small. Without a thermostat or barostat, the integration samples the microcanonical (NVE) ensemble, in which the number of particles N, the volume V, the total energy E, and the total linear momentum are conserved. The drift of these quantities over a run is a standard diagnostic of integration quality.

The timestep, typically of the order of a femtosecond, must be a small fraction of the period of the fastest vibrational mode in the system. In atomistic biomolecular simulations, the O–H stretch near 3500 cm^{-1} has a period of about 10 fs and limits the timestep to $\Delta t \lesssim 1$ fs, unless bond constraints are imposed through algorithms such as SHAKE, RATTLE, or LINCS. When the timestep is too large, the shadow Hamiltonian departs from the true Hamiltonian and the integration becomes unstable, with the energy drifting without bound. Marginally unstable choices may not diverge visibly, but they sample incorrect regions of phase space. In MDGYM, such failures appear as the Timestep, Energy Divergence, Lost Atoms, and Constraint Failure categories in LAMMPS, and as the LINCS/Constraint and Energy Blow-up categories in GROMACS (App. E).

A.2 FORCE FIELDS

The empirical potential $V(\{\mathbf{q}_i\})$ approximates the true quantum mechanical potential energy surface as a sum of bonded and non-bonded terms. For molecular systems, the standard decomposition is

$$V = \sum_{\text{bonds}} k_b(r - r_0)^2 + \sum_{\text{angles}} k_\theta(\theta - \theta_0)^2 + \sum_{\text{dihedrals}} \sum_n \frac{V_n}{2}[1 + \cos(n\phi - \delta)] \tag{3}$$

$$+ \sum_{i<j} \left[4\varepsilon_{ij} \left(\frac{\sigma_{ij}^{12}}{r_{ij}^{12}} - \frac{\sigma_{ij}^{6}}{r_{ij}^{6}} \right) + \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}} \right], \tag{4}$$

where the parameters $\{k_b, r_0, k_\theta, \theta_0, V_n, \varepsilon_{ij}, \sigma_{ij}, q_i\}$ constitute the force field constants that are optimised to reproduce target properties either obtained from experiments or from first principle simulations. For materials systems, simpler pair potentials (Lennard-Jones, Morse) or many-body

potentials (EAM for metals, Tersoff and AIREBO for covalent systems, ReaxFF for reactive chemistry) replace the molecular mechanics decomposition. Each force field is parameterized against a specific class of systems and a specific range of thermodynamic conditions. Applying a force field outside its parameterization domain, mixing parameters from incompatible force fields, or neglecting long-range electrostatics when partial charges are significant, all produce forces that bear no meaningful relationship to the true interatomic interactions. The engine accepts these inputs without any error and integrates them providing no feedback to the user about the correctness.

A.3 Thermodynamic Ensembles and Equilibration

The basic velocity Verlet integrator samples the NVE ensemble. Most physical measurements require NVT or NPT conditions, achieved by coupling the system to a thermostat (V-rescale, Nosé-Hoover) or barostat (Parrinello-Rahman, Berendsen) with associated coupling time constants τ_T and τ_P. These must be matched to the physical relaxation timescales of the system. Aggressive coupling suppresses the fluctuations that define the ensemble and introduces artifacts in dynamical properties; weak coupling prevents equilibration within any practical simulation window. Equilibration itself proceeds in stages: energy minimization via steepest descent to remove steric clashes, followed by NVT and NPT runs of sufficient length that the relevant observables, such as potential energy, density, and pressure, have converged to stable mean values. Insufficient equilibration produces trajectories that are thermodynamically inconsistent with the intended state, without any engine-level indication that this has occurred.

A.4 Physical Validity Conditions

Beyond integration stability, a physically valid simulation must satisfy several conditions that the engine does not enforce. **System size** must be large enough that periodic images do not introduce spurious correlations; properties sensitive to long-wavelength fluctuations, such as elastic constants, transport coefficients, interfacial tensions, converge to their thermodynamic limit only when the box dimension substantially exceeds the relevant correlation length. For non-equilibrium protocols, such as uniaxial straining for elastic moduli, planar shear for viscosity, imposed thermal gradients for conductivity, the applied perturbation must lie within the **linear response regime**; at high loading or shear rates the measured response is rate-dependent and does not reflect an intrinsic material property. **Post-processing** introduces its own requirements: computing a diffusion coefficient from mean-squared displacement requires identifying the Fickian regime and fitting over an appropriate time window; extracting a glass transition temperature requires fitting two linear segments to the density-temperature curve and locating their intersection; computing a viscosity via the Green-Kubo relation requires integrating a stress autocorrelation function to convergence. The same trajectory yields different answers depending on how the analysis is performed, and the engine provides no guidance on which analysis is physically correct.

Collectively, these requirements—timestep stability, force field validity, ensemble consistency, finite-size convergence, linear response, and methodological precision in post-processing—must be satisfied simultaneously. None is enforced automatically. A simulation that violates any one of them runs to completion, produces output of the expected format, and gives no signal that anything is wrong. This is the regime that separates MD simulation from software engineering tasks, and the regime that MDGym is designed to measure.

B AI Agents for Scientific Workflows

Early systems based on LLMs demonstrated that agents could autonomously plan and execute chemical synthesis. For instance, ChemCrow (Bran et al., 2024) integrated 18 expert-designed tools with GPT 4 to automate tasks across organic synthesis, drug discovery, and materials design, while Coscientist (Boiko et al., 2023) demonstrated autonomous chemical research in a laboratory setting. More recent systems have extended this vision further along the discovery pipeline: The AI Scientist (Lu et al., 2024) demonstrated fully automated hypothesis generation, experimentation, and paper writing, and its successor (Yamada et al., 2025) produced manuscripts accepted at a peer-reviewed workshop. In the materials domain, SciToolAgent (Ding et al., 2025) automates hundreds

of scientific tools across biology, chemistry, and materials science via a knowledge-graph-driven retrieval system.

C MDGym: Technical Details

C.1 Architecture

Figure C.1 illustrates the three-layer architecture of the MDGym evaluation harness. The orchestration layer acts as the central coordinator, dispatching task prompts to the agent layer (Step 1), receiving agent outputs (Step 2), and routing them to the validator layer for automated scoring (Step 3). Each layer is governed by a well-defined abstract interface, making it independently extensible.

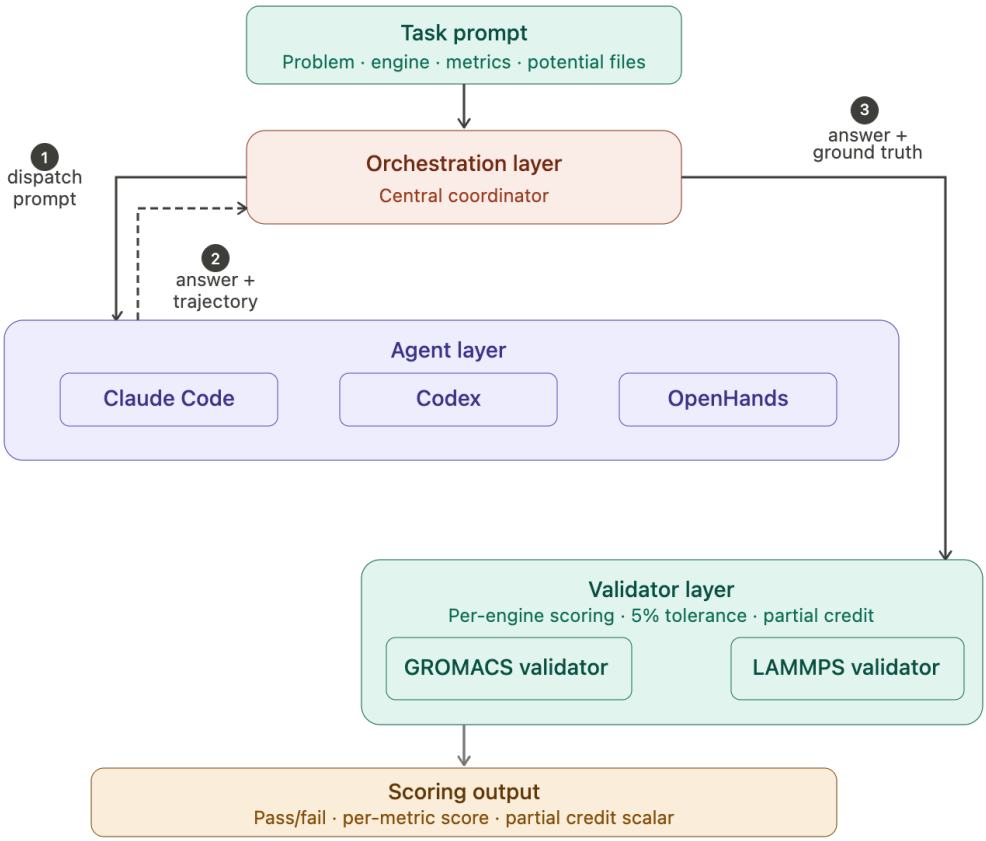


Figure C.1: Architecture of the MDGym evaluation framework, showing the three independently extensible layers — agent, orchestration, and validator — and the data flow between them.

C.1.1 Agent Layer

All agents implement a common `BaseAgent` abstract interface that exposes a single `execute(prompt, working_dir, timeout, engine)` method. Each concrete implementation wraps the corresponding CLI tool as a subprocess, streams output to a `trajectory_log.jsonl` file in the problem's working directory, and returns control to the orchestrator upon completion or timeout. Currently supported agents and their corresponding implementations are: `ClaudeCodeAgent`, `CodexAgent`, `OpenHandsAgent`. Registering a new agent

requires only subclassing `BaseAgent`, implementing the `execute` method, and adding a single entry to the agent registry via `AgentFactory.register_agent(AgentType.NEW_AGENT, NewAgentClass)`.

C.1.2 ORCHESTRATION LAYER

The `SimulationOrchestrator` coordinates a single evaluation episode. Its responsibilities are:

1. Build the structured task prompt from the problem JSON specification
2. Invoke the agent via the `BaseAgent` interface
3. Read the agent's `final_answer.json` output from the working directory
4. Pass the output and ground truth to the appropriate validator

The orchestrator is constructed via an `OrchestratorBuilder` that takes an agent type and an engine type as the only required inputs, resolving the concrete agent and validator implementations internally via their respective factories. This keeps the top-level evaluation loop entirely decoupled from the details of any specific agent or engine.

C.1.3 VALIDATOR LAYER

Each MD engine has a corresponding `BaseValidator` subclass that implements a `validate(output, working_dir, ground_truth)` method and returns a `ValidationResult` containing:

- A binary pass/fail flag
- A per-metric breakdown of which quantities passed or failed
- A scalar score in $[0, 1]$ representing the fraction of required metrics within 5% relative tolerance of the ground-truth value

Currently registered validators are `LAMMPSValidator` and `GROMACSValidator`. Adding support for a new MD engine requires subclassing `BaseValidator` and a single registration call: `ValidatorFactory.register_validator(MDEngine.AMBER, AmberValidator`.

The validator is free to incorporate engine-specific logic, such as reading auxiliary output files, applying engine-specific unit conversions, or using tighter tolerances for specific property categories, without affecting any other part of the framework.

C.1.4 SESSION MANAGEMENT AND LOGGING

In batch mode, the framework processes all problems in a session sequentially, writing per-problem results to isolated subdirectories under a session root. Each trajectory log (`trajectory_log.jsonl`) records all agent actions, tool calls, and outputs at the granularity of individual steps, and is automatically converted to a structured JSON format after each problem to facilitate offline analysis of agent reasoning and tool-use patterns. A configurable inter-problem delay mitigates API rate limiting when evaluating cloud-hosted agents. Session-level results are accumulated in a `results.csv` summary file containing per-problem scores, pass/fail flags, and metadata for downstream analysis.

C.2 JSON FORMAT

The following presents the format of the JSON file in which the tasks are prepared which is passed as input during evaluation.

Field	Description
`id`	Unique problem identifier
`problem_description`	Full natural language description of the simulation task
`level`	1 (Easy) / 2 (Medium) / 3 (Hard)
`engine`	GROMACS or LAMMPS
`metrics`	List of physical observables to compute
`ground_truth`	Expected values for each metric $\pm$ estimated error
`time_limit`	Maximum allowed simulation time in seconds

C.3 LAMMPS VS GROMACS

LAMMPS and GROMACS are both molecular dynamics simulation packages, but they target different scientific communities and use cases. GROMACS was developed primarily for biomolecular simulations and is highly optimized for proteins, nucleic acids, lipid membranes, and solvated biological systems using fixed-charge force fields like AMBER, CHARMM, GROMOS, and OPLS. Its strength lies in raw performance on standard biomolecular workflows, with aggressive SIMD vectorization and efficient GPU offloading for the kinds of homogeneous systems typical in structural biology and drug discovery. LAMMPS, by contrast, is a general-purpose particle simulator with a much broader scope: it supports atomistic, coarse-grained, mesoscale, and granular models, and ships with a wide range of interatomic potentials including reactive (ReaxFF), bond-order (Tersoff, AIREBO), embedded-atom (EAM), and machine-learning potentials. This makes it the standard tool for materials science, soft matter, polymers, solid-state systems, and any problem where the physics extends beyond classical biomolecular force fields. In short, GROMACS is the default choice when simulating biology with established force fields, while LAMMPS is preferred when flexibility across potentials, system types, or custom physics matters more than peak throughput on a narrow class of systems.

C.4 EXAMPLE TASK PROMPT

Prompt Used in MDGym to Perform an Autonomous Simulation

```
Description: Compute the average potential energy per particle for
    a system of polyethylene (hydrocarbon polymer) using the AIREBO
    potential. Model the hydrocarbon polymer using atomic units in
    the metal unit system and an atomic atom style. Construct the
    system with 60 atoms (20 C and 40 H atoms). Replicate the
    polyethylene structure 17 times along the x direction, 16 times
    along y, and 2 times along z to build a sufficiently large bulk
    system. Construct short range neighbour lists using a binning
    method with a neighbour cutoff of 0.5 Angstrom, updated every
    timestep with a delay of five steps. Model interatomic
    interactions using the AIREBO potential with a Lennard Jones
    cutoff of 3.0 Angstrom and both Lennard Jones and torsional
    interactions enabled, using appropriate carbonhydrogen
    parameterization. Assign initial velocities corresponding to a
    temperature of 300 K. Evolve the system under constant-energy
    (NVE) dynamics with a timestep of 0.0005 ps, and print
    thermodynamic quantities every 10 steps during a short
    production run of 1000 timesteps.

Required Metrics: average_potential_energy_per_particle

============================================================
IMPORTANT INSTRUCTIONS:
============================================================
0. You have a total time limit of 3450 seconds to solve this
    problem. Plan and execute efficiently within this budget.
1. Generate all necessary code, scripts, and input files for the
    simulation in LAMMPS and write them to the working directory.
2. Format code blocks with the filename for automatic extraction:
```

```
   '''language:filename
   code here...
   '''
3. Example formats for LAMMPS:
   - '''lammps:simulation.in
   - '''python:run_simulation.py
   - '''bash:run_simulation.sh
   Example formats for GROMACS:
   - '''gromacs:md.mdp
   - '''bash:run_gromacs.sh
4. The potential name is specified in the problem description. You
    don't need to download the potential file from the internet;
    you can access the potential file from the path:
    /MDGym/data/potentials/36_airebo/.
5. You can access the structure file from the path:
    /MDGym/data/structures/36_airebo/. If the structure file is not
    present in that directory, you can make make your own structure
    file based on the information provided in the problem
    description.
6. You don't need to check the compatibility of the environment or
    install the MD engine; always assume that the environment is
    pre-configured with the necessary software and dependencies to
    run the simulations. Start solving the problem right away as we
    have a time limit to solve the problem.
7. Run the code and perform any required postprocessing to obtain
    the final answer.
8. Report the quantities asked in the problem description in JSON
    format, where the key is the name of the quantity and the value
    is its numerical value without units.
9. Pay careful attention to units. All required units are
    specified in the problem description, and the final answer must
    use those exact units.
10. Your final response must be ONLY a raw JSON object---no
    markdown, no code fences, no backticks, no explanation, and no
    preamble. The response must start with { and end with }.
11. Write the final JSON object to final_answer.json for automatic
    extraction.
12. These are the only instructions. There in no AGENTS.md or such
    files.
```

C.5 DATASET DETAILS

The input files used for the GROMACS are taken from multiple sources including original research papers, repositories associated tutorials or examples for these packages, or created from scratch. Specifically, 14 files from GROMACS repository were taken and modified (with different material, ensemble etc.), 1 file each from Rajput et al. (2023); Koch et al. (2025), 3 files from Lemkul (2018), and 6 files from Gravelle et al. (2025), and the remaining files are developed by the author. The simulation workflow in GROMACS typically follow these steps. First, system is energy minimized using the steepest-descent algorithm. Second, equilibration is performed under NVT and/or NPT ensembles. Finally, production runs are carried out in the NPT ensemble. All observables are computed through post-processing of data obtained from either the equilibration or production runs.

For the LAMMPS tasks, 73 input files are taken from the official LAMMPS repository Thompson et al. (2022), 9 files from Gravelle et al. (2025), and the remaining files are taken from the *Mississippi State University* tutorial webpage. All these were subsequently modified with a different material, simulation ensemble or other conditions such as loading or deformation. The simulation workflow in LAMMPS follows a similar protocol unless otherwise specified in the problem description. First, the system is initialized using the provided input parameters. Second, the force field is defined. Third, energy minimization is performed, followed by velocity initialization and equilibration. Finally, production runs are carried out. Unless stated otherwise, all observables are computed by post-processing the production trajectories.

Table C.1: Distribution of MDGYM simulation tasks across difficulty levels.

Engine	Easy	Medium	hard
LAMMPS	31	40	30
GROMACS	33	30	36
Total	**64**	**70**	**66**

C.6 PROPERTY CATEGORIES

MDGYM covers a broad range of physical quantities across six property categories, reflecting the diversity of scientific questions that MD simulation is used to answer. Table C.2 summarises the distribution of metrics across categories, and Table C.3 lists the most frequently occurring individual metrics in the benchmark.

The benchmark is dominated by thermodynamic properties, which appear in the largest number of problems — average temperature, average density, and average potential energy per particle are among the most common individual metrics, reflecting their centrality to standard MD workflows. Structural properties constitute the second-largest category, with coordination number and average radius of gyration being the most frequent. Mechanical properties constitute the most diverse category by number of distinct metrics, covering quantities such as Young's modulus, shear modulus, Poisson ratio, and vacancy formation energy that reflect the breadth of materials science applications targeted by LAMMPS tasks. Transport properties (diffusion coefficient, viscosity, thermal conductivity, surface tension) and magnetic properties are represented by a smaller number of problems, providing coverage of more specialised simulation workflows. A small set of numerical validation metrics, such as force relative error and Born matrix relative error, is also included to test whether agents correctly implement numerical verification procedures.

Table C.2: Distribution of output metrics across property categories in MDGYM.

Property Category	Distinct Metrics	Total Occurrences
Thermodynamic	24	211
Mechanical	24	94
Structural	13	41
Transport	6	6
Magnetic	2	3
Other/Numerical	7	11
Total	**76**	**366**

Table C.3: Most frequently occurring output metrics in MDGYM.

Metric	Category	Problems
Average temperature	Thermodynamic	114
Coordination number	Structural	49
Average density	Thermodynamic	41
Average radius of gyration	Structural	16
Average potential energy per particle	Thermodynamic	13
Average total energy	Thermodynamic	11
Average total energy per particle	Thermodynamic	9
Young's modulus	Mechanical	8
Average end-to-end distance	Structural	7
Solvent Accessible Surface Area	Structural	6

C.7 MD SIMULATION DETAILS

Molecular dynamics (MD) simulations were carried out using two widely used MD packages: GROMACS and LAMMPS. Specifically, GROMACS v2026.0 package and LAMMPS (22 Jul 2025 version) were used to perform the simulations. To enable simulation execution through LLM models, we provided the initial system structures along with the corresponding topological parameters.

D AGENT FRAMEWORK ARCHITECTURES

To characterize precisely what kind of system each evaluated framework is, this section documents the publicly available architectural details of Claude Code, Codex, and OpenHands along four criteria relevant to general-purpose agentic capability: system prompt structure, state management, tool descriptions, and error recovery mechanisms. We report only what is documented or inspectable in each framework's public materials, without interpretation.

D.1 CLAUDE CODE

Claude Code's documented architecture is an agentic loop of three phases — gather context, take action, verify results — in which the model determines the requirements of each step from the outcome of the previous one, chaining actions and course-correcting as needed[3]. The official documentation states explicitly: *"While it excels at coding, it can help with anything you can do from the command line: writing docs, running builds, searching files, researching topics, and more."* State is persisted via a project-level `CLAUDE.md` file and auto-loaded memory. The framework additionally exposes a public glossary defining its *agentic harness* — "the tools, context management, and execution environment that turn a language model into a capable coding agent" — as distinct from the underlying model.

D.2 CODEX

Codex's documented architecture centers on a shared *harness* — the core agent loop and execution logic underlying Codex CLI, Codex Cloud, and the Codex VS Code extension. The loop follows a standard inference-then-tool-call pattern: user input is incorporated into a prompt sent to the Responses API; the model either produces a final assistant message or requests a tool call, which the harness executes and appends to the prompt before re-querying the model; this repeats until an assistant message terminates the turn. The default tool set includes a `shell` tool for spawning local processes, a built-in `update_plan` tool, a `web_search` tool provided by the Responses API, and user-configured MCP server tools. State and task-specific instructions are aggregated at prompt-construction time from `AGENTS.override.md` and `AGENTS.md` files found from the project root down to the working directory, analogous to Claude Code's `CLAUDE.md`. Codex additionally supports *skills*, which OpenAI describes as extending Codex "beyond code generation to tasks that require gathering and synthesizing information, problem-solving, writing, and more"[4]. We note, however, that OpenAI's own product framing of Codex is more coding-centric than Claude Code's: OpenAI's developer guidance recommends general-purpose models for non-coding work and positions "the Codex surface" specifically for tasks involving "file edits, shell commands, reviews, and sandboxes", and third-party summaries describe Codex as "an AI coding agent... for software engineering tasks."

D.3 OPENHANDS

OpenHands' system prompt is assembled from a typed, open-source section registry comprising sixteen sections in its base configuration: ROLE, MEMORY, EFFICIENCY, FILE_SYSTEM_GUIDELINES, CODE_QUALITY, VERSION_CONTROL, PULL_REQUESTS, PROBLEM_SOLVING_WORKFLOW, SELF_DOCUMENTATION, SECURITY, SECURITY_RISK_ASSESSMENT, EXTERNAL_SERVICES, ENVIRONMENT_SETUP, TROUBLESHOOTING, PROCESS_MANAGEMENT, and an optional model-specific IMPORTANT

[3] `https://code.claude.com/docs/en/how-claude-code-works`

[4] `https://openai.com/index/introducing-the-codex-app/`

section, followed by dynamic per-conversation content. The default tool set consists of `TerminalTool`, `FileEditorTool`, `TaskTrackerTool`, and `BrowserToolSet` (disabled in CLI mode), with `FinishTool` and `ThinkTool` attached automatically as additional built-ins[5].

Of the sixteen base sections, three — CODE_QUALITY, VERSION_CONTROL, and PULL_REQUESTS — carry explicit software-engineering framing. We report this for transparency: these sections are present in the default configuration and were not disabled during evaluation. We also note that OpenHands' own project documentation self-describes with an explicit coding emphasis: its repository README states *"Emphasis on coding: Purpose-built for software development tasks with specialized tools and workflows"*. Of the three frameworks evaluated, this is the most explicitly coding-scoped self-description; Claude Code's documentation makes the opposite claim (general-purpose, coding as one use case among several), and Codex's positioning falls between the two, offering general-purpose extension via skills while being marketed primarily as a coding surface.

D.4 Cross-Framework Comparison

Table D.4 summarizes how each of the four architectural criteria is instantiated across the evaluated harnesses.

Table D.4: Architectural components of evaluated agent frameworks, mapped to four criteria relevant to general-purpose agentic capability.

Criterion	Instantiation in evaluated harnesses
System prompts	Typed section registry (OpenHands); model-specific bundled instructions read from `config.toml` or built-in defaults (Codex); documented three-phase agentic loop (Claude Code)
State management	MEMORY section and `TaskTrackerTool` (OpenHands); `AGENTS.md`/`AGENTS.override.md` aggregation (Codex); `CLAUDE.md` and auto-loaded memory (Claude Code)
Tool descriptions	JSON-schema function-call tools (`shell`, `update_plan`, `web_search`, MCP) (Codex); typed tool classes (`TerminalTool`, `FileEditorTool`, etc.) (OpenHands); documented built-in tool set (Claude Code)
Error recovery loops	TROUBLESHOOTING and PROBLEM_SOLVING_WORKFLOW sections (OpenHands); the verify phase of the agentic loop (Claude Code); iterative tool-call/re-inference cycle with automatic context compaction (Codex)

D.5 Discussion

We report this distribution and self-description transparently rather than omitting it, and note that it applies unevenly across the three frameworks: Claude Code's documentation makes the strongest explicit claim to general-purpose, non-coding capability, while OpenHands' own materials make the strongest explicit claim to coding-specific purpose-building. This observation does not, on its own, establish that MDGym performance is measuring software-engineering-specific priors rather than general agentic capability. The architectural machinery common to all three evaluated frameworks — a terminal, a file editor, a task-tracking mechanism, a documented troubleshooting or verification protocol, and state persisted via a project-level instructions file (`CLAUDE.md` / `AGENTS.md` / `MEMORY.md`) — is precisely the machinery required to operate any MD engine, since MD workflows are executed through command-line binaries acting on text-based input decks.

E Error Analysis Methodology

E.1 Trajectory-Level Error Taxonomy

The trajectory-level error taxonomy was developed through iterative manual inspection of failed agent trajectories. We began by reviewing a random sample of failed runs across all four agents,

[5] `https://github.com/OpenHands/software-agent-sdk`

identifying recurring patterns in how agents fail. This process yielded five mutually non-exclusive error categories — a trajectory may exhibit multiple error types simultaneously. For each category, we define both the operational definition and the rule used for automated classification.

Fabricated Answer. The agent reports numerical output values in `final_answer.json` without having executed the underlying simulation, producing plausible-looking but ungrounded results.
Detection rule: The agent produced an answer but either no simulation output files exist in the agent's working directory or the simulation log contains error lines, indicating the reported values were not derived by executing a simulation.

Syntax Error. The agent produces malformed input scripts that prevent the simulation from executing.
Detection rule: A failed agent action where the action output contains a syntax-indicating keyword such as `syntax error`, `unknown command`, `invalid`, `illegal`, `unrecognized`, or `fatal error` alongside an engine-specific marker such as `lmp`, `.in`, `gmx`, `.mdp`, or `.top`.

Environment Misunderstanding. The agent fails to correctly interpret the task setup, available tools, or the pre-configured simulation environment, leading to actions that are irrelevant or counterproductive to the task.
Detection rule: A failed action whose output contains `command not found`.

Premature Termination. The agent abandons the task upon encountering domain-specific error messages, failing to attempt iterative debugging or recovery.
Detection rule: The agent produced no answer and the run did not exhaust the time budget. Timed-out runs are excluded, as failing to produce an answer under a binding time constraint is expected behaviour rather than premature Termination.

Incorrect Post-Processing. The agent executes the simulation but applies an incorrect analysis procedure to extract the required output metric, producing a wrong final value despite a physically valid simulation.
Detection rule: The agent produced an answer, and the simulation ran without errors, ruling out fabrication, but the trajectory still failed.

E.2 Simulation-Level Error Classification

The simulation-level error analysis characterizes the specific errors produced by the MD engines during failed simulation runs, providing a finer-grained view of where and how simulations break down. Error categories for each engine were predefined by domain experts based on the known failure modes of LAMMPS and GROMACS:

LAMMPS categories Command Syntax, Variable/Expression, Pair/Force Field, Data File and Read, Fix/Compute Incompatibility, Thermo and Output, Unit and Dimension, Region and Geometry, Package/Partition, Lost Atoms, Energy Divergence, Neighbor List, Timestep, Script Logic Order, Boundary Condition, Thermostat/Barostat, MPI/Parallel Decomposition, and Constraint Failure.

GROMACS categories MDP Parameter, Integrator/Thermostat Incompatibility, Topology, Index Group, Coord/Topology Mismatch, Position Restraint, LINCS/Constraint, Energy Blow-up, Periodic Boundary/Box Error, PME/Electrostatics, MPI/GPU Parallelisation, Force Field Param Missing, Velocity/Temperature Initialisation, Pressure Coupling/Barostat, and Output/File Write.

For each failed simulation run, engine error logs were extracted from the trajectory and provided to `gpt-5.4-2026-03-05` along with the predefined category list and their definitions. To improve robustness, three independent classifications were generated per error and the majority label was taken as the final category. The system prompts used for LAMMPS and GROMACS classification are provided below.

E.3 LLM CLASSIFIER PROMPTS

E.3.1 LAMMPS ERROR CLASSIFIER PROMPT

LAMMPS Error Classifier System Prompt

```
You are a LAMMPS simulation error classifier. You will be given a
list of numbered error messages extracted from a LAMMPS
trajectory. For each error, classify it into exactly one category
from the taxonomy below.

=== SYNTAX / STATIC ERRORS (S1-S9) ===
These are detected during input script parsing before any timestep
runs.

--- S1: Command Syntax Error ---
Definition: A misspelled LAMMPS command name, incorrect number of
arguments, arguments in the wrong order, or use of a keyword that
does not exist in the installed build. Detected during input
script parsing before any timestep runs.
Trigger: Parser fails to recognise the command token or cannot
match the argument list to any known signature for that command.
Signals: 'Unknown command', 'Illegal command', or 'Expected N
arguments'; error line references the exact input script line;
error occurs before timestep 0.
Classification rule: Classify as S1 if ALL criteria hold: (1)
LAMMPS reports 'Unknown command', 'Illegal command', 'Expected N
arguments', or 'Unrecognized keyword'; (2) the error is raised
during input script parsing before the run starts; (3) the message
cites a specific script line.
Examples: pair_style lj/cut 2.5 extra_arg # too many args;
atom_sytle atomic # typo in command name

--- S2: Variable & Expression Error ---
Definition: Use of an undefined variable, an invalid mathematical
expression inside a variable definition, or the wrong variable
style for a context.
Trigger: LAMMPS tries to evaluate a variable or expression and
cannot resolve the reference or parse the math.
Signals: 'Variable ... is not defined'; 'Invalid math expression';
'Variable ... has wrong style'.
Classification rule: Classify as S2 if ALL criteria hold: (1) the
message contains 'variable', 'undefined variable', 'invalid
expression', or 'wrong style'; (2) the failure occurs during
variable evaluation or preprocessing; (3) no atom positions or
forces are involved.

--- S3: Pair / Force Field Error ---
Definition: Mismatch between pair_style and pair_coeff
declarations: missing coefficients for one or more atom-type
pairs, wrong number of parameters, inconsistent mixing rules, or
use of a pair style not compiled into the LAMMPS binary.
Signals: 'No pair style is defined'; 'Pair coeffs are not set';
'Incorrect args for pair_coeff'.
Classification rule: Classify as S3 if ALL criteria hold: (1) the
message references 'pair_style', 'pair_coeff', 'pair coeffs', or
'mixing'; (2) the error appears before or at the start of the run;
(3) it does not reference lost atoms or energy divergence.

--- S4: Data File & Read Error ---
Definition: Malformed LAMMPS data file: atom count in the header
does not match the Atoms section, a required section is absent,
```

```
atom type indices are out of range, or the file format does not
match the declared atom_style.
Signals: 'Did not assign all atoms correctly'; 'Unexpected end of
data file'; 'Too many/few lines in ... section'; 'Invalid atom
type'.
Classification rule: Classify as S4 if ALL criteria hold: (1) the
error is raised inside or immediately after read_data; (2) it
references atom counts, sections, or file structure; (3) it does
not reference pair coefficients or force constants.

--- S5: Fix / Compute Incompatibility ---
Definition: A fix or compute applied to a group that does not
exist, references a compute/fix ID not yet defined, or has invalid
arguments.
Signals: 'Group ID does not exist'; 'Fix requires ... atom style';
'Compute ID ... does not exist'.
Classification rule: Classify as S5 if ALL criteria hold: (1) the
message names a fix or compute ID; (2) it references a missing
group, missing dependency, or incompatible atom style; (3) the
error is raised before the first timestep.

--- S6: Thermo & Output Error ---
Definition: Error in thermo_style, dump, or restart configuration:
referencing a compute or fix keyword not defined, or mismatching
dump format arguments.
Signals: 'Unknown keyword in thermo_style'; 'Invalid keyword in
dump ... command'.
Classification rule: Classify as S6 if ALL criteria hold: (1) the
message involves thermo_style, dump, or restart; (2) it references
an unrecognised or undefined keyword/ID; (3) it is not a runtime
energy or force error.

--- S7: Unit & Dimension Error ---
Definition: Commands that depend on a unit system issued before
the units command is set, or 2D-specific commands used without
declaring dimension 2.
Signals: 'Cannot use ... before units are set'; 'Must use
dimension 2 before ...'.
Classification rule: Classify as S7 if ALL criteria hold: (1) the
message references 'units', 'dimension', or a unit-dependent
quantity; (2) the error appears before the run starts; (3) it does
not concern atom types or pair coefficients.

--- S8: Region & Geometry Error ---
Definition: Wrong keyword for a region shape type, incorrect
number of geometric parameters, invalid simulation box dimensions,
or create_atoms referencing an undefined region.
Signals: 'Illegal region command'; 'Invalid region ID'; 'Box
bounds are invalid'.
Classification rule: Classify as S8 if ALL criteria hold: (1) the
message references 'region', 'box', or 'create_atoms'; (2) the
error involves geometric parameters or undefined region IDs; (3)
it occurs before the run.

--- S9: Package / Partition Error ---
Definition: Requesting a GPU, KOKKOS, INTEL, or OMP accelerated
style not compiled into the LAMMPS binary, or supplying invalid
arguments to the package command.
Signals: 'Package gpu is not available'; 'Unrecognized package
name'; style suffix /gpu or /kk present but build lacks that
suffix.
Classification rule: Classify as S9 if ALL criteria hold: (1) the
message references 'package', a style suffix (/gpu, /kk, /intel,
```

```
/omp), or 'partition'; (2) it states the feature is unavailable or
the command is illegal; (3) it occurs at startup before setup.

=== RUNTIME / LOGICAL ERRORS (R1-R9) ===
These occur during a run at timestep > 0.

--- R1: Lost Atoms Error ---
Signals: 'Lost atoms: ...' during the run; 'Atoms lost in xxx
timesteps'.
Classification rule: Classify as R1 if ALL criteria hold: (1) the
message contains 'lost atoms' or 'atoms lost'; (2) the error is
raised during a run (timestep > 0); (3) a count of lost atoms or a
timestep number is reported.

--- R2: Energy Divergence Error ---
Signals: Energy printed as NaN or Inf in thermo output; 'Numerical
instability' warning.
Classification rule: Classify as R2 if ALL criteria hold: (1)
thermo output shows NaN, Inf, or astronomically large energy
values; (2) the error occurs during a run (timestep > 0); (3) no
'lost atoms' message appears before the NaN.

--- R3: Neighbor List Error ---
Signals: 'Dangerous builds' counted in output; 'Neighbor list
overflow'.
Classification rule: Classify as R3 if ALL criteria hold: (1) the
message references 'neighbor', 'neigh', 'skin', or 'dangerous
builds'; (2) it appears during setup or at a rebuild step; (3) it
does not report lost atoms.

--- R4: Timestep Error ---
Signals: Energy divergence or lost atoms within the first ~50
steps; extremely high temperature in the first few steps.
Classification rule: Classify as R4 if ALL criteria hold: (1) the
instability begins within the first ~50 steps; (2) decreasing the
timestep resolves the issue; (3) there is no obvious initial
geometry overlap.

--- R5: Ordering / Script Logic Error ---
Signals: 'Pair style not yet defined'; 'Must define atom_style
before read_data'.
Classification rule: Classify as R5 if ALL criteria hold: (1) the
message says a required precondition has not been set; (2)
re-ordering the commands would fix the error; (3) no atom
positions or forces are involved.

--- R6: Boundary Condition Error ---
Signals: 'Pair style requires periodic boundary in all
dimensions'; lost atoms attributed to non-periodic boundary.
Classification rule: Classify as R6 if ALL criteria hold: (1) the
message references 'boundary', 'periodic', 'shrink-wrap', or
'fixed wall'; (2) the pair style or ensemble conflicts with the
boundary setting; (3) the error does not arise from atom geometry
or overlaps.

--- R7: Thermostat / Barostat Error ---
Signals: 'Fix npt requires ...'; pressure or temperature
oscillating wildly.
Classification rule: Classify as R7 if ALL criteria hold: (1) the
message references a thermostat or barostat fix; (2) the error
involves incompatible settings or T/P instability; (3) it is not a
lost-atom or energy-divergence error.
```

```
--- R8: MPI / Parallel Decomposition Error ---
Signals: 'Too many processors for simulation'; 'Processor grid
does not match domain'.
Classification rule: Classify as R8 if ALL criteria hold: (1) the
message references 'processors', 'MPI', 'domain', or 'ranks'; (2)
the error is triggered by the number of MPI processes or processor
grid; (3) it does not involve pair coefficients or atom types.

--- R9: Constraint Failure ---
Signals: 'SHAKE atoms missing'; 'RATTLE failed to converge';
'Linear constraint solver did not converge'.
Classification rule: Classify as R9 if ALL criteria hold: (1) the
message contains 'SHAKE', 'RATTLE', 'LINCS', or 'constraint'; (2)
it references convergence failure or missing atoms; (3) it is a
runtime error (timestep > 0).

=== INSTRUCTIONS ===
For each numbered error below, return exactly one category ID
from: S1 S2 S3 S4 S5 S6 S7 S8 S9 R1 R2 R3 R4 R5 R6 R7 R8 R9 or
"Unclassified".

Respond with a valid JSON object only - no markdown, no
explanation:
{
  "classifications": [
    {"index": 1, "category_id": "S3", "category_name": "Pair /
    Force Field Error", "category_type": "Syntax"},
    ...
  ]
}
```

E.3.2 GROMACS ERROR CLASSIFIER PROMPT

GROMACS Error Classifier System Prompt

```
You are a GROMACS simulation error classifier. You will be given a
list of numbered error messages extracted from a GROMACS
trajectory. For each error, classify it into exactly one category
from the taxonomy below.

=== PREPROCESSING / STATIC ERRORS (P1-P6) ===
These are detected by grompp before the simulation starts.

--- P1: MDP Parameter Error ---
Definition: An unrecognised, misspelled, or deprecated parameter
key in the .mdp run-input file, or a value out of the allowed
range.
Signals: Warning: Unknown left-hand 'xyz' in parameter file; Fatal
error: Invalid value for mdp option.
Classification rule: Classify as P1 if ALL criteria hold: (1) the
message references an .mdp file option by name; (2) it says the
option is 'unknown', 'invalid', 'deprecated', or out of range; (3)
the error is raised by grompp before the simulation starts.

--- P2: Integrator / Thermostat Incompatibility ---
Definition: An incompatible combination of integrator and
thermostat or barostat in the .mdp file.
Signals: Fatal error: Integrator ... is not compatible with tcoupl
...; Fatal error: Cannot use ... with integrator ...
```

```
Classification rule: Classify as P2 if ALL criteria hold: (1) the
message names both an integrator and a thermostat or barostat; (2)
it says they are incompatible; (3) the error is raised by grompp.

--- P3: Topology Error ---
Definition: An error in the .top or .itp topology file: broken
#include chain, mismatched moleculetype declarations, wrong atom
count, undefined atom types, or missing force field files.
Signals: Fatal error: No such file or directory (for included
.itp); Fatal error: number of atoms in ... does not match.
Classification rule: Classify as P3 if ALL criteria hold: (1) the
message references a .top or .itp file; (2) it reports a missing
include, undefined atom type, or atom count mismatch; (3) it is
raised by grompp.

--- P4: Index Group Error ---
Definition: A group name referenced in the .mdp file does not
exist in the .ndx index file, or the index file is malformed or
missing.
Signals: Fatal error: Group ... not found in index file; Error
reading index file.
Classification rule: Classify as P4 if ALL criteria hold: (1) the
message references a group name or index file; (2) it says the
group is 'not found' or the index file is invalid/missing; (3) it
is raised by grompp.

--- P5: Coordinate / Topology Mismatch ---
Definition: The number or order of atoms in the coordinate file
does not match what the topology expects.
Signals: Fatal error: number of coordinates in ... does not match
topology; Fatal error: Atom ... not found in residue.
Classification rule: Classify as P5 if ALL criteria hold: (1) the
message compares a coordinate file to the topology; (2) it reports
a count or naming mismatch; (3) it is raised by grompp.

--- P6: Position Restraint Error ---
Definition: The position restraint file does not match the current
topology: wrong number of restrained atoms, mismatched atom
indices, or the posre.itp #include is missing.
Signals: Fatal error: Position restraint file ... has ... entries,
expected ...; Fatal error: Atom index ... in position restraints
is out of range.
Classification rule: Classify as P6 if ALL criteria hold: (1) the
message references a position restraint file or the posres
directive; (2) it reports a count or index mismatch; (3) it is
raised by grompp.

=== RUNTIME / LOGICAL ERRORS (R1-R9) ===
These occur during mdrun at step >= 0.

--- R1: LINCS / Constraint Failure ---
Signals: LINCS WARNING: ... constraints not satisfied; Fatal
error: Too many LINCS warnings.
Classification rule: Classify as R1 if ALL criteria hold: (1) the
message contains 'LINCS', 'SETTLE', or 'constraint'; (2) it
references non-convergence or deviation during the run; (3) it
occurs at a specific timestep (step > 0).

--- R2: Energy Blow-up / Numerical Instability ---
Signals: Fatal error: Nan or Inf in coordinates or velocities;
Step ...: NAN detected in ...
Classification rule: Classify as R2 if ALL criteria hold: (1) the
message contains 'NaN', 'Inf', 'blow-up', or 'not converging'; (2)
```

```
it is raised during an mdrun step (step > 0); (3) no LINCS warning
appears before it.

--- R3: Periodic Boundary / Box Error ---
Signals: Fatal error: The box is too small for the largest
cut-off; Fatal error: Atom ... is outside the box.
Classification rule: Classify as R3 if ALL criteria hold: (1) the
message references the simulation box, cutoff, or PBC; (2) it
reports atoms outside the box or box too small; (3) it is a
runtime error (step >= 0).

--- R4: PME / Electrostatics Error ---
Signals: Fatal error: The PME grid ... is not compatible with ...;
Fatal error: Cannot use PME with non-periodic boundary.
Classification rule: Classify as R4 if ALL criteria hold: (1) the
message references 'PME', 'Ewald', 'grid', or 'electrostatics';
(2) it reports an incompatibility or initialisation failure; (3)
it does not reference LINCS or NaN.

--- R5: MPI / GPU Parallelisation Error ---
Signals: Fatal error: Too few ranks ... for domain decomposition;
Fatal error: Cannot use GPU with ...
Classification rule: Classify as R5 if ALL criteria hold: (1) the
message references 'ranks', 'domain decomposition', 'GPU', or
'MPI'; (2) it reports an incompatibility or setup failure; (3) it
occurs at initialisation (before step 1).

--- R6: Force Field Parameter Missing ---
Signals: Fatal error: No parameters for ... in ...; Fatal error:
No default ... parameters.
Classification rule: Classify as R6 if ALL criteria hold: (1) the
message says 'no parameters', 'no default', or 'not found' for a
bonded or non-bonded interaction; (2) it is raised during mdrun
(step >= 0); (3) it is not a topology #include or atom-count error.

--- R7: Velocity / Temperature Initialisation Error ---
Signals: Fatal error: gen-temp must be > 0; WARNING: Velocities
were not generated.
Classification rule: Classify as R7 if ALL criteria hold: (1) the
message references 'gen-vel', 'gen-temp', 'velocities', or
temperature initialisation; (2) it reports an invalid temperature
or velocity incompatibility; (3) it occurs at startup (step 0 or
grompp).

--- R8: Pressure Coupling / Barostat Error ---
Signals: WARNING: Step ..., time ...: the box is growing/shrinking
rapidly; Fatal error: Box has collapsed.
Classification rule: Classify as R8 if ALL criteria hold: (1) the
message references 'pressure coupling', 'barostat', 'box vectors',
or 'compressibility'; (2) it reports rapid box change, collapse,
or incompatibility; (3) it is a runtime error (step > 0).

--- R9: Output / File Write Error ---
Signals: Fatal error: Cannot write to file ...; Fatal error:
Incompatible checkpoint file.
Classification rule: Classify as R9 if ALL criteria hold: (1) the
message references a file path or file type (.trr, .xtc, .edr,
.cpt); (2) it says the file cannot be written, is corrupt, or is
incompatible; (3) it is raised during mdrun output.

=== INSTRUCTIONS ===
```

```
For each numbered error below, return exactly one category ID
from: P1 P2 P3 P4 P5 P6 R1 R2 R3 R4 R5 R6 R7 R8 R9 or
"Unclassified".

Respond with a valid JSON object only - no markdown, no
explanation:
{
  "classifications": [
    {"index": 1, "category_id": "P3", "category_name": "Topology
    Error", "category_type": "Preprocessing"},
    ...
  ]
}
```

Simulation Stage Completion Classifier System Prompt

```
You are an expert computational materials scientist and an
automated diagnostic engine for molecular dynamics (MD)
simulations.

Your task is to analyze the provided simulation log file (which
may be from LAMMPS, GROMACS, or similar engines) and determine the
exact stage at which the simulation ended, whether it completed
successfully, or if it failed, what type of error caused the
failure.

### The 4 Stages of MD Simulation (Strict Definitions)
Use the following explicit criteria to classify the progress of
the simulation. Do not guess; rely on these strict physical and
textual signatures:

1. INITIALIZATION (System Setup)
- Definition: The engine is translating inputs into a mathematical
state prior to any particle displacement. Time is strictly at t=0.
This includes parsing topology, applying forcefield parameters
(charges, LJ parameters), building initial neighbor lists, and
allocating memory.
- Success Signatures: The log explicitly prints the simulation box
dimensions, total number of atoms/interactions, and successfully
evaluates step 0 (or pre-run) energies without throwing "NaN" (Not
a Number) or infinite force errors.
- Exclusion: If the engine crashes before building the neighbor
list or printing step 0 thermals, Initialization has FAILED.

2. ENERGY MINIMIZATION (Static Relaxation)
- Definition: A purely mathematical optimization to find a local
potential energy minimum and resolve steric clashes. There is no
concept of time, temperature, or kinetic energy in this stage.
Algorithms used are typically Steepest Descent (SD) or Conjugate
Gradient (CG).
- Success Signatures: The log explicitly reports "Minimization
converged", the maximum force (F_max) drops below the target
tolerance (e.g., ftol), or the maximum requested iterations are
safely reached.
- Exclusion: If atoms move "too far" or forces become infinite
here, it is a Minimization failure.

3. THERMODYNAMIC EQUILIBRATION (NVT / NPT)
- Definition: Time-integration begins (Newton's equations are
solved), but the primary goal is coupling the system to a heat
```

```
bath (thermostat) or pressure bath (barostat) to reach a target
macrostate. Trajectory data here is considered "burn-in" and not
meant for final analysis.
- Success Signatures: Completion of the specified NVT/NPT `run`
segments. The log must show rolling averages of Temperature,
Pressure, and Density oscillating steadily around their target
values (e.g., T ~ 300K) with no exponential drift or sudden spikes.
- Exclusion: If a `run` command finishes, but the final
temperature is thousands of degrees away from the target,
Equilibration has FAILED due to physical blow-up, even if the
engine didn't explicitly crash.

4. PRODUCTION RUN (Data Acquisition)
- Definition: The final, stable integration phase where the
thermodynamic ensemble is maintained, and coordinates/velocities
are actively dumped to trajectory files for scientific analysis.
- Success Signatures: The final `run` command completes entirely.
The log prints the ultimate timing summary (e.g., "Total wall
time", "Performance: ns/day", "Loop time of X on Y procs"),
signaling a clean exit of the simulation loop.
- Exclusion: Any crash, lost atom, or constraint failure occurring
after equilibration completes is a Production failure.

### Task Instructions
1. Scan the log file sequentially from top to bottom.
2. Identify the highest stage that completed successfully. If a
stage started but crashed before finishing, the *previous*
completed stage is the last successful one. If it crashes during
Initialization, the last successful stage is "None".
3. If the simulation crashed, classify the failure using one of
these categories: [Syntax/Parsing, Topology/Parameterization,
Step-Zero Instability, Integration/Dynamics Instability,
Hardware/Parallelization].
4. Extract the specific error message or the line right before the
failure.

### Output Format
You must output your analysis as a valid JSON object using the
exact schema below. Do not include any markdown formatting or
conversational text outside of the JSON block.

{
  "last_successful_stage": "None" | "Initialization" |
  "Minimization" | "Equilibration" | "Production",
  "error_category": "String (from the categories listed above, or
  null if Success)",
  "error_evidence": "String (the exact error message or last
  printed line indicating the crash, or null if Success)",
  "reasoning": "A brief, 1-2 sentence explanation justifying the
  classification."
}
```

E.4 Agent Behavior Modes

During manual inspection of agent trajectories, we observed recurring behavioral patterns across agents. We document these here as qualitative observations to complement the automated error analysis.

Plan Then Execute. The agent produces substantial reasoning about the problem — identifying required parameters, decomposing sub-tasks, and anticipating potential pitfalls before writing any simulation files.

Exploratory Probing. The agent begins by exploring the environment, for example, listing files, checking directory contents, reading existing structure or potential files, before writing any simulation code.

Code Generation. The agent generates a simulation input script, either writes the first file immediately after reading the problem or produces all required files in a single batch before executing any of them.

Iterative Refinement. The agent writes a simulation file, executes it, encounters an error, edits the file, and retries — repeating this write-run-debug cycle. The same core approach is preserved and incrementally corrected rather than abandoned.

Trial and Error. The agent makes multiple substantially different attempts to solve the problem, abandoning one approach for another when it fails. Distinguished from Iterative Refinement by the scope of change between attempts: the agent rewrites the simulation strategy rather than patching a specific error.

Verification. The agent explicitly checks whether its simulation output is correct before submitting a final answer, including parsing output logs to extract specific metric values, cross-referencing computed results against expected physical ranges, and performing unit conversions on extracted values.

F Additional error analysis

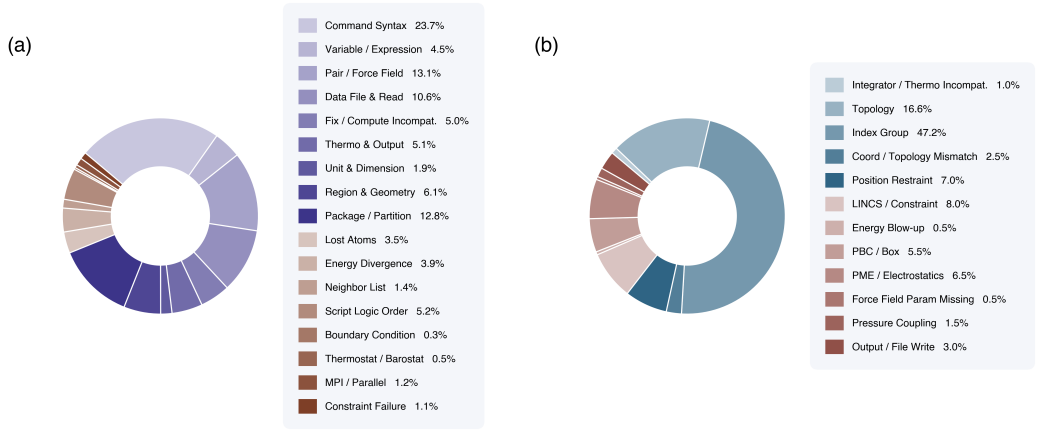


Figure F.2: Distribution of engine errors raised within failed trajectories, pooled over all configurations, for (a) LAMMPS and (b) GROMACS. Categories follow App. E.

G Performance of Agents on Different Simulation Packages

H Performance of all model–framework combinations

To separate the contributions of the model and the agent framework, we evaluate every combination of eight LLMs and the three frameworks on a common subset of 50 problems sampled uniformly at random from the Medium and Hard levels of MDGYM (41 LAMMPS and 9 GROMACS problems). Besides the four models of the main evaluation, the subset includes Claude Sonnet 4.6, GPT 5.2 Codex, GPT OSS 20B, and Qwen3 Coder 30B. Figure H.8 reports the full-success rate of each combination.

Consistent with the main results, we observe that the model determines performance far more strongly than the framework. Claude Opus 4.8 and GPT 5.5 solve 10–16% of the problems under every framework, whereas GPT OSS 20B and Qwen3 Coder 30B solve none under any. Within each model family, the stronger model performs at least as well under every framework: Claude Opus 4.8 over Claude Sonnet 4.6, GPT 5.5 over GPT 5.2-Codex, and Qwen3.6-27B over Qwen3 Coder

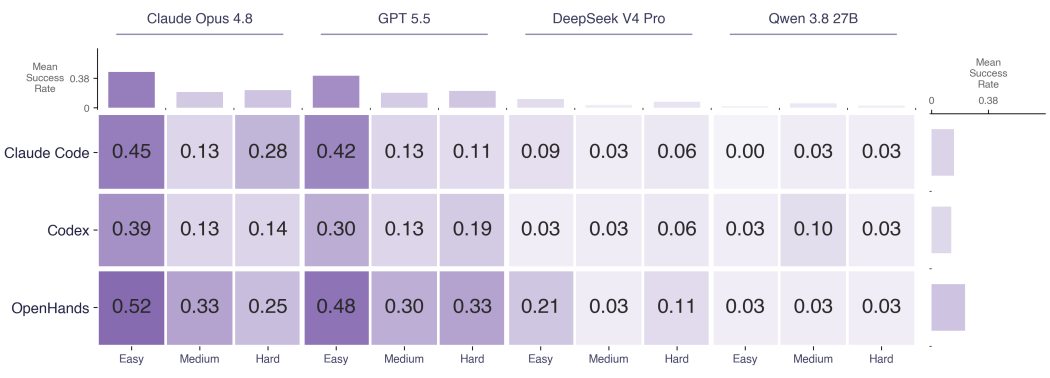


Figure G.3: Full-success rate of each agent–model configuration by difficulty level on all LAMMPS tasks. Bars above and to the right give mean values over models and frameworks, respectively.

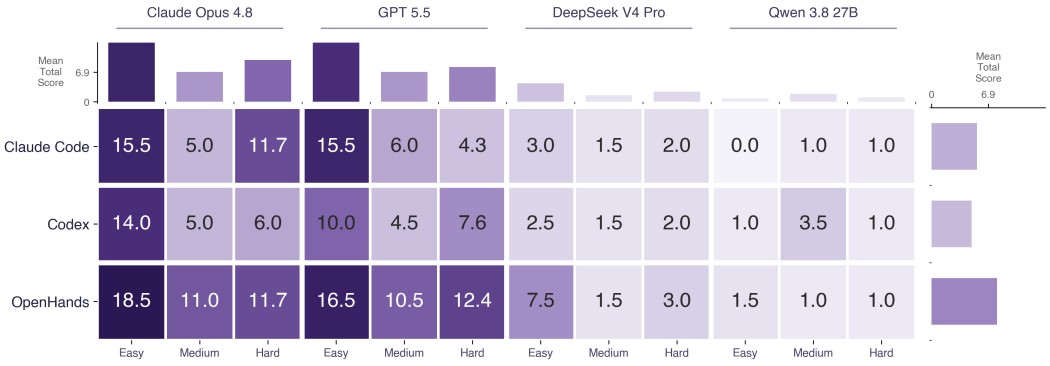


Figure G.4: Total partial-credit score, the sum of per-problem scores s, of each agent–model configuration by difficulty level on all LAMMPS tasks. Bars above and to the right give means over models and frameworks, respectively.

30B. For a fixed proprietary model, the frameworks differ by at most 6 percentage points, or three problems, which lies within the sampling uncertainty of the subset. The native framework of a model is not consistently its best: Claude Sonnet 4.6 and GPT 5.2 Codex score highest under OpenHands, and GPT 5.5 under Claude Code, although none of these differences exceeds two problems. The open-weight models DeepSeek-V4 Pro and Qwen3.6-27B solve problems only under OpenHands. These results agree with the evaluation on the full benchmark in Section 5 and with the finding of Ríos-García et al. (2026), across eight scientific domains, that the base model explains far more of the variance in agent performance than the scaffold.

I STATISTICAL SIGNIFICANCE OF RESULTS

I.1 TESTS AND METHODOLOGY

We conduct four complementary statistical analyses on the per-problem partial credit scores ($s \in [0, 1]$) and binary success rates across all 12 agent-model configurations (three harnesses × four models) reported in Section 5. All tests operate on per-problem scores and are computed using `scipy` and `statsmodels`. Raw scores are released alongside the benchmark to allow independent verification.

Claim 1: Performance degrades monotonically with difficulty. We apply the Kruskal-Wallis test and Spearman rank correlation between difficulty level (coded ordinally as 1, 2, 3) and per-problem

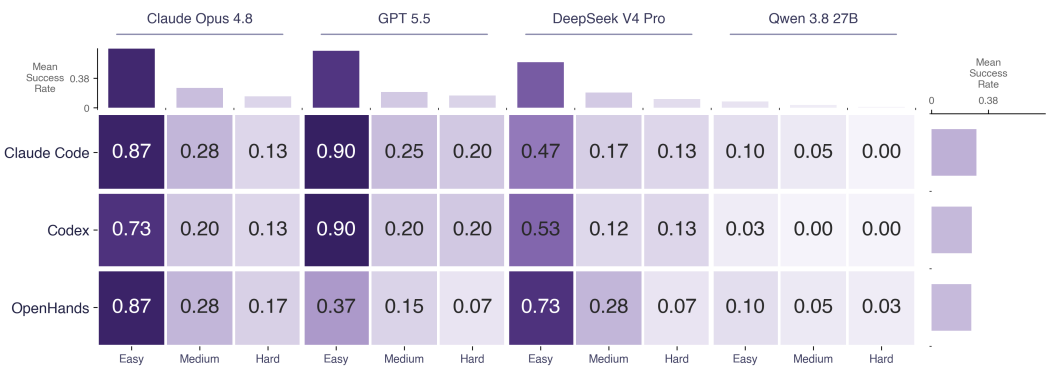


Figure G.5: Full-success rate of each agent–model configuration by difficulty level on all GROMACS tasks. Bars above and to the right give mean values over models and frameworks, respectively.

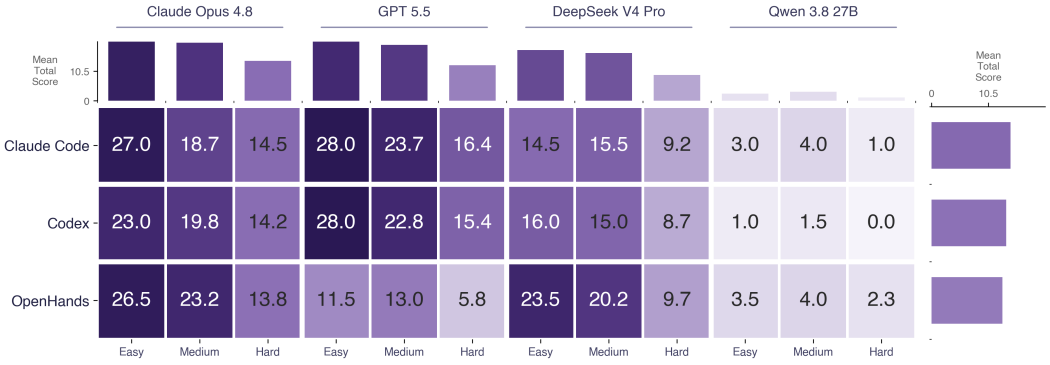


Figure G.6: Total partial-credit score, the sum of per-problem scores s, of each agent–model configuration by difficulty level on all GROMACS tasks. Bars above and to the right give means over models and frameworks, respectively.

score, for each model (aggregated across harnesses) and combined overall. Degradation is highly significant for Claude Opus 4.8 ($H = 49.89$, $p < 0.001$; $\rho = -0.257$, $p < 0.001$), GPT 5.5 ($H = 31.58$, $p < 0.001$; $\rho = -0.222$, $p < 0.001$), and DeepSeek V4 Pro ($H = 12.06$, $p = 0.002$; $\rho = -0.139$, $p < 0.001$). Qwen 3.8 27B shows a directionally consistent but non-significant trend ($H = 5.50$, $p = 0.064$), reflecting a floor effect: scores are already near zero on easy tasks, leaving little statistical power to detect further decline. Combined across all 12 configurations, degradation is highly significant ($H = 51.29$, $\rho = -0.145$, both $p < 0.001$).

Claim 2: Model capability, not harness, is the dominant driver of performance. To directly address whether performance differences arise from the underlying model or from harness-specific engineering, we conduct a two-way ANOVA on per-problem scores with model, harness, and their interaction as factors, reporting the proportion of variance explained (η^2) by each. Model identity explains 15.8% of total variance ($F = 149.37$, $p < 0.001$), harness identity explains only 0.14% and is not statistically significant ($F = 1.92$, $p = 0.147$), and the model×harness interaction explains a further 0.94% ($F = 4.42$, $p < 0.001$). Model capability therefore accounts for over 100× more variance in performance than harness choice. We additionally verify rank consistency: the ranking of the four models by mean score is identical between Claude Code and Codex (Spearman $\rho = 1.00$, $p < 0001$), and highly consistent between each of these and OpenHands ($\rho = 0.80$ for both comparisons), confirming that stronger models remain the strongest performers regardless of harness.

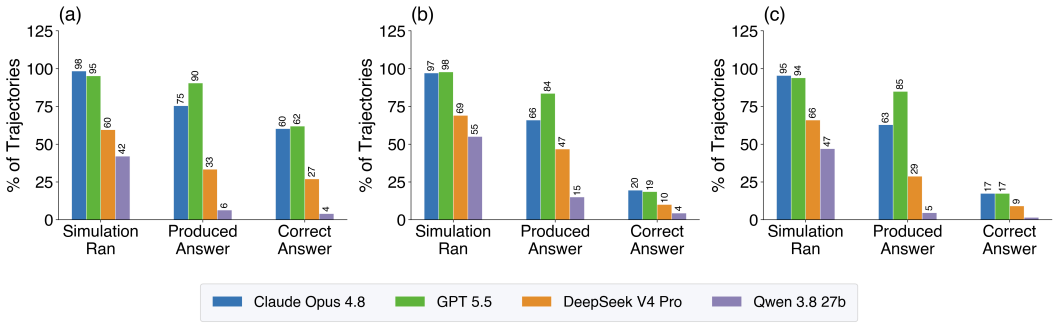


Figure G.7: Task completion across difficulty levels, showing the percentage of trajectories reaching each of three stages: Simulation Ran (agent attempted execution within the time budget), Produced Answer (agent generated a `final_answer.json` output), and Correct Answer (reported values within 5% tolerance of reference). (a) Easy tasks. (b) Medium tasks. (c) Hard tasks. A consistent three-stage drop is observed across all difficulty levels and all agents.

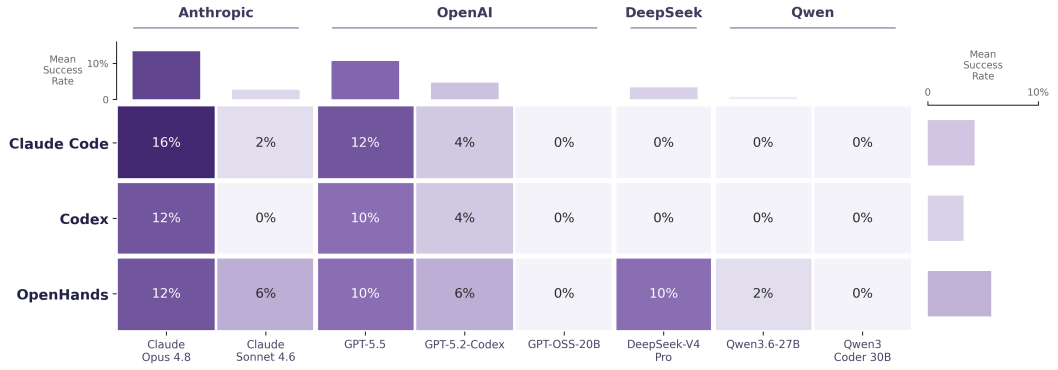


Figure H.8: Full-success rate of every model–framework combination on a common subset of 50 Medium and Hard MDGYM problems. One problem corresponds to 2 percentage points.

Claim 3: Stronger models significantly outperform weaker models. A Kruskal-Wallis test across the four models (pooling harnesses) confirms a highly significant overall difference ($H = 406.15$, $p < 0.001$). Mean scores are 0.472 (Claude Opus 4.8), 0.426 (GPT 5.5), 0.265 (DeepSeek V4 Pro), and 0.053 (Qwen 3.8 27B). Pairwise Mann-Whitney U tests show that all pairwise differences are significant at $p < 0.001$ except Claude Opus 4.8 vs. GPT 5.5 ($p = 0.074$), which are statistically indistinguishable frontier-tier models; every other pair, including all comparisons against Qwen 3.8 27B, is significant at $p < 0.001$.

Claim 4: The benchmark is genuinely hard. We report Wilson score 95% confidence intervals on binary success rates for all 36 (harness, model, difficulty) combinations, summarized in Table I.5. The single highest observed success rate — Claude Opus 4.8 via OpenHands on easy tasks — has an upper confidence bound of 78.4%, and all configurations at hard difficulty have upper confidence bounds at or below 17.0%, confirming that even the strongest agent-model pairs remain far from reliable performance and that this gap is not an artefact of sample size.

Table I.5: Wilson 95% confidence intervals on success rates, top and bottom 10 of 36 (harness, model, difficulty) configurations by success rate.

Harness	Model	Level	Successes/N	Rate	95% CI
OpenHands	Claude Opus 4.8	Easy	43/63	68.3%	56.0%–78.4%
Claude Code	Claude Opus 4.8	Easy	41/63	65.1%	52.8%–75.7%
Claude Code	GPT 5.5	Easy	41/63	65.1%	52.8%–75.7%
Codex	GPT 5.5	Easy	37/63	58.7%	46.4%–70.0%
Codex	Claude Opus 4.8	Easy	35/63	55.6%	43.3%–67.2%
		...			
Codex	Qwen 3.8 27B	Medium	3/70	4.3%	1.5%–11.9%
Codex	Qwen 3.8 27B	Easy	2/63	3.2%	0.9%–10.9%
OpenHands	Qwen 3.8 27B	Hard	2/64	3.1%	0.9%–10.7%
Claude Code	Qwen 3.8 27B	Hard	1/64	1.6%	0.3%–8.3%
Codex	Qwen 3.8 27B	Hard	1/64	1.6%	0.3%–8.3%