🔍 PDE-SHARP: PDE SOLVER HYBRIDS THROUGH ANALYSIS AND REFINEMENT PASSES

Shaghayegh Fazliani *
Department of Mathematics
Stanford University, Stanford, CA

Madeleine Udell
Department of Management Science & Engineering
Stanford University, Stanford, CA

ABSTRACT

Current LLM-driven approaches using test-time computing to generate PDE solvers execute a large number of solver samples to identify high-accuracy solvers. These paradigms are especially costly for complex PDEs requiring substantial computational resources for numerical evaluation. We introduce PDE-SHARP, a framework to reduce computational costs by replacing expensive scientific computation by cheaper LLM inference that achieves superior solver accuracy with 60-75% fewer computational evaluations. PDE-SHARP employs three stages: **(1) Analysis**: mathematical chain-of-thought analysis including PDE classification, solution type detection, and stability analysis; **(2) Genesis**: solver generation based on mathematical insights from the previous stage; and **(3) Synthesis**: collaborative selection-hybridization tournaments in which LLM judges iteratively refine implementations through flexible performance feedback. To generate high-quality solvers, PDE-SHARP requires fewer than 13 solver evaluations on average compared to 30+ for baseline methods, improving accuracy uniformly across tested PDEs by $4\times$ on average, and demonstrates robust performance across LLM architectures, from general-purpose to specialized reasoning models.

1 INTRODUCTION

Partial Differential Equations (PDEs) are fundamental to scientific modeling across physics, engineering, and computational sciences, yet writing robust numerical solvers requires specialized numerical analysis expertise for PDE-specific implementation and tuning, with limited flexibility as each solver targets specific PDE types. The success of deep learning has motivated the development of neural PDE solvers, with Physics-Informed Neural Networks (PINNs) (Raissi et al., 2019; Karniadakis et al., 2021) and operator learning methods (Li et al., 2020) emerging as promising alternatives that leverage neural networks to approximate PDE solutions. However, these approaches require extensive training data, lack interpretability, suffer from generalization limits across PDE families, and offer limited accuracy (Rahaman et al., 2019; Wang et al., 2022) The result is an ecosystem of specialized PDE solvers that address particular failure modes without a systematic understanding of underlying limitations (Cuomo et al., 2022; Krishnapriyan et al., 2021; Zhang et al., 2021; Wang et al., 2021a).

Meanwhile, large language models (LLMs) have demonstrated remarkable aptitude for complex mathematical and scientific challenges (Romera-Paredes et al., 2024; Tian et al., 2024). Sophisticated code generation frameworks employ Chain-of-Thought (CoT) reasoning (Welleck et al., 2024; Wei et al., 2023; Kojima et al., 2023), Mixture-of-Agents (MoA) strategies (Sharma, 2024; Wang et al., 2024a), and advanced inference-time scaling techniques (Snell et al., 2024) to achieve state-of-the-art performance across programming tasks. LLM-as-a-judge frameworks (Jiang et al., 2025a; Zheng et al., 2023) typically employ predetermined evaluation rubrics. However, PDE solver evaluation presents unique challenges requiring assessment of mathematical correctness, numerical stability, computational efficiency, and domain-specific accuracy, factors that demand context-dependent evaluation criteria rather than static rubrics, as optimal trade-offs and performance standards vary significantly across PDE families and application domains. The task of creating reliable solver codes for PDEs sits at the intersection of applied mathematics, numerical analysis, and code generation, making it an ideal testbed to evaluate LLMs' mathematical and technical capabilities. Current approaches fall into two general categories. 1) Fine-tuning methods specialize models for mathematical reasoning (Lu et al., 2024) and subsequent domain-specific adaptation to particular PDE families (Soroco et al., 2025). These require substantial computational resources for multi-stage training and offer limited generalizability across PDE types. 2) Inference-only frameworks using general-purpose LLMs and techniques such as automated debugging (Chen et al., 2023), self-refinement (Madaan et al., 2023), and test-time scaling (Snell et al., 2024). CodePDE (Li et al.,

---

2025) avoids fine-tuning but relies on brute-force sampling strategies, generating and executing 30+ solver candidates to identify optimal solutions. This paradigm becomes especially costly for complex PDEs requiring high-performance computing resources for numerical evaluation.

To address these limitations, we introduce **PDE-SHARP**, an LLM-driven PDE solver generation framework that achieves superior accuracy with 60-75% fewer computational evaluations — through intelligent generation rather than exhaustive sampling — in three stages: **(1) Analysis** analyzes the PDE through structured questions to develop a numerically-stable solver plan; **(2) Genesis** generates solver candidates without immediate execution; **(3) Synthesis** uses LLM judges to iteratively select, execute, and refine solvers based on provided performance feedback in each round. With this approach, PDE-SHARP swaps inexpensive LLM inference for expensive scientific computation, only executing refined solvers each round. This exchange is worthwhile for computationally intensive PDEs for which GPU/HPC resources dominate costs.
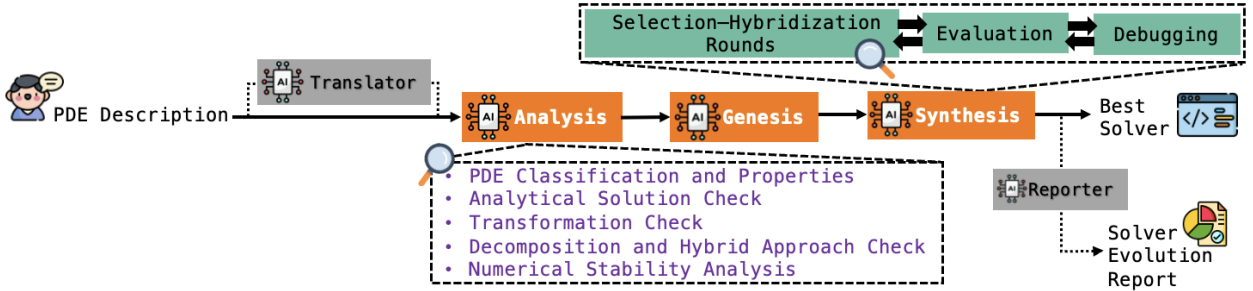

Figure 1: PDE-SHARP framework overview. The three core stages are Analysis, Genesis, and Synthesis. Optional components (Translator, Reporter) enhance usability as explained in section 3. PDE-SHARP generates higher accuracy solvers with 60-75% fewer solver evaluations compared to tested baselines.

Contributions. The experimental results highlight PDE-SHARP's key contributions:

- **Computational Efficiency.** PDE-SHARP reduces expensive solver evaluations by 60-75% (requiring fewer than 13 solver evaluations on average compared to 30+ in best-of-n baselines) while achieving superior solution accuracy, demonstrating considerable resource savings for complex simulations.

- **Mathematical Analysis.** PDE-specific mathematical chain-of-thought reasoning with targeted stability analysis produces mathematically-informed solver strategies, leading to higher initial code quality compared to generic template-based generation.

- **Collaborative LLM Tournaments.** PDE-SHARP's synthesis phase improves on standard test-time computing approaches by $4\times$ on average using fewer evaluations.

- **Enhanced Implementation Quality.** Experiments indicate PDE-SHARP solvers achieve bug-free execution in 63-67% fewer debugging iterations (0.33 vs. 0.9-1.4 iterations per solver) and enjoy superior numerical convergence properties.

- **Robustness to LLM Choice.** PDE-SHARP achieves more consistent performance across diverse LLM types (general-purpose, coding-specific, reasoning models) compared to the baselines, showing robustness to the underlying code generator LLM choice.

- **Flexible Feedback Integration.** PDE-SHARP can improve solvers using several feedback mechanisms — solution-based metrics (relative error), physics-based metrics (PDE residual), and no feedback — to adapt to research scenarios from benchmark validation with known solutions to real-world cases with limited simulation data or physics-only assessments.

2 BACKGROUND & RELATED WORK

Classical Solvers & Neural Methods. Traditional numerical methods for PDE solving, e.g. finite difference, finite element, and spectral methods, require considerable domain expertise for effective implementation (Strang, 2007; LeVeque, 2007). Modern scientific computing frameworks such as FEniCS (Alnaes et al., 2015), deal.II (Arndt et al., 2021) for finite element, and PETSc (Balay et al., 2025) have facilitated access to these methods for broad PDE classes. However, 1) considerable numerical analysis knowledge is still required for optimal performance; and 2) general approaches fail at exploiting PDE-specific mathematical structure to achieve superior performance. The key challenge is thus identifying which approach suits a particular PDE without extensive domain expertise.

The success of deep learning has motivated extensive research into neural PDE solvers. PINNs variants (Raissi et al., 2019; Wang et al., 2022) approximate PDE solutions through residual minimization. Physics-informed operator learning methods (Li et al., 2020; Lu et al., 2021) learn solution operators rather than individual solutions, offering improved generalization. Feature engineering techniques such as random Fourier features (Wang et al., 2021b; Fazliani et al., 2025), residual-based attention (Anagnostopoulos et al., 2023), and radial basis functions (Zeng et al., 2024) have further enhanced neural solver capabilities. Foundation models leverage transformer architectures for multiphysics problems (McCabe et al., 2024; Hao et al., 2024; Shen et al., 2024; Herde et al., 2024). These neural approaches, however, require extensive training data, lack transparency and interpretability regarding solution generation processes, and have generalization limits.

Custom solver generation offers several advantages over neural surrogates and black-box library usage: full algorithmic transparency enables targeted PDE-specific optimization, simplified debugging and modification, and direct control over every detail. This is crucial when solver behavior needs explanation or when problem-specific modifications are required.

LLM-Driven Code Generation for PDEs. The integration of LLMs into scientific computing has emerged along two primary paradigms. First is fine-tuning models pretrained on mathematical tasks for domain-specific applications. MathCoder2 (Lu et al., 2024) demonstrates improved mathematical reasoning through continued training. PDE-Controller (Soroco et al., 2025) continues this approach by fine-tuning MathCoder2-DeepSeekMath on specific PDE families such as heat and wave equations. While effective for targeted applications, this paradigm requires substantial computational resources for multi-stage training and limits generalizability across diverse PDE types. Second is leveraging inference-time optimization techniques to enhance performance. CodePDE (Li et al., 2025) implements automated debugging and test-time sampling for diverse solver generation. Frameworks such as OptiLLM (Sharma, 2024) integrate multiple inference optimization strategies including Chain-of-Thought (CoT), Mixture-of-Agents (MoA), self-reflection, PlanSearch, etc. These approaches typically rely on computationally expensive best-of-n sampling strategies, generating and evaluating large numbers of solver candidates to identify optimal solutions, which becomes prohibitive for complex PDEs requiring substantial evaluation resources.

Both paradigms face fundamental limitations in balancing solution quality with computational efficiency, motivating the need for more intelligent synthesis approaches that leverage mathematical reasoning without exhaustive sampling or extensive fine-tuning requirements.

3 PDE-SHARP FRAMEWORK

Stage 1: Analysis. PDE-SHARP conducts a systematic five-step mathematical analysis to guide solver generation. The process begins with PDE classification (order, linearity, type, boundary conditions) that informs all subsequent decisions. Sequential checks determine if analytical solutions exist, whether transformations can simplify the problem, and if operator decomposition (e.g., separating diffusion and reaction terms) is viable. Each step either directs the framework toward specialized solution strategies in Stage 2 or continues to the next analysis step as shown in Figure 2. The final stability analysis computes symbolic time-step bounds and selects numerically stable schemes, performed before hybrid/numerical solver generation to ensure robustness. Ablation studies (Appendix B.2) demonstrate the effectiveness of this multi-step paradigm over other alternatives.
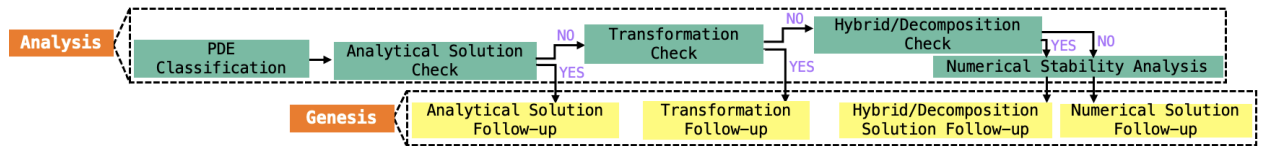

Figure 2: PDE-SHARP Analysis and Genesis stages.

Stage 2: Genesis. PDE solver code is generated using information from the Analysis stage.

Stage 3: Synthesis This stage uses Selection–Hybridization Tournaments with LLM judges to iteratively refine solver implementations. Numerical accuracy of the solver can inform judge decisions through a configurable feedback mechanism. Synthesis consists of two main steps:

(i) Initial Judgment & Selection: Given the n generated initial solvers and a specified feedback type, each judge LLM produces a selection of its top $\frac{n}{2}$ choices from the initial list with reasoning behind each choice (prompt format detailed in Appendix F.3). Each judge also designates one solver from its top $\frac{n}{2}$ list as a nominee for execution and evaluation using the allowed feedback.
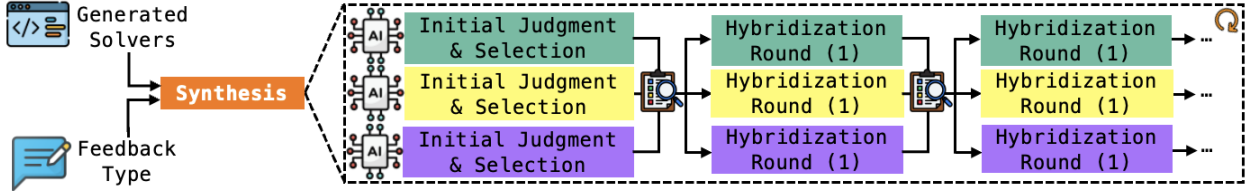
Figure 3: PDE-SHARP Synthesis. This stage can be repeated to address performance saturation.

(ii) Hybridization Rounds: The three nominated base solvers are executed and their performance results are shared with all judges. Each judge then proposes modifications to their base solver using a diff/patch format to ensure incremental changes that preserve working code structure and encourage local fixes, with technical justification for each modification. The modified solvers are executed and results again shared with all judges. This process repeats until performance improvements saturate across consecutive rounds or as specified by the user.

When performance improvements saturate or the maximum number of hybridization rounds is reached, the framework initiates another **judging cycle** that repeats steps (i) and (ii) with an expanded solver set including all previously generated hybrids, their technical justifications, and performance feedback from previous rounds. Judges maintain context within each cycle but reset between cycles, evaluating the expanded set from scratch, to encourage exploration of new strategies.

Feedback Types. The Synthesis stage can incorporate different performance metrics to guide judge decisions during tournaments. We discuss three feedback types: (1) nRMSE: normalized root mean squared error against reference solutions; (2) PDE residual feedback: physics-based residual computation that requires no reference data; and (3) no feedback: judges rely purely on code analysis. The choice of feedback type allows adaptation to different research scenarios — from benchmark validation with known solutions to real-world cases with limited reference data. PDE-specific feedback types and their combinations could also be employed for domain-specific optimization. Additional discussions and results appear in Appendix B.2.6.

Optional Stages. PDE-SHARP includes two optional components for enhanced usability (Figure 1): **Translator** converts natural language PDE descriptions into the structured mathematical templates required by the Analysis stage. When user input lacks necessary detail, it requests additional information before proceeding. Users can alternatively bypass this stage by directly providing pre-formatted templates. **Reporter** generates detailed reports on solver evolution throughout the tournament process, enhancing framework interpretability. An example of such report for the reaction-diffusion PDE is provided in Appendix E. These reports can serve as feedback for subsequent runs on the same problem, enabling iterative refinement strategies.

4 EXPERIMENTS

We compare PDE-SHARP against multiple baseline methods across five representative PDE tasks from PDEBench (Takamoto et al., 2024) (Table 1). Discussions on neural methods and some LLM-driven approaches (agentic workflows, fine-tuned mathematical models, etc.) appear in Appendix A. In our experiments, we focus on LLM-driven baselines using test-time computing for code generation that directly compete with PDE-SHARP's approach. **Code-PDE** (Li et al., 2025) generates solvers using chain-of-thought prompting and executes all samples to report the best performance. A refined variant, **CodePDE-R**, is also tested as a baseline. **OptiLLM** (Sharma, 2024) implements inference optimization techniques including Chain-of-Thought (CoT), Mixture-of-Agents (MoA), and Cerebras Planning and Optimization (CePO). Experimental details appear in Appendix A.

Table 1: Tested PDEs; details in Appendix C. **Dimension** column indicates the *spatial dimension* and NL stands for non-linear in the table.

PDE	Dimension	Type	State	Solution Behavior
Advection	1D	Linear	Time-dependent	Smooth
Burgers	1D	Highly NL	Time-dependent	Shock-forming
Reaction-Diffusion	1D	Mildly NL	Time-dependent	Smooth
Navier-Stokes	1D	Highly NL	Time-dependent	Shock-forming
Darcy Flow	2D	Mildly NL	Steady-state	Smooth

Experimental Setup: All methods generate $n = 32$ initial solver candidates for fair comparison (Appendix B.2.4). Baselines execute all candidates (CodePDE-R executes 44 with refinements). PDE-SHARP uses three judge LLMs (Appendix B.2.5) in collaborative tournaments, executing only refined candidates per hybridization round. For Section 4 experiments, PDE-SHARP uses nRMSE on 100 validation samples as tournament feedback. All methods are evaluated on a separate test set of 100 random PDEBench samples per PDE task (Table 2). Additional feedback types and judge configurations appear in Appendix B.2.

4.1 RESULTS & ANALYSIS

Table 2 shows solver accuracy across all PDEs and baselines.

Table 2: PDE-SHARP improves solver accuracy and is robust to choice of LLM. Solution accuracy is measured by nRMSE relative to the reference solution from PDEBench. Cell colors use a colormap log-normalized independently within each PDE column to highlight per-task variation.

		Advection	Burgers	Reaction-Diffusion	Navier-Stokes	Darcy
OptiLLM-CoT	Gemma 3	5.34e-03	5.32e-02	2.07e-01	9.58e-02	8.01e-02
	LLaMA 3.3	7.71e-03	4.38e-02	2.24e-01	2.42e-01	1.01e+00
	Qwen 3	4.67e-03	1.52e-03	9.38e-01	2.63e-01	6.34e-01
	DeepSeek-R1	4.97e-03	3.04e-04	2.45e-01	8.34e-02	5.34e-03
	GPT-4o	1.72e-03	2.12e-03	2.23e-02	2.01e-01	8.51e-01
	o3	9.74e-04	4.08e-04	2.21e-01	3.12e-02	5.47e-03
OptiLLM-MoA	Gemma 3	3.97e-03	4.21e-03	1.74e-01	6.78e-02	4.69e-02
	LLaMA 3.3	1.23e-03	4.71e-03	1.49e-01	2.29e-01	2.13e-01
	Qwen 3	1.01e-03	3.45e-04	9.68e-02	1.79e-02	5.12e-03
	DeepSeek-R1	9.74e-04	2.49e-04	1.48e-01	1.65e-02	5.01e-03
	GPT-4o	2.01e-03	2.41e-04	1.94e-02	2.56e-02	5.02e-03
	o3	1.74e-03	2.91e-04	2.09e-01	1.39e-02	5.07e-03
OptiLLM-CePO	Gemma 3	3.74e-03	4.01e-03	1.89e-01	6.32e-02	4.12e-02
	LLaMA 3.3	1.11e-03	4.53e-03	1.36e-01	2.18e-01	1.98e-01
	Qwen 3	1.01e-03	3.23e-04	8.91e-02	1.97e-02	4.83e-03
	DeepSeek-R1	9.71e-04	2.43e-04	1.39e-01	1.79e-02	4.78e-03
	GPT-4o	9.88e-04	2.31e-04	1.67e-02	2.31e-02	4.88e-03
	o3	9.88e-04	2.74e-04	2.03e-01	1.49e-02	4.81e-03
CodePDE	Gemma 3	5.61e-03	5.17e-02	2.13e-01	9.29e-02	7.69e-02
	LLaMA 3.3	7.37e-03	4.59e-02	2.18e-01	2.36e-01	1.03e+00
	Qwen 3	4.89e-03	1.35e-03	9.55e-01	2.59e-01	6.57e-01
	DeepSeek-R1	1.01e-03	3.04e-04	2.13e-01	2.80e-02	4.80e-03
	GPT-4o	1.55e-03	3.65e-04	1.99e-02	1.81e-01	6.57e-01
	o3	9.74e-04	2.74e-04	1.99e-02	9.29e-02	4.88e-03
CodePDE-R	Gemma 3	4.20e-03	4.63e-03	1.69e-01	6.44e-02	4.47e-02
	LLaMA 3.3	1.02e-03	4.59e-03	1.43e-01	2.36e-01	1.92e-01
	Qwen 3	9.74e-04	3.60e-04	9.13e-02	1.67e-02	4.90e-03
	DeepSeek-R1	1.01e-03	3.15e-04	1.67e-02	1.67e-02	4.80e-03
	GPT-4o	9.74e-04	2.57e-04	1.67e-02	2.36e-02	4.80e-03
	o3	1.01e-03	3.60e-04	1.43e-01	1.31e-02	4.90e-03
PDE-SHARP	Gemma 3	1.01e-03	5.60e-04	3.01e-03	3.14e-02	1.72e-02
	LLaMA 3.3	9.98e-04	4.61e-04	3.61e-03	5.06e-02	1.72e-02
	Qwen 3	7.76e-04	2.97e-04	2.32e-03	2.80e-02	4.80e-03
	DeepSeek-R1	5.24e-04	1.48e-04	2.29e-03	1.37e-02	4.74e-03
	GPT-4o	6.11e-04	2.31e-04	2.29e-03	1.51e-02	3.97e-03
	o3	9.74e-04	3.42e-04	5.78e-03	1.89e-02	7.78e-03

PDE-SHARP is more robust to code generator LLM selection. Table 2 shows that the solution quality for baseline methods depends strongly on the LLM. In contrast, PDE-SHARP performs more consistently across all tested LLMs; results for more LLMs are appear Appendix B.1. This uniform performance indicates PDE-SHARP's tournament hybridization stage effectively mitigates the limitations of individual code generators, producing higher-quality solvers that are largely independent of the underlying LLM.

PDE-SHARP significantly improves solver accuracy for specific PDEs. PDE-SHARP improves accuracy by over $4\times$ overall (geometric mean), with particularly impressive performance on the reaction-diffusion and advection tasks. For reaction-diffusion, PDE-SHARP's Analysis stage immediately identifies that the reaction component admits an analytical solution, directing all 32 initial solver candidates toward hybrid analytical-numerical approaches that achieve superior numerical stability. Baseline methods rarely discover this hybrid strategy, as shown in Figure 4a.
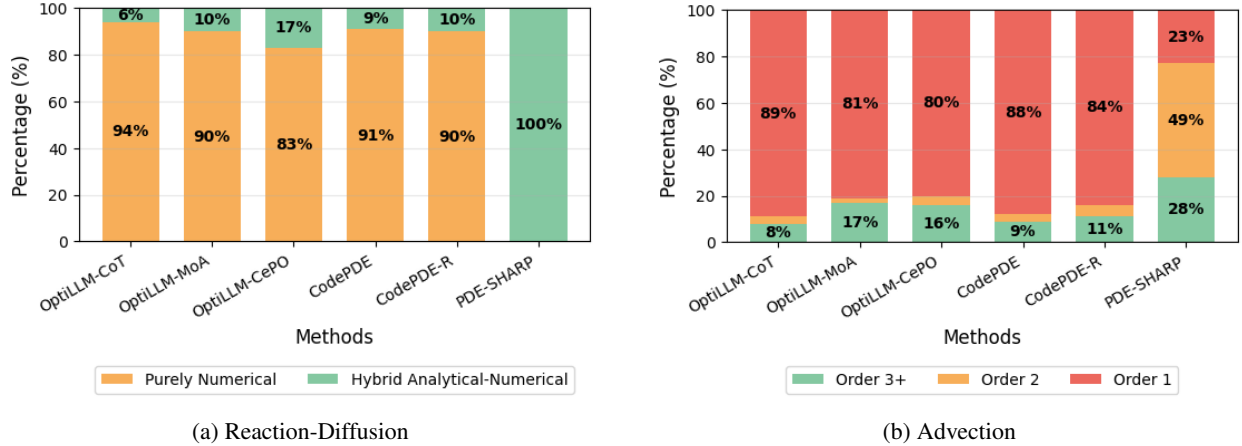

| (a) Reaction-Diffusion | (b) Advection |

Figure 4: (a) Other frameworks tend to choose the less accurate purely-numerical approach for the reaction-diffusion PDE, while PDE-SHARP always goes with the superior hybrid approach. (b) PDE-SHARP transitions from first-order discretized analytical to second-order finite-volume approaches through performance-informed tournaments.

For advection, PDEBench reference solutions are generated using finite volume methods (Takamoto et al., 2024), reflecting standard shock-safe computational practice. PDE-SHARP and all other baselines initially attempt analytical solutions, and the baselines keep their analytical approach even through refinement (e.g. in CodePDE-R). PDE-SHARP's performance-informed tournaments, on the other hand, encourage PDE-SHARP to adapt to the data, as demonstrated in Figure 4b. When persistent 10^{-3} errors reported as feedback indicate a mismatch between analytical and reference solutions, the judge LLMs converge on second-order finite-volume schemes that better match the dataset characteristics. This adaptation occurs through feedback alone, without manual intervention, demonstrating how collaborative tournaments can optimize for evaluation criteria while maintaining computational efficiency. This adaptive behavior varies with different feedback types as users can choose an optimization target to reflect available data (Figure 5). A study on advection solvers appears in Appendix D.1.
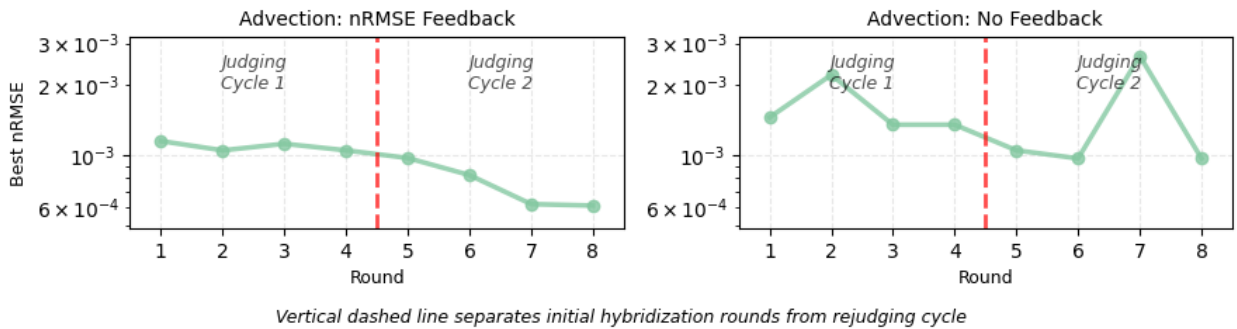

Figure 5: Without proper feedback, the judges stick to analytical approaches. Figure 16 gives details.
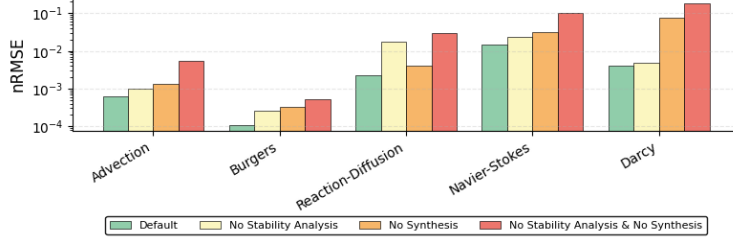
Figure 6: Ablation study of PDE-SHARP components across five PDE tasks. Four variants: (1) Default: full PDE-SHARP with both stability analysis and synthesis, (2) No Stability Analysis: PDE-SHARP with the stability analysis step removed from the Analysis stage, (3) No Synthesis: PDE-SHARP with best-of-32 sampling instead of the Synthesis stage, and (4) No Stability Analysis & No Synthesis. Results show both components contribute to accuracy improvements, with each component being more critical for different PDE types, e.g. stability analysis is more critical for reaction-diffusion, while synthesis contributes more to the Darcy flow task.

Figures 4, 5, 6 highlight how PDE-SHARP's **Analysis** and **Synthesis** stages leverage mathematical insight and performance feedback, both playing significant roles in PDE-SHARP's performance. Detailed ablation studies in Appendix B.2 quantify each component's contribution in more detail.

4.1.1 CASE STUDY SUMMARY: REACTION–DIFFUSION SOLVER EVOLUTION

A detailed account of this synthesis process appears in Appendix E. Here, we summarize a report of the evolution of a top-performing solver for the 1D reaction–diffusion PDE

$$\partial_t u - \nu \partial_{xx} u - \rho u(1-u) = 0 \quad \text{with periodic boundaries on } x \in (0,1),$$

under a parameter regime of $\nu = 0.5$ and $\rho = 1.0$ during PDE-SHARP's hybrid tournaments. The process involved four rounds of judge-guided iteration and hybridization, and ultimately reduced the L2 error by a factor of $77\times$—from 0.166 to 0.002—without changing the time step size or switching to implicit methods.

Round 1 (Baseline): The initial solver used Lie splitting (reaction followed by diffusion), where the reaction was integrated analytically via a logistic formula. While the implementation was correct and stable, its time step was overly conservative due to a misapplied stability constraint that included both diffusion and reaction terms. Since the reaction step is exact (not discretized), its inclusion unnecessarily limited Δt, resulting in over two million substeps per output trajectory and moderate L2 error (0.166).

Round 2 (Hybridization Attempt): To address inefficiencies, judges merged features from other candidate codes: using only the diffusion-based Δt bound, switching to Strang splitting for second-order accuracy, and retaining the analytical reaction. However, these changes failed to reduce cost or improve accuracy. The Δt bound remained governed by diffusion due to spatial resolution ($N = 1024$), and the increased number of operator applications in Strang splitting amplified phase errors. The L2 error worsened slightly (0.185), despite using the same number of internal steps.

Round 3 (Implicit Diffusion): Attempting a more substantial architectural change, the judges replaced explicit diffusion with an unconditionally stable Crank–Nicolson method. This permitted much larger time steps (up to $39\times$ larger), but performance degraded sharply (L2 error 0.301). Investigation revealed that the tridiagonal solver was incompatible with periodic boundary conditions and that the simplified Lie splitting strategy (one reaction step followed by one implicit diffusion step per output time) created significant truncation error. The attempted optimization thus failed due to both boundary mismatch and operator imbalance.

Round 4 (Targeted Local Fix): Judges reverted to the more promising Strang splitting configuration from Round 2, but focused on a subtle issue in the reaction step. Although mathematically correct, the original logistic integration formula suffered from catastrophic floating-point cancellation: when $u \approx 0$ or $u \approx 1$, the formula introduced numerical instability or overflow. The key breakthrough was a numerically stable reformulation of the same expression:

$$\texttt{return } 1/\left(1 + e^{-\rho \Delta t} \cdot \frac{1-u}{u+\epsilon}\right), \quad \epsilon = 10^{-10}.$$

7

This stabilized the computation without altering the underlying algorithm. The result was a dramatic improvement: L2 error dropped to 0.002 while keeping the same time step size and structure as the original baseline. This final solver was robust, accurate, and production-quality—free of NaNs, Infs, or edge-case failures.

Key Lessons:

- *Numerical stability often outweighs algorithmic complexity.* The most impactful change was not in the solver structure, but in improving a single line of floating-point math.
- *Blind use of advanced methods can harm performance.* The implicit method in Round 3 degraded accuracy due to mismatched assumptions and poor boundary treatment.
- *Incremental, analysis-driven refinement is highly effective.* Careful error tracing and domain-specific insight enabled major improvements with minimal code changes.

4.2 CODE QUALITY & INSIGHTS

Figure 7 demonstrates PDE-SHARP reduces the number of debugging iterations required and produces solvers with competitive execution times. PDE-SHARP averages 0.33 debugging iterations per solver execution (approximately 1 in 3 generated solvers requires debugging in a hybridization round), significantly outperforming baseline methods that require 0.9–1.4 debugging iterations per generated solver. This reduction shows that PDE-SHARP's Analysis stage produces more robust initial implementations, and that the synthesis stage efficiently eliminates implementation errors.

(a)   (b)
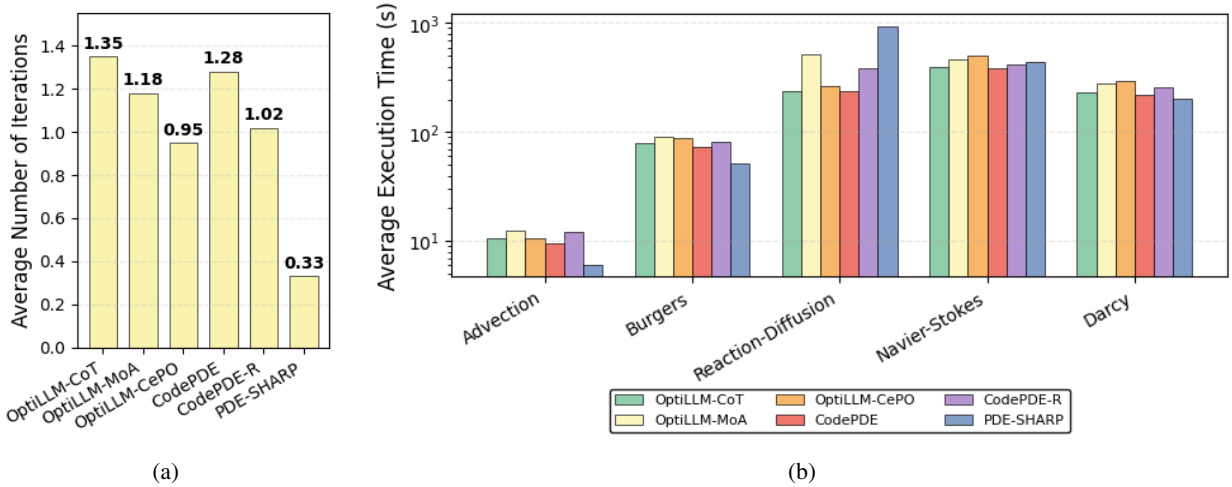
Figure 7: (a) Average number of debugging iterations required per solver execution across different methods. (b) Average execution times across PDE tasks. PDE-SHARP achieves lower execution times than the average baseline in 4/5 cases. For reaction-diffusion, higher execution time reflects the rigorous numerical methods selected by stability analysis as expected, which produce significantly higher accuracy solvers (Table 2).

Figure 4b demonstrates the distribution of empirical convergence orders (definition in Appendix A) — showing solver improvements with grid refinement — across methods for the advection PDE. PDE-SHARP generates solvers with superior convergence properties, leading to higher accuracy in this case (Table 2). In addition, Figure 15 indicates that on average, PDE-SHARP's solvers use less PyTorch (down to ≈25–33 % of library calls) and more SciPy + NumPy + JAX (up to ≈60–75 %), whereas the baselines keep PyTorch at roughly 50–67% and SciPy below 7% on average. Using JAX for computational kernels is highly encouraged in PDE-SHARP prompts in particular as evident in the library usage proportions across all methods and PDE tasks. Additional empirical convergence rate results all PDEs as well as library usage proportions for each baseline appear in Appendix B.3.

Cost. We analyze the efficiency and cost of each method by calculating the average cost for GPU and LLM API calls for the experiments in this section. Table 2 shows among the tested LLMs, GPT-4o as the code generation LLM yields higher accuracy results on average. Table 3 shows the total average API cost of the results for GPT-4o in Table 2. Details of the calculations appear in Appendix A.4. GPU usage depends on the number of solver executions, code complexity, and implementation efficiency. The number of solver executions for PDE-SHARP depends on the number

of hybridization rounds required, averaging 13.2 evaluations across all test cases (9-12 evaluations for most PDEs, with advection requiring 24 to better match data as discussed in Section 4.1). Figure 8 shows nRMSE vs. total average cost (API call + GPU usage) for three PDE tasks.

Table 3: Cost comparison of input, output, and total API usage per method using GPT-4o as the code generating LLM.

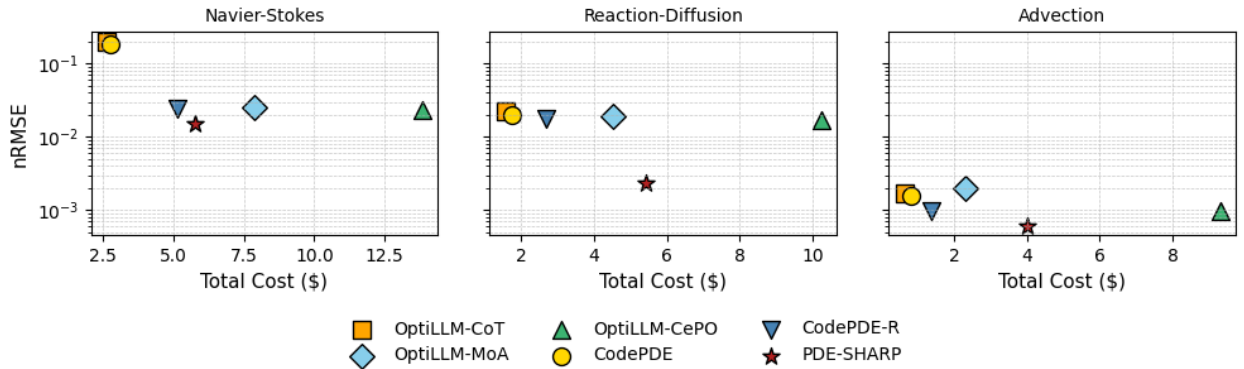| Framework	$ Inputs	$ Output	$ Total
OptiLLM-CoT	0.10	0.48	0.58
OptiLLM-MoA	0.53	2.12	2.65
OptiLLM-CePO	0.96	8.27	9.23
CodePDE	0.07	0.68	0.75
CodePDE-R	0.41	0.88	1.29
PDE-SHARP	1.12	2.89	4.01

Figure 8: Trade-off between solution accuracy (nRMSE) and total cost for LLM-driven PDE solver generation methods across three PDE tasks of varying computational complexity. From Navier-Stokes (hours per solver evaluation) to Reaction-Diffusion (moderate) to Advection (lightweight, seconds per evaluation), PDE-SHARP demonstrates consistent cost-effectiveness.

4.3 DISCUSSION & LIMITATIONS

Discussion: PDE-SHARP uses numerical feedback to improve the generated solver. This extra information can be easy to compute — such as the (data-free) PDE residual — or may require collecting data, such as distance to the solution at a sampled set of times and locations. PDE-SHARP can also use problem-specific constraints like the CFL condition (LeVeque, 2007) as feedback, or can run without feedback if no information is available. Results for PDE-SHARP using residual feedback and no numerical feedback appear in Appendix B.2. LLM agents can also suggest feedback types. As seen in Appendix B.2.6 (examples of LLM-suggested feedback types for each tested PDE), an additional LLM agent could be used to determine optimal problem-specific metrics before Synthesis begins. This is particularly beneficial for complex PDEs requiring specialized feedback, and represents important future work. Additional promising directions include scaling to higher-dimensional problems with complex geometries where traditional numerical methods face greater challenges. Finally, hybrid approaches combining PDE-SHARP's interpretable numerical solvers with neural PDE methods could leverage the strengths of both paradigms for problems requiring both accuracy and computational efficiency.

Limitations: Our evaluation establishes PDE-SHARP's effectiveness on moderate-complexity PDEs from established benchmarks, with high-fidelity computational simulations representing a natural extension constrained by current LLM training data coverage. LLM-driven PDE solver generation using test-time computing approaches rely on LLM mathematical reasoning capabilities, which means performance may degrade for cutting-edge PDE formulations that are not well-represented in training data or require highly specialized domain knowledge beyond current model capabilities.

9

5 CONCLUSION

PDE-SHARP demonstrates that intelligent LLM-driven solver generation can dramatically improve efficiency over brute-force sampling approaches. Our three-stage framework reduces computational evaluations by 60-75% while achieving superior accuracy on average across five representative PDEs. The mathematical chain-of-thought analysis in the Analysis stage produces more robust initial implementations, requiring on average 67% fewer debugging iterations compared to baseline methods. The hybrid tournaments in the Synthesis stage efficiently refines solvers through performance-informed feedback, with flexible type, demonstrating consistent robust improvements across diverse LLM models.

REFERENCES

M. S. Alnaes, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3, 2015. doi: 10.11588/ans.2015.100. 20553.

Sokratis J. Anagnostopoulos, Juan Diego Toscano, Nikolaos Stergiopulos, and George Em Karniadakis. Residual-based attention and connection to information bottleneck theory in pinns, 2023. URL https://arxiv.org/abs/2307.00379.

Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. The deal.ii finite element library: Design, features, and insights. *Computers & Mathematics with Applications*, 81:407–422, January 2021. ISSN 0898-1221. doi: 10.1016/j.camwa.2020.02.022. URL http://dx.doi.org/10.1016/j.camwa.2020.02.022.

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc Web page. https://petsc.org/, 2025. URL https://petsc.org/.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific Machine Learning Through Physics–Informed Neural Networks: Where We Are and What's Next. *J. Sci. Comput.*, 92(3), 2022.

Shaghayegh Fazliani, Zachary Frangella, and Madeleine Udell. Enhancing physics-informed neural networks through feature engineering, 2025. URL https://arxiv.org/abs/2502.07209.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

Zhongkai Hao, Chang Su, Songming Liu, Julius Berner, Chengyang Ying, Hang Su, Anima Anandkumar, Jian Song, and Jun Zhu. Dpot: Auto-regressive denoising operator transformer for large-scale pde pre-training, 2024. URL https://arxiv.org/abs/2403.03542.

Maximilian Herde, Bogdan Raonić, Tobias Rohner, Roger Käppeli, Roberto Molinaro, Emmanuel de Bézenac, and Siddhartha Mishra. Poseidon: Efficient foundation models for pdes, 2024. URL https://arxiv.org/abs/2405.19101.

Hongchao Jiang, Yiming Chen, Yushi Cao, Hung yi Lee, and Robby T. Tan. Codejudgebench: Benchmarking llm-as-a-judge for coding tasks, 2025a. URL https://arxiv.org/abs/2507.10535.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code, 2025b. URL https://arxiv.org/abs/2502.13138.

George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023. URL `https://arxiv.org/abs/2205.11916`.

Aditi S. Krishnapriyan, Amir Gholami, Shandian Zhe, Robert M. Kirby, and Michael W. Mahoney. Characterizing possible failure modes in physics-informed neural networks, 2021. URL `https://arxiv.org/abs/2109.01050`.

Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

Shanda Li, Tanya Marwah, Junhong Shen, Weiwei Sun, Andrej Risteski, Yiming Yang, and Ameet Talwalkar. Code-pde: An inference framework for llm-driven pde solver generation, 2025. URL `https://arxiv.org/abs/2505.08783`.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.

Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. Math-coder2: Better math reasoning from continued pretraining on model-translated mathematical code, 2024. URL `https://arxiv.org/abs/2410.08196`.

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

Michael McCabe, Bruno Régaldo-Saint Blancard, Liam Holden Parker, Ruben Ohana, Miles Cranmer, Alberto Bietti, Michael Eickenberg, Siavash Golkar, Geraud Krawezik, Francois Lanusse, Mariel Pettee, Tiberiu Tesileanu, Kyunghyun Cho, and Shirley Ho. Multiple physics pretraining for physical surrogate models, 2024. URL `https://arxiv.org/abs/2310.02994`.

Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, pp. 5301–5310. PMLR, 2019.

Maziar Raissi, Zhicheng Wang, Michael S Triantafyllou, and George Em Karniadakis. Deep learning of vortex-induced vibrations. *Journal of Fluid Mechanics*, 861:119–137, 2019.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 2023. doi: 10.1038/s41586-023-06924-6.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL `https://arxiv.org/abs/2308.12950`.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL `https://arxiv.org/abs/2402.03300`.

Asankhaya Sharma. Optillm: Optimizing inference proxy for llms, 2024. URL `https://github.com/codelion/optillm`.

Junhong Shen, Tanya Marwah, and Ameet Talwalkar. Ups: Efficiently building foundation models for pde solving via cross-modal adaptation, 2024. URL `https://arxiv.org/abs/2403.07187`.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL `https://arxiv.org/abs/2408.03314`.

Mauricio Soroco, Jialin Song, Mengzhou Xia, Kye Emond, Weiran Sun, and Wuyang Chen. Pde-controller: Llms for autoformalization and reasoning of pdes, 2025. URL `https://arxiv.org/abs/2502.00963`.

Gilbert Strang. *Computational Science and Engineering*. SIAM, 2007. doi: 10.1137/1.9780961408817. URL `https://epubs.siam.org/doi/book/10.1137/1.9780961408817`.

Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Dan MacKinlay, Francesco Alesiani, Dirk Pflüger, and Mathias Niepert. Pdebench: An extensive benchmark for scientific machine learning, 2024. URL `https://arxiv.org/abs/2210.07182`.

Qwen Team. Qwen3 technical report, 2025. URL `https://arxiv.org/abs/2505.09388`.

Minyang Tian, Luyu Gao, Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, HAO TONG, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu A Huerta, and Hao Peng. Scicode: A research coding benchmark curated by scientists. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL `https://openreview.net/forum?id=ADLaALtdoG`.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities, 2024a. URL `https://arxiv.org/abs/2406.04692`.

Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and Mitigating Gradient Flow Pathologies in Physics-Informed Neural Networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021a.

Sifan Wang, Hanwen Wang, and Paris Perdikaris. On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 384:113938, 2021b.

Sifan Wang, Xinling Yu, and Paris Perdikaris. When and why PINNs fail to train: A neural tangent kernel perspective. *Journal of Computational Physics*, 449:110768, 2022.

Sifan Wang, Bowen Li, Yuhan Chen, and Paris Perdikaris. Piratenets: Physics-informed deep learning with residual adaptive networks. *arXiv:2402.00326*, 2024b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL `https://arxiv.org/abs/2201.11903`.

Sean Welleck, Amanda Bertsch, Matthew Finlayson, Hailey Schoelkopf, Alex Xie, Graham Neubig, Ilia Kulikov, and Zaid Harchaoui. From decoding to meta-generation: Inference-time algorithms for large language models, 2024. URL `https://arxiv.org/abs/2406.16838`.

Chengxi Zeng, Tilo Burghardt, and Alberto M Gambaruto. Feature mapping in physics-informed neural networks (pinns), 2024. URL `https://arxiv.org/abs/2402.06955`.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.

Leo Zhiyuan Zhao, Xueying Ding, and B. Aditya Prakash. Pinnsformer: A transformer-based framework for physics-informed neural networks, 2023.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. URL `https://arxiv.org/abs/2306.05685`.

APPENDIX

A ADDITIONAL INFORMATION & EXPERIMENTAL SETUPS

A.1 MATHEMATICAL METRICS

nRMSE. For S test cases, each with true solution $u^{(s)}(x,t)$ and solver prediction $\hat{u}^{(s)}(x,t)$:

$$\text{nRMSE} = \frac{1}{S} \sum_{s=1}^{S} \frac{\|u^{(s)}(x,t) - \hat{u}^{(s)}(x,t)\|_2}{\|u^{(s)}(x,t)\|_2}$$

where $\|\cdot\|_2$ denotes the L2 norm. This metric normalizes the root mean squared error by the magnitude of the true solution, enabling fair comparison across problems with different solution scales.

Convergence Rate. To evaluate numerical correctness, we assess solver convergence behavior across multiple grid resolutions. A robust solver should exhibit predictable error reduction following $E(h) \approx Ch^p$, where $E(h)$ is the solution error on grid spacing h, C is a problem-dependent constant, and p is the convergence order.

We estimate the empirical convergence order using two grid resolutions:

$$p \approx \frac{\log\left(\frac{E(h_1)}{E(h_2)}\right)}{\log\left(\frac{h_1}{h_2}\right)}$$

For each generated solver, we evaluate performance on progressively refined grids (typically h, $h/2$, $h/4$) and compute the average convergence order. Expected theoretical orders vary by numerical method: first-order schemes ($p \approx 1$), second-order finite difference/volume methods ($p \approx 2$), and spectral methods (exponential convergence). Most LLM-generated solvers achieve first-order convergence, with occasional higher-order behavior depending on the chosen discretization scheme and implementation quality.

A.2 NEURAL NETWORKS & FOUNDATION MODELS

Limitations of Cross-Paradigm Comparisons. Direct comparison between LLM-generated solvers using traditional numerical methods and neural PDE solvers involves inherent methodological challenges. Neural network baselines are drawn from prior literature with different experimental conditions while our LLM approach benefits from extensive inference-time optimization (debugging, refinement, best-of-n sampling) not applied to these baselines. Additionally, the computational budgets differ fundamentally: neural methods require training time and data preparation, while numerical methods require implementation and parameter tuning effort. These paradigmatic differences make it difficult to establish truly equivalent experimental conditions. Our results should be interpreted as demonstrating the promise of LLM-based solver generation rather than definitive superiority over alternative approaches. Future work should focus on controlled comparisons with standardized evaluation protocols across all methods.

We thus include the following reported numbers verbatim from the original papers of FNO (Li et al., 2020), PirateNets (Wang et al., 2024b), PINNsFormer (Zhao et al., 2023), and UPS (Shen et al., 2024) as examples of neural and foundation models only for the sake of completeness and to give readers an at-a-glance sense of scale (parameters, memory, time/epoch) and accuracy on overlapping PDE families, however, as each method utilizes distinct settings, we do not provide a direct ranking between them. The following is intended only to document the resource scale and the published accuracy on broadly overlapping PDE families.

FNO Reports results for 1D Burgers and 2D Navier–Stokes (space–time operator learning). Hardware noted by the authors: single NVIDIA V100 16 GB.

PirateNets has PINN backbone with physics-informed residual adaptive blocks. The paper emphasizes accuracy comparisons and ablations; it does not tabulate parameter counts, GPU memory, or wall-clock per epoch. Below we list the state-of-the-art test errors the authors report.

PINNsFormer is a transformer-style PINN variant. The authors report parameter counts and training overhead (V100), and test errors on overlapping 1D PDEs.

Table 4: FNO on *1D Burgers* (relative ℓ_2 error at different spatial resolutions s).

Method	s=256	512	1024	2048	4096	8192
FNO	0.0149	0.0158	0.0160	0.0146	0.0142	0.0139

Notes. Table reproduced from the paper; parameters, GPU memory, and time/epoch were not reported for the Burgers experiment. See Table 5 for Navier–Stokes resource numbers as reported by the authors.

Table 5: FNO on *2D Navier–Stokes* (relative ℓ_2 error over different viscosities ν and dataset sizes N; per-epoch time reported by the authors).

Method	Params	Time/epoch	$\nu=10^{-3}, T=50, N=1000$	$\nu=10^{-4}, T=30, N=1000$	$\nu=10^{-4}, T=30, N=10000$	$\nu=10^{-5}, T=20, N=1000$
FNO-3D	6,558,537	38.99 s	0.0086	0.1918	0.0820	0.1893
FNO-2D	414,517	127.80 s	0.0128	0.1559	0.0973	0.1556
U-Net	24,950,491	48.67 s	0.0245	0.2051	0.1190	0.1982
TF-Net	7,451,724	47.21 s	0.0225	0.2253	0.1168	0.2268
ResNet	266,641	78.47 s	0.0701	0.2871	0.2311	0.2753

Notes. Reported at 64×64 spatial resolution; FNO-3D convolves in space–time while FNO-2D uses 2D convolutions with an RNN in time.

UPS learns to map symbolic PDE specifications and initial/boundary conditions to numerical solutions. The architecture combines Fourier Neural Operators and transformers with autoregressive decoding over space-time grids.

The model was trained on ∼20k PDE trajectories using a single NVIDIA A6000 GPU. Training was run for 60,000 steps and completed in under 100 GPU-hours. UPS achieves strong sample efficiency, outperforming baselines with $4\times$ less data and $26\times$ less compute.

A.3 LLM-Driven Architectures

A.3.1 LLM Models Used in Section 4 for Code Generation

A.3.2 Agentic Workflows

Frameworks like FunSearch (Romera-Paredes et al., 2023) and AIDE (Jiang et al., 2025b) wrap an LLM in an iterative search/refinement loop. They treat the LLM as an agent that can branch, try multiple approaches, and refine code via feedback.

FunSearch (DeepMind, 2023) pairs a pre-trained code-generating LLM with an automated evaluator in a loop. The LLM proposes candidate programs/solutions, an evaluator (a test or objective function) checks them, and then the process generates new candidates (mutations, combinations) based on feedback. FunSearch features algorithm discovery based on a program database. The program database consists of a few "islands" of programs. The experimental setup is the same as (Li et al., 2025). The number of islands is set to 4 and the island reset period to 3600s. The FunSearch process runs for 32 iterations. In each iteration, the language model decoding temperature is set to 0.7.

AIDE (Weco AI, 2025) formulates code generation as a tree search problem. For a given high-level task (like "build an ML pipeline that achieves X accuracy on Y dataset"), AIDE would have the LLM propose a solution. Then it measures how good that solution is (it runs the code and sees accuracy). If not satisfied, AIDE can either refine the current solution (edit some parts of the code via another LLM call) or try a different approach (branch out in the search tree). Over multiple iterations, it explores the space of programs. The experimental setup is the same as (Li et al., 2025). AIDE runs for 96 steps and the max debug depth, debug probability, and number of drafts are set to 5, 0.9, and 24, respectively. The language model decoding temperature is set to 0.5 for code generation following the original paper (Jiang et al., 2025b).

A.3.3 Other Related Work

Recent work Soroco et al. (2025) introduces PDE-Controller, a framework that fine-tunes LLMs specifically for PDE control problems. Their approach trains specialized models for autoformalization (converting natural language to formal specifications), program synthesis, and multi-step reasoning through reinforcement learning from human feedback (RLHF). While demonstrating strong performance on their target domains, this approach differs from PDE-SHARP in several key aspects.

Table 6: PirateNets: reported relative ℓ_2 test errors across PDEs (paper's Table 1).

Benchmark	Error (PirateNet)	Params	GPU Mem	Time/epoch
Allen–Cahn (1D)	2.24×10^{-5}	—	—	—
Korteweg–De Vries (1D)	4.27×10^{-4}	—	—	—
Grey–Scott (2D)	3.61×10^{-3}	—	—	—
Ginzburg–Landau (2D)	1.49×10^{-2}	—	—	—
Lid-driven cavity (2D)	4.21×10^{-2}	—	—	—

Notes. Architecture details (e.g., depth/width) and training pipelines are provided, but resource metrics are not tabulated.

Table 7: PINNsFormer: model size and training overhead (Appendix Table 4–5 in the paper).

Model	Params	GPU Mem (MiB)	Time/epoch (s)
PINNsFormer (pseudo-seq. length $k=5$)	454,000	2,827	2.34

Notes. Reported on a single NVIDIA Tesla V100; overheads shown for $k=5$.

Table 8: PINNsFormer: reported test errors on 1D PDEs used widely in PINN literature.

PDE (dimension)	Metric (paper)	Error	Params	Time/epoch / GPU Mem
Convection (1D)	rRMSE ($\approx$ rel. ℓ_2)	0.027	454k	2.34 s / 2,827 MiB
Reaction (1D)	rRMSE ($\approx$ rel. ℓ_2)	0.030	454k	2.34 s / 2,827 MiB

Notes. Errors are taken directly from the paper's main results tables; rRMSE is the paper's standard relative ℓ_2 metric. The reaction/convection formulations and sampling follow the setups specified in Zhao et al. (2023).

While effective for specific classes of PDEs, the fine-tuning approach presents several limitations compared to LLM-driven approaches using test-time computing: **(1) Computational overhead:** Requires extensive fine-tuning of multiple specialized models (translator, controller, coder) with over 1.7M training samples; **(2) Domain specificity:** Limited to only heat and wave equations in 1D, requiring retraining for new PDE types; **(3) Data requirements:** Needs large-scale synthetic data generation and manual curation by domain experts; **(4) Scalability constraints:** Each new PDE family would require collecting new training data and retraining models; **(5) Generalization gap:** Performance drops significantly on manual problems (99.2% to 68.0% accuracy), indicating limited robustness to real-world variations.

PDE-SHARP offers more flexibility across PDE types without domain-specific training, though potentially at the cost of specialized performance on specific equation families. The fundamental trade-off lies between the specialized efficiency of fine-tuned approaches versus the broader applicability and reduced computational overhead of general prompting strategies.

A.3.4 OptiLLM

We use the OptiLLM framework from `github.com/codelion/optillm` as a baseline to test PDE-SHARP. OptiLLM is an optimizing inference proxy that implements 20+ state-of-the-art techniques to improve LLM accuracy and performance on reasoning tasks without requiring any model training or fine-tuning. We test three of OptiLLM's implemented techniques in our study.

CoT (Chain-of-Thought) with Reflection. Implements chain-of-thought reasoning with structured <thinking>, <reflection> and <output> sections to enhance reasoning quality through explicit self-evaluation. The approach generates intermediate reasoning steps in the thinking phase, critically reviews the reasoning in the reflection phase, and produces the final output, enabling improved accuracy on complex reasoning tasks without requiring model fine-tuning.

MoA (Mixture-of-Agents). Combines responses from multiple model critiques in a collaborative framework where 3 different agent perspectives are aggregated to produce higher-quality solutions.

CePO (Cerebras Planning and Optimization). Combines Best-of-n sampling (without code execution), Chain-of-Thought reasoning, Self-Reflection, and Self-Improvement in a four-stage process: plan generation with confidence scoring, initial solution development, plan refinement through inconsistency analysis, and final solution production.

Table 9: UPS: test errors on PDEBench benchmarks (relative ℓ_2 or nRMSE as reported).

PDE	Metric	Error (UPS)	Training Steps	GPU	Total GPU Hours
Advection (1D)	nRMSE	2.20×10^{-3}	60,000	A6000	¡100
Burgers (1D)	nRMSE	3.73×10^{-2}	60,000	A6000	¡100
Reaction–Diffusion (2D)	nRMSE	5.57×10^{-2}	60,000	A6000	¡100
Navier–Stokes (2D)	nRMSE	4.50×10^{-3}	60,000	A6000	¡100

Notes. Errors and training configuration are from the paper's PDEBench experiments. Training used $\sim$20k PDE samples across equations; GPU time and steps refer to total training, not per-PDE.

Table 10: LLM models used in Section 4 for solver generation; more LLMs – including the coding and math-aware variants of these – are tested in Appendix B.1

LLM	Type	Access
Gemma 3	Non-reasoning	Open Source
LLama 3.3	Non-reasoning	Open Source
Qwen3	Non-reasoning	Open Source
DeepSeek-R1	Reasoning	Open Source
GPT-4o	Non-reasoning	API Service
o3	Reasoning	API Service

Table 11: nRMSE values for Agentic Workflows on different PDEs. Results from Li et al. (2025)

	Advection	Burgers	Reaction-Diffusion	Navier-Stokes	Darcy
AIDE	1.03e-3	1.05e-4	5.07e-2	5.77e-2	4.78e-3
FunSearch	1.05e-3	1.13e-4	3.72e-2	5.86e-2	4.78e-3

The method applies Best-of-n to multiple solution candidates with optional plan diversity, using parameters like `planning_n` proposals and `planning_m` maximum attempts to generate robust solutions for complex reasoning tasks. The following are the default parameters used in this study.

A.3.5 CODEPDE

CodePDE. CodePDE (Li et al., 2025) is an inference framework for LLM-driven PDE solver generation that frames PDE solving as a code generation task. The framework operates through a five-step process: (1) *Task Specification* converts PDE problems into natural language descriptions including governing equations, domain specifications, boundary conditions, and initial conditions; (2) *Code Generation* uses chain-of-thought prompting to instruct models to generate complete solver implementations with predefined function signatures; (3) *Debugging* performs iterative self-debugging for up to 4 rounds when solvers encounter execution errors, feeding error traces back to the LLM for autonomous correction; and (4) *Evaluation* assesses solver performance using normalized root mean squared error (nRMSE), convergence tests, and execution time; For our comparison, we use CodePDE with the same setup as (Li et al., 2025) with steps 1-4 (reasoning + debugging), generating 32 solver samples with best-of-32 selection, using up to 4 debugging iterations per solver.

CodePDE-R. CodePDE-R extends the base CodePDE framework by incorporating the solver refinement step (step 5). This variant selects the 5 best-performing programs from the reasoning + debugging stage as "seed" programs for refinement. The refinement process provides the nRMSE obtained during evaluation along with the solver implementation back to the LLM, instructing it to analyze execution results, identify numerical instabilities and bottlenecks, and generate improved implementations accordingly. For each seed program, the framework generates 4 refined versions across different refinement configurations (using 3, 4, or 5 seed implementations), resulting in 12 refined programs total. The final result reports the best nRMSE among these 12 refined samples. This iterative feedback-driven optimization enables models to systematically improve solver accuracy and efficiency beyond the initial generation and debugging phases.

Table 12: PDE-Controller: Training Requirements and Performance

Metric	Value
Training Data	
Heat equation samples	867,408
Wave equation samples	845,088
Total training samples	1,712,496
Evaluation Data	
Synthetic test samples	426,432
Manual test problems	34
Performance (Synthetic)	
Autoformalization accuracy (IoU)	99.2%
Code executability	97.99%
Performance (Manual)	
Autoformalization accuracy (IoU)	68.0%
Code executability	91.2%
Scope	
PDE types covered	2 (heat, wave)
Spatial dimensions	1D

Table 13: Default configuration values for CePO planning and verification stages

Parameter	Description	Default Value
`--cepo_bestofn_n`	Number of responses to be generated in best of n stage	3
`--cepo_bestofn_temperature`	Temperature for verifier in best of n stage	0.1
`--cepo_bestofn_max_tokens`	Max tokens for verifier in best of n stage	4096
`--cepo_bestofn_rating_type`	Rating type ("absolute" or "pairwise")	`"absolute"`
`--cepo_planning_n`	Number of plans generated in planning stage	3
`--cepo_planning_m`	Attempts to generate n plans in planning stage	6
`--cepo_planning_temperature_step1`	Temperature in step 1 of planning stage	0.55
`--cepo_planning_temperature_step2`	Temperature in step 2 of planning stage	0.25
`--cepo_planning_temperature_step3`	Temperature in step 3 of planning stage	0.1
`--cepo_planning_temperature_step4`	Temperature in step 4 of planning stage	0
`--cepo_planning_max_tokens_step1`	Max tokens in step 1 of planning stage	4096
`--cepo_planning_max_tokens_step2`	Max tokens in step 2 of planning stage	4096
`--cepo_planning_max_tokens_step3`	Max tokens in step 3 of planning stage	4096
`--cepo_planning_max_tokens_step4`	Max tokens in step 4 of planning stage	4096
`--cepo_print_output`	Whether to print the output of each stage	`False`
`--cepo_config_file`	Path to CePO configuration file	`None`
`--cepo_use_plan_diversity`	Use additional plan diversity step	`False`
`--cepo_rating_model`	Rating model (if different from completion)	`None`

A.4 ADDITIONAL INFORMATION ON FRAMEWORK COST

Table 3 shows the average API call cost for each framework using GPT-4o as the code generator LLM. GPT-4o input cost is $2.50 per 1M tokens, and the output cost is $10.00 per 1M tokens. Table 14 shows the average input-output counts for each framework from Section 4. An NVIDIA T4 GPU costs $0.35 per hour, which is used to calculate the total average costs in Figure 8.

Table 14: Approximation of the total input-output counts for running each framework once

Framework	# Inputs	# Output
OptiLLM (CoT)	48,000	105,600
OptiLLM (MoA)	200,000	422,400
OptiLLM (CePO)	600,000	105,600
CodePDE	102,400	294,400
PDE-SHARP	600,000	450,800

B ADDITIONAL EXPERIMENTAL RESULTS

B.1 RESULTS WITH DIFFERENT LLMS

The following additional LLM models are tested for code generation in addition to the results of Table 2.

Table 15: Additional LLMs

LLM	Type	Access
Qwen3-Coder (Team, 2025)	Coding-specific	Open Source
Code Llama (Rozière et al., 2024)	Coding-specific	Open Source
GPT-5	Non-reasoning	API Service
DeepSeekMath (Shao et al., 2024)	Mathematical reasoning	Open Source
DeepSeek-Coder (Guo et al., 2024)	Coding-specific	Open Source
MathCoder2-DeepSeekMath (Lu et al., 2024)	Math aware Coding-specific	Open Source

B.2 PDE-SHARP ABLATION STUDIES

In this section, we present ablation study results on PDE-SHARP. Note that we take the default PDE-SHARP framework to be one used in Section 4. The ablation studies of this section each target a different aspect of PDE-SHARP's design.

B.2.1 ANALYSIS PROMPTING STRATEGY

We compare the following prompting strategies for the Analysis stage.

- Multi-Step prompting (PDE-SHARP default)
- Single Prompt (all the PDE-SHARP steps merged into one)
- LLM-generated multi-step prompting
- LLM-generated single prompt

For the LLM-generated alternatives, the LLM, GPT-4o in this ablation, is first asked to generate either a series of prompts or a single prompt to run as the analysis stage for a give PDE before proceeding to the code generation stage. The Synthesis stage is done exactly as in Section 4. Table 17 summarizes these results.

Our experiments demonstrate that the Multi-Step Prompting strategy consistently yields the best performance across all LLMs and PDEs. When all the PDE-SHARP Analysis prompts are merged together into a single prompt, LLMs tend to not follow the instructions thoroughly as they become too long to follow. Moreover, when the LLM is tasked with generating the prompts for the analysis stage, it is observed that many details, such as checking for hybrid approaches or doing a rigorous numerical stability analysis is overlooked. Analyzing the strategies used in the generated solvers (Table 17) for the reaction-diffusion task is a great demonstration of this shortcoming as reaction diffusion is more sensitive to method choice and stability analysis (Figure 17). Naturally, the most pronounced impact is observed on the Reaction-Diffusion PDE, where the default multi-step approach achieves the lowest average nRMSE of 2.88e-03 across all LLMs. In contrast, the average nRMSE rises to 6.88e-03 with Single Prompting, 4.30e-02 with LLM-Generated Multi-Step Prompting, and peaks at 7.86e-02 with LLM-Generated Single Prompting. This corresponds to a $27\times$ increase in error from the best case to the worst, highlighting the critical role of well-structured multi-step analysis in improving solution accuracy for complex PDEs.

Table 16: nRMSE comparison of the baseline frameworks using different LLMs.

		Advection	Burgers	Reaction-Diffusion	Navier-Stokes	Darcy
OptiLLM-CoT	Qwen3-Coder	4.67e-03	1.52e-03	9.38e-01	2.63e-01	6.34e-01
	GPT-5	5.36e-03	1.88e-03	1.04e+00	2.83e-01	7.18e-01
	DeepSeekMath	4.89e-03	3.12e-04	2.38e-01	8.51e-02	5.22e-03
	DeepSeek-Coder	4.89e-03	3.04e-04	2.41e-01	8.72e-02	5.11e-03
	MathCoder2-DeepSeekMath	4.89e-03	3.27e-04	2.43e-01	8.66e-02	5.29e-03
OptiLLM-MoA	Qwen3-Coder	1.01e-03	3.45e-04	9.68e-02	1.79e-02	5.12e-03
	GPT-5	4.18e-03	4.11e-04	1.14e-01	2.02e-02	1.89e-02
	DeepSeekMath	1.32e-03	2.66e-04	3.57e-02	1.72e-02	5.23e-03
	DeepSeek-Coder	1.32e-03	3.04e-04	1.55e-01	1.78e-02	5.18e-03
	MathCoder2-DeepSeekMath	1.01e-03	2.66e-04	4.07e-02	1.74e-02	5.22e-03
OptiLLM-CePO	Qwen3-Coder	1.01e-03	3.23e-04	8.91e-02	1.97e-02	1.83e-02
	GPT-5	3.17e-03	3.89e-04	1.03e-01	2.24e-02	4.72e-02
	DeepSeekMath	9.98e-04	2.55e-04	2.45e-02	1.85e-02	4.92e-03
	DeepSeek-Coder	1.01e-03	2.66e-04	1.47e-01	1.91e-02	4.92e-03
	MathCoder2-DeepSeekMath	9.98e-04	3.04e-04	3.56e-02	1.93e-02	4.33e-03
CodePDE	Qwen3-Coder	4.89e-03	1.35e-03	9.55e-01	2.59e-01	6.57e-01
	GPT-5	5.75e-03	1.63e-03	1.08e-01	2.82e-01	7.91e-01
	DeepSeekMath	5.10e-03	2.87e-04	2.45e-02	7.91e-02	4.97e-03
	DeepSeek-Coder	4.69e-03	2.87e-04	2.78e-01	7.82e-02	5.02e-03
	MathCoder2-DeepSeekMath	5.10e-03	3.15e-04	2.32e-02	7.84e-02	4.97e-03
CodePDE-R	Qwen3-Coder	9.74e-04	3.60e-04	9.13e-02	9.67e-02	4.90e-02
	GPT-5	1.14e-03	4.41e-04	1.07e-01	7.93e-02	5.81e-02
	DeepSeekMath	9.89e-04	2.62e-04	1.47e-02	3.63e-02	5.01e-03
	DeepSeek-Coder	9.89e-04	3.15e-04	1.47e-02	2.67e-02	6.01e-03
	MathCoder2-DeepSeekMath	9.74e-04	2.62e-04	1.47e-02	1.65e-02	4.97e-03
PDE-SHARP	Qwen3-Coder	9.74e-04	2.97e-04	5.39e-03	2.80e-02	7.80e-03
	GPT-5	1.01e-03	3.45e-04	7.78e-03	3.19e-02	9.93e-03
	DeepSeekMath	7.46e-04	1.55e-04	2.39e-03	1.47e-02	4.78e-03
	DeepSeek-Coder	7.46e-04	2.53e-04	3.67e-03	2.76e-02	4.78e-03
	MathCoder2-DeepSeekMath	5.54e-04	1.38e-04	2.99e-03	1.47e-02	3.93e-03

B.2.2 THE EFFECTS OF STABILITY ANALYSIS

To evaluate the individual contributions of PDE-SHARP's key components — the stability analysis in the Analysis stage and the tournaments in the Synthesis stage — we conduct an ablation study examining four variants: (1) the default framework with both mathematical stability analysis and tournaments, (2) tournaments without stability analysis, (3) stability analysis without tournaments (best-of-32 sampling with stability analysis), and (4) neither component (best-of-32 sampling without stability analysis). Figure 9 demonstrates that mathematical stability analysis provides substantial accuracy improvements across all tested PDEs. Removing stability analysis while maintaining tournaments increases average nRMSE by 2-8× depending on the PDE complexity. The tournaments component shows mixed but generally positive effects, with the largest improvements observed for reaction-diffusion and Darcy flow problems. Most critically, removing both components results in significant performance degradation, with nRMSE increases of 5-45× for complex PDEs like Darcy flow. These results confirm that PDE-SHARP's mathematical analysis stage is essential for generating numerically stable solvers, while the tournament-based refinement provides additional accuracy gains particularly for challenging nonlinear problems.

The stability analysis component of PDE-SHARP plays a crucial role in guiding solver strategy selection. Figure 18 illustrates the percentage of hybrid analytical-numerical versus purely numerical approaches chosen by each PDE-SHARP variant for the reaction-diffusion equation. The default framework and the variant without tournaments both achieve 100% hybrid approach selection, demonstrating that mathematical stability analysis consistently identifies the superiority of hybrid methods for this PDE. In contrast, removing stability analysis results in predominantly numerical approaches (87-93%), as the framework lacks the mathematical insight to recognize that the reaction component admits an analytical solution. This strategic difference directly explains the accuracy improvements observed in the previous ablation study, as hybrid approaches achieve superior numerical stability and precision for reaction-diffusion problems.

Table 17: nRMSE comparison of the baseline frameworks using different Analysis prompting strategies.

|		Advection	Burgers	Reaction-Diffusion	Navier-Stokes	Darcy
Multi-Step Prompting (Default)	Gemma 3	1.01e-03	5.60e-04	3.01e-03	3.14e-02	1.72e-02
	LLaMA 3.3	9.98e-04	4.61e-04	3.61e-03	5.06e-02	1.72e-02
	Qwen 3	7.76e-04	2.97e-04	2.32e-03	2.80e-02	4.80e-03
	DeepSeek-R1	5.24e-04	1.48e-04	2.29e-03	1.37e-02	4.74e-03
	GPT-4o	6.11e-04	2.31e-04	2.29e-03	1.51e-02	3.97e-03
	o3	9.74e-04	3.42e-04	5.78e-03	1.89e-02	7.78e-03
Single Prompt (Default merged into one)	Gemma 3	1.03e-03	4.89e-04	1.18e-02	4.31e-02	8.11e-03
	LLaMA 3.3	1.05e-03	4.79e-04	1.75e-02	7.32e-02	1.79e-02
	Qwen 3	8.01e-04	3.11e-04	2.41e-03	4.94e-02	4.91e-03
	DeepSeek-R1	6.53e-04	1.56e-04	2.37e-03	1.41e-02	4.83e-03
	GPT-4o	7.39e-04	3.48e-04	3.33e-03	2.62e-02	4.13e-03
	o3	8.70e-04	4.54e-04	3.89e-03	2.96e-02	4.87e-03
LLM-Generated Multi-Step Prompting	Gemma 3	1.02e-03	4.82e-04	9.21e-02	7.27e-02	7.93e-03
	LLaMA 3.3	1.04e-03	4.72e-04	8.69e-02	7.24e-02	1.77e-02
	Qwen 3	1.89e-03	6.05e-04	3.39e-02	3.89e-02	4.85e-03
	DeepSeek-R1	8.37e-04	5.30e-04	1.33e-02	3.40e-02	4.85e-03
	GPT-4o	7.27e-04	4.15e-04	1.31e-02	2.59e-02	4.05e-03
	o3	6.96e-04	7.48e-04	1.84e-02	3.93e-02	4.85e-03
LLM-Generated Single Prompt	Gemma 3	1.04e-03	4.95e-04	1.29e-01	5.42e-02	8.19e-03
	LLaMA 3.3	1.06e-03	6.87e-04	1.81e-01	6.43e-02	1.81e-02
	Qwen 3	1.13e-03	6.19e-04	8.47e-02	3.98e-02	3.95e-03
	DeepSeek-R1	9.59e-04	4.95e-04	1.39e-02	4.43e-02	4.85e-03
	GPT-4o	2.47e-03	7.22e-04	2.36e-02	3.65e-02	4.85e-03
	o3	9.19e-04	7.48e-04	3.91e-02	3.01e-02	5.92e-03
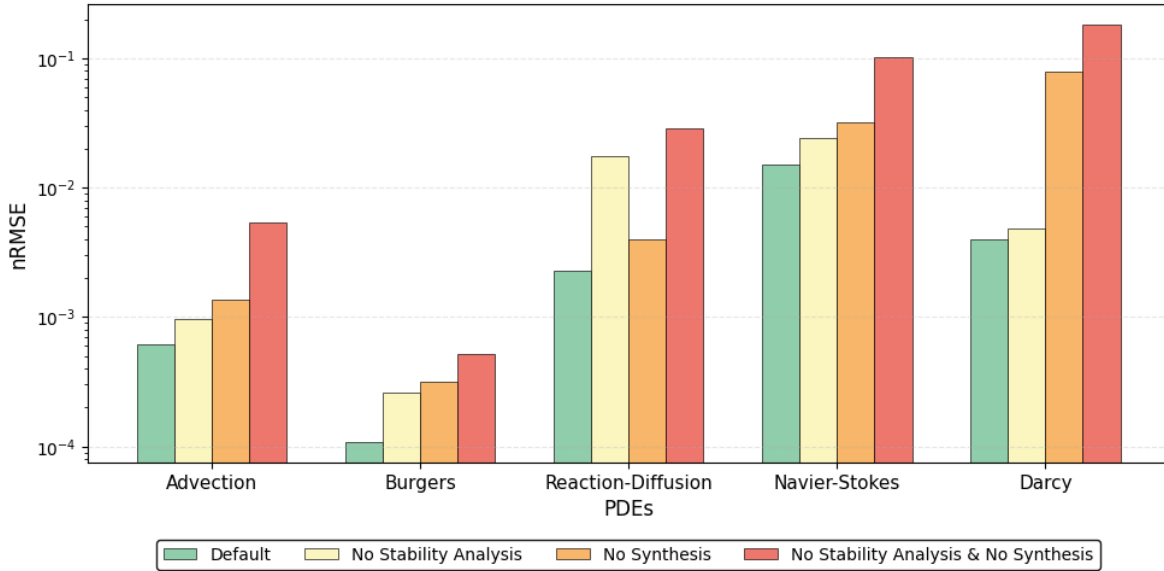
Figure 9: Ablation study of PDE-SHARP components across five PDE tasks. Results show that mathematical stability analysis is critical for solver accuracy, while tournaments provide additional improvements. Removing both components leads to significant performance degradation, particularly for complex PDEs like Darcy flow.

B.2.3 REASONING VS. NON-REASONING LLMS FOR CODE GENERATION IN GENESIS

Experiments indicate that in PDE-SHARP, there is negligible difference between the final results using reasoning, non-reasoning, coding-specific, and mathematical LLM models (Tables 10 & 15) as the code generator in the Genesis stage. See Tables 2 and 16 for nRMSE results.

B.2.4 TEST-TIME SCALING FOR PDE-SHARP

Based on our test-time scaling study (Figure 10) for PDE-SHARP and to be consistent with findings from (Li et al., 2025) on the same PDE tasks, we use $n = 32$ initial solver candidates in our experiments. This choice balances computational efficiency with sufficient diversity for effective solver selection in the subsequent Synthesis stage.
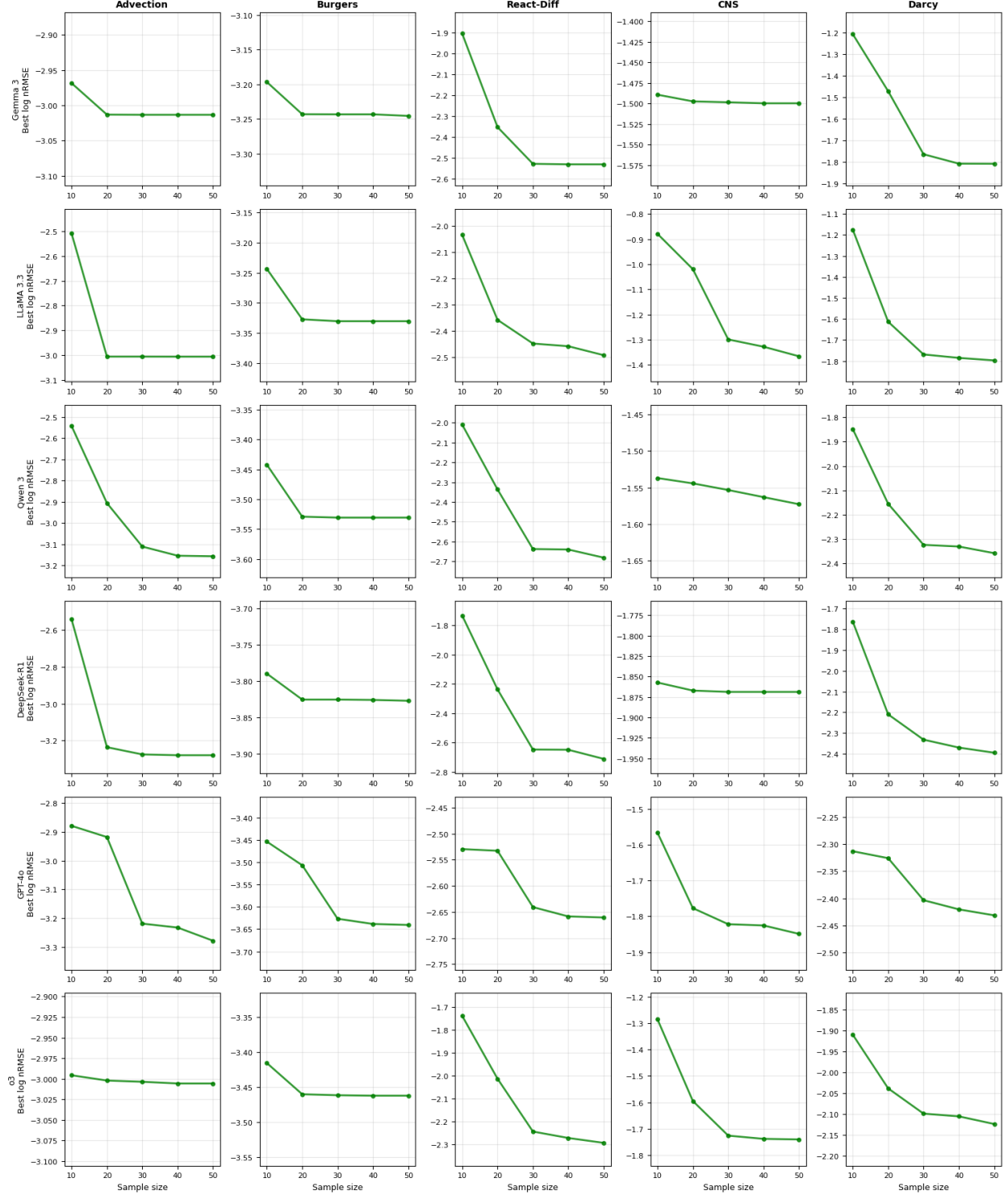

Figure 10: Varying the number of solver samples generated for each LLM and each PDE family in PDE-SHARP.

B.2.5 STRUCTURE OF THE TOURNAMENTS

In this ablation study, we keep the default PDE-SHARP strategy from Section 4 for the Analysis and Genesis stages and replace the Synthesis stage with various strategies to study its effectiveness. In PDE-SHARP's default Synthesis stage in Section 4, three LLM instances, which we call "judges", are tasked with the selection and hybridization tournaments. To achieve the best performance (Table 2) — i.e. fewer tournament rounds to get the highest performing PDE solver codes — these three judges are taken to be a mixture of reasoning and non-reasoning LLMs (o3, DeepSeek-R1, and GPT-4o) in Section 4. This set of LLM judges are chosen to balance efficient code generation and code stability details with the detailed reasoning and attention to numerical implementation details that the reasoning models bring in. In this section, we consider other possibilities for the three judges to justify our choice of LLM judges. Tables 2 and 16 demonstrate that using different LLM models to generate 32 samples of solver codes leads to overall negligible difference in the final results in PDE-SHARP as the tournaments lead to solvers robust to LLM choice. Thus, we stick to the default GPT-4o for code generation in this ablation study and use the same 32 samples generated by GPT-4o for all of the stage 3 strategies studied. Note that in these tournaments, feedback type is set to be nRMSE similar to Section 4. Results for different feedback types are presented later in this section. Since numerous LLM configurations exist, we select a minimal representative subset from each category. Current models have sufficient input capacity for tournament solver lists; future work could incorporate summarizer agents to compress information for smaller models.

We test six tournament structure categories:

1. Mixed Judges (Default): Combines reasoning and non-reasoning models to balance code generation efficiency with detailed numerical reasoning:

- o3 + GPT-4o + DeepSeek-R1 (Section 4 default)
- o3 + GPT-4o + GPT-4o
- DeepSeek-R1 + GPT-4o + GPT-4o

2. All Reasoning Judges: Uses only reasoning-capable models:

- o3 + o3 + o3
- DeepSeek-R1 + DeepSeek-R1 + DeepSeek-R1
- o3 + o3 + DeepSeek-R1

3. All Non-Reasoning Judges: Uses only standard language models:

- GPT-4o + GPT-4o + GPT-4o

4. Best-of-32 Baseline: Executes all 32 solvers from Analysis and Genesis stages without tournaments.

5. Fixed Criteria Judging: Applies categories 1-3 with predetermined evaluation criteria:

- Numerical stability and convergence properties
- Computational efficiency and scalability
- Mathematical correctness and precision
- Implementation robustness and error handling
- Solution accuracy on benchmark problems

6. Self-Generated Criteria: Applies categories 1-3 where judges first generate their own evaluation criteria before selection.

All strategies use identical 32 solver samples from GPT-4o code generation to ensure fair comparison.

Table 18: nRMSE values for each PDE-SHARP using different LLM combinations for the Synthesis stage.

		Advection	Burgers	Reaction-Diffusion	Navier-Stokes	Darcy
Mixed Judges (Default)	o3 + GPT-4o + DeepSeek-R1	6.11e-04	2.31e-04	2.29e-03	1.51e-02	3.97e-03
	o3 + GPT-4o + GPT-4o	7.34e-04	4.45e-04	5.41e-03	3.58e-02	4.12e-03
	DeepSeek-R1 + GPT-4o + GPT-4o	6.98e-04	2.31e-04	4.33e-03	1.51e-02	4.91e-03
All Reasoning	o3 + o3 + o3	9.74e-04	5.19e-04	4.21e-03	3.45e-02	3.84e-03
	DeepSeek-R1 + DeepSeek-R1 + DeepSeek-R1	8.92e-04	3.23e-04	3.25e-03	2.47e-02	3.84e-03
	o3 + o3 + DeepSeek-R1	7.79e-04	2.35e-04	4.33e-03	1.51e-02	3.97e-03
All Non-Reasoning	GPT-4o + GPT-4o + GPT-4o	9.74e-04	2.57e-04	1.01e-02	2.62e-02	4.90e-03
Best-of-32 Baseline	No Tournaments	1.35e-03	3.19e-04	3.99e-03	3.18e-02	7.82e-02
Fixed Criteria - Mixed Judges	o3 + GPT-4o + DeepSeek-R1	9.86e-04	5.25e-04	7.24e-03	1.48e-02	3.89e-03
	o3 + GPT-4o + GPT-4o	9.18e-04	2.38e-04	2.36e-02	1.54e-02	4.05e-03
	DeepSeek-R1 + GPT-4o + GPT-4o	1.01e-03	2.21e-04	8.27e-03	1.46e-02	3.85e-03
Fixed Criteria - All Reasoning	o3 + o3 + o3	1.73e-03	6.11e-04	1.15e-02	1.41e-02	7.76e-03
	DeepSeek-R1 + DeepSeek-R1 + DeepSeek-R1	9.74e-04	3.17e-04	3.19e-03	1.44e-02	3.82e-03
	o3 + o3 + DeepSeek-R1	1.68e-03	2.08e-04	1.12e-02	2.89e-02	3.73e-03
Fixed Criteria - All Non-Reasoning	GPT-4o + GPT-4o + GPT-4o	1.01e-03	3.43e-04	2.42e-03	9.29e-02	5.01e-03
Self-Generated Criteria - Mixed Judges	o3 + GPT-4o + DeepSeek-R1	8.12e-04	4.67e-04	9.15e-03	1.62e-02	4.21e-03
	o3 + GPT-4o + GPT-4o	8.53e-04	3.02e-04	1.89e-02	1.38e-02	4.57e-03
	DeepSeek-R1 + GPT-4o + GPT-4o	1.15e-03	2.94e-04	6.83e-03	1.71e-02	3.42e-03
Self-Generated Criteria - All Reasoning	o3 + o3 + o3	1.58e-03	7.24e-04	1.38e-02	1.27e-02	8.35e-03
	DeepSeek-R1 + DeepSeek-R1 + DeepSeek-R1	9.13e-04	2.85e-04	4.06e-03	1.59e-02	4.18e-03
	o3 + o3 + DeepSeek-R1	1.52e-03	2.76e-04	9.84e-03	2.53e-02	4.29e-03
Self-Generated Criteria - All Non-Reasoning	GPT-4o + GPT-4o + GPT-4o	9.27e-04	3.89e-04	3.17e-03	8.46e-02	5.68e-03

Table 19: Number of rounds to achieve the results of Table 18 for each PDE-SHARP using different LLM combinations for the Synthesis stage. The number of rounds is reported before performance saturation/degradation, indicating the minimum number of hybridization rounds. The "+" sign indicates a rejudging cycle as explained in Table 20. Note that no hybrid tournaments accur in the best-of-32 strategy.

|		Advection	Burgers	Reaction-Diffusion	Navier-Stokes	Darcy
Mixed Judges (Default)	o3 + GPT-4o + DeepSeek-R1	4+4	3	4	3	4
	o3 + GPT-4o + GPT-4o	4+2	4	4+1	4	4+1
	DeepSeek-R1 + GPT-4o + GPT-4o	4+3	3	4	4+1	4
All Reasoning	o3 + o3 + o3	4+4	3	3	3	3
	DeepSeek-R1 + DeepSeek-R1 + DeepSeek-R1	4+3	3	4	4	4
	o3 + o3 + DeepSeek-R1	4+3	3	3	3	3
All Non-Reasoning	GPT-4o + GPT-4o + GPT-4o	4+4	4+2	4+2	4+4+2	4+3
Best-of-32 Baseline	No Tournaments	-	-	-	-	-
Fixed Criteria - Mixed Judges	o3 + GPT-4o + DeepSeek-R1	4+3	3	3	3	3
	o3 + GPT-4o + GPT-4o	4+2	4	4	4	4
	DeepSeek-R1 + GPT-4o + GPT-4o	4+3	3	3	4	4
Fixed Criteria - All Reasoning	o3 + o3 + o3	4+3	3	3	3	3
	DeepSeek-R1 + DeepSeek-R1 + DeepSeek-R1	4+2	3	3	3	4
	o3 + o3 + DeepSeek-R1	3	3	3	3	3
Fixed Criteria - All Non-Reasoning	GPT-4o + GPT-4o + GPT-4o	4+2	4+1	4+1	4+3	4+2
Self-Generated Criteria - Mixed Judges	o3 + GPT-4o + DeepSeek-R1	4+3	4	4	4	4+1
	o3 + GPT-4o + GPT-4o	4+3	4+1	4+2	4+2	4+2
	DeepSeek-R1 + GPT-4o + GPT-4o	4+3	4	4	4+1	4
Self-Generated Criteria - All Reasoning	o3 + o3 + o3	4+4	3	4	3	4
	DeepSeek-R1 + DeepSeek-R1 + DeepSeek-R1	4+2	4	4	4	4
	o3 + o3 + DeepSeek-R1	4+1	3	4	3	4
Self-Generated Criteria - All Non-Reasoning	GPT-4o + GPT-4o + GPT-4o	4+4	4+3	4+3	4+4+3	4+4
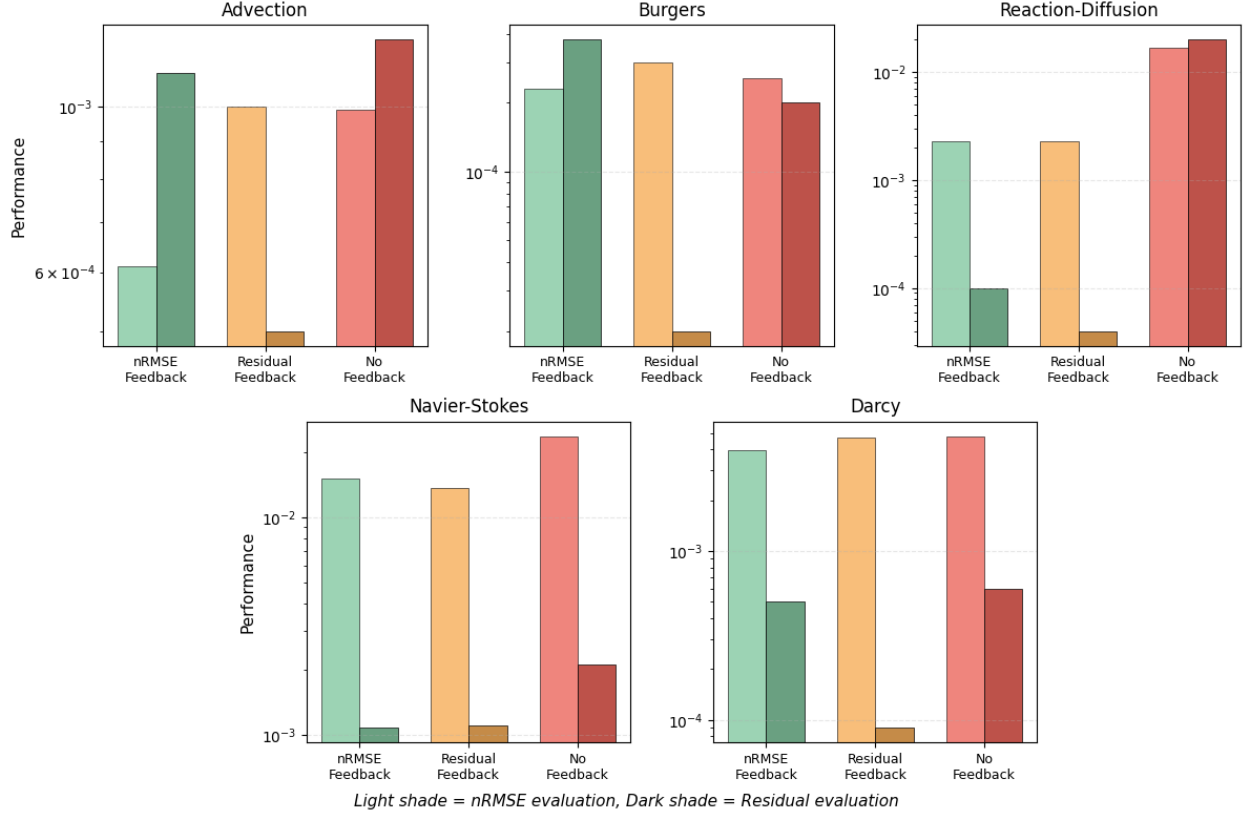
Figure 11: Impact of feedback type on PDE-SHARP solver accuracy across five PDE tasks. Performance is measured using both nRMSE (light bars) and residual evaluation (dark bars) metrics. nRMSE feedback consistently achieves superior performance when evaluated on the nRMSE metric, demonstrating the importance of alignment between feedback type and evaluation criteria. Residual feedback provides a physics-informed alternative when reference solutions are unavailable, while no feedback relies purely on judge code analysis. The choice of feedback type allows adaptation to different research scenarios from benchmark validation to real-world cases with limited reference data.

Remark: LLM-suggested Feedback Types.   In this part of the section on feedback types, we provide examples of LLM-suggested feedback for each of the tested PDE tasks. The results are generated using GPT-4o as follows.

(1) Advection: $\partial_t u + \beta\,\partial_x u = 0$ (periodic; β constant)

General feedback types:

- **nRMSE**
- **PDE residual L^2**: $\|r\|_2$ with $r := \partial_t u + \beta\,\partial_x u$, discretized consistently with the scheme.
- **BC/IC mismatch**: $\|u(t_0,\cdot) - u_0(\cdot)\|_2$, and periodic-wrap mismatch at boundaries.
- **Empirical convergence order** p via two grids $(h, h/2)$:

$$p \approx \frac{\log\big(E(h)/E(h/2)\big)}{\log 2}.$$

- **CFL ratio monitor**:

$$\mathrm{CFL}_{\max} = \max_x \frac{|\beta|\,\Delta t}{\Delta x}$$

(used as a stability penalty when $>$ target).

PDE-specific feedback types:

- **Phase-error (Fourier) metric** — detects dispersive drift from exact shift:
  For any wavenumber k, let $\hat{u}_k(t)$ be the DFT of $u(\cdot, t)$. The analytic evolution is

$$\hat{u}_k(t) = \hat{u}_k(0) \, e^{-ik\beta t}.$$

  Define

$$\epsilon_{\text{phase}}(t) = \left(\sum_{k \in \mathcal{K}} w_k \left| \arg \hat{u}_k(t) - \arg\left(\hat{u}_k(0) \, e^{-ik\beta t} \right) \right|^2 \right)^{1/2}.$$

  (Choose $\mathcal{K} =$ dominant modes; w_k normalize by spectral energy.)
  Why: linear advection is phase-exact; any phase drift degrades solution even when L^2 error is small.

- **Amplitude-damping metric** — detects artificial diffusion:

$$\epsilon_{\text{amp}}(t) = \left(\sum_{k \in \mathcal{K}} w_k \big| |\hat{u}_k(t)| - |\hat{u}_k(0)| \big|^2 \right)^{1/2}.$$

  Why: upwinding or overly diffusive fluxes damp modes; useful when the reference data were generated by a specific finite-volume scheme and you want to "match" it. (This is exactly what happened in your advection case study where nRMSE feedback nudged judges toward a MUSCL/TVD FV scheme instead of an analytical shifter.)

- **Invariant-conservation drift** — detects systematic bias:
  Mass and L^2 are constant for periodic, constant-β advection:

$$\delta_{\text{mass}}(t) = \frac{\left| \int_0^1 u(x, t) \, dx - \int_0^1 u_0(x) \, dx \right|}{\left| \int_0^1 u_0(x) \, dx \right|}, \qquad \delta_{L^2}(t) = \frac{\|u(\cdot, t)\|_2 - \|u_0\|_2}{\|u_0\|_2}.$$

  Why: catches subtle dissipation or numerical pumping even when nRMSE is small.

(2) Burgers: $\quad \partial_t u + \partial_x(u^2/2) = \nu \, \partial_{xx} u \quad$ (periodic; $\nu = 0.01$)

General feedback types:

- **nRMSE, PDE residual L^2** with $r := \partial_t u + \partial_x(u^2/2) - \nu \partial_{xx} u$.
- **Convergence order** p (as above).
- **Max CFL monitor** with characteristic speed $\lambda_{\max} = |u|_\infty \cdot \frac{\Delta t}{\Delta x}$.
- **Boundary/periodicity mismatch**.

PDE-specific feedback types:

- **Entropy inequality violation (integrated)** — penalizes non-admissible shocks/oscillations:
  With entropy $\eta(u) = \frac{1}{2}u^2$, viscous Burgers satisfies:

$$\frac{d}{dt} \int_0^1 \tfrac{1}{2} u^2 \, dx = -\nu \int_0^1 (\partial_x u)^2 \, dx \ \leq \ 0.$$

  Define

$$\Phi_{\text{entropy}} = \sum_n \max \left(0, \int_0^1 \tfrac{1}{2} u^2(x, t_{n+1}) \, dx - \int_0^1 \tfrac{1}{2} u^2(x, t_n) \, dx \right).$$

  Why: any net increase flags spurious energy injection near steep gradients.

- **Total variation (TV) growth** — damps Gibbs and enforces TVD behavior:

$$\mathrm{TV}(u) = \sum_j |u_{j+1} - u_j|, \qquad \Phi_{\mathrm{TV}} = \sum_n \max\left(0, \, \mathrm{TV}(u^{n+1}) - \mathrm{TV}(u^n)\right).$$

Why: shocks should not create oscillations; TV growth is a crisp signal.

- **Mean (mass) conservation drift** — periodic Burgers conserves $\int u \, dx$:

$$\delta_{\mathrm{mean}}(t) = \frac{\left| \int_0^1 u(x,t) \, dx - \int_0^1 u_0(x) \, dx \right|}{\left| \int_0^1 u_0(x) \, dx \right|}.$$

Why: catches subtle bias from asymmetric limiters or boundary handling.

(3) Reaction–Diffusion (Fisher–KPP form): $\quad \partial_t u - \nu \partial_{xx} u - \rho \, u(1-u) = 0 \quad$ (periodic; $\nu = 0.5$, $\rho = 1$)

General feedback types:

- **nRMSE, PDE residual** L^2 with $r := \partial_t u - \nu \partial_{xx} u - \rho u(1-u)$.
- **Convergence order** p.
- **Diffusive CFL monitor** (for explicit pieces): $\max \dfrac{\nu \Delta t}{\Delta x^2}$.

PDE-specific feedback types:

- **Maximum-principle / positivity violation** — enforces physically meaningful range:
  For logistic reaction, the continuous solution stays in $[0,1]$ when $u_0 \in [0,1]$. Define

$$\Phi_{\mathrm{MP}} = \left(\int_0^1 \left(\max(0,-u)\right)^2 dx \right)^{1/2} + \left(\int_0^1 \left(\max(0,u-1)\right)^2 dx \right)^{1/2}.$$

Why: catches overshoot/undershoot from aggressive time steps or limiters.

- **Split-step (hybrid) consistency error** — encourages the analytically-integrated reaction that your analysis stage favors:
  If Strang/IMEX or analytical-reaction is used, compare the reaction sub-update to the exact ODE update:

$$R_{\Delta t}(u) = \frac{u \, e^{\rho \Delta t}}{1 + u \left(e^{\rho \Delta t} - 1\right)}.$$

Define $\varepsilon_{\mathrm{react}} = \|u^{n+\frac{1}{2}} - R_{\Delta t}(u^n)\|_2$ (or analogous placement per scheme).
Why: rewards the hybrid analytical–numerical strategy your framework discovers for this PDE.

- **Stiffness-aware step safety** — keeps reaction eigenvalue under control for explicit parts:
  Spectral radius for reaction $J = \rho(1-2u) \Rightarrow |\rho(J)| \leq \rho$. Penalize $\max_n \max_x \dfrac{\Delta t \, \rho}{\rho_{\mathrm{exact}}} > 1$.
  Why: prevents overshoot/explosions when reaction is treated explicitly.

(4) Compressible Navier–Stokes ($\Gamma = 5/3$):

$$\partial_t \rho + \partial_x (\rho v) = 0,$$

$$\rho \left(\partial_t v + v \partial_x v \right) = -\partial_x p + \eta \, \partial_x^2 v + \left(\zeta + \frac{\eta}{3} \right) \partial_x (\partial_x v),$$

$$\partial_t \left(\epsilon + \frac{\rho v^2}{2} \right) + \partial_x \left[\left(\epsilon + p + \frac{\rho v^2}{2} \right) v - v \, \sigma' \right] = 0, \quad \epsilon = \frac{p}{\Gamma - 1}, \quad \sigma' = \left(\zeta + \frac{4}{3} \eta \right) \partial_x v.$$

General feedback types:

- **nRMSE** on chosen state(s) (ρ, v, p, or conservative variables).

- **Vector PDE residual** (mass, momentum, energy) in normalized L^2 (sum of per-equation residual norms).

- **Convergence order** p.

- **Maximum acoustic CFL:**

$$\max \frac{(|v| + c)\Delta t}{\Delta x}, \qquad c = \sqrt{\Gamma p/\rho}.$$

- **BC/periodicity mismatch**.

PDE-specific feedback types:

- **Conservation-law drift** — ensures discrete conservation:

$$\delta_{\text{mass}}(t) = \frac{\left| \int \rho(x,t)\, dx - \int \rho(x,0)\, dx \right|}{\int \rho(x,0)\, dx}, \qquad \delta_{\text{mom}}(t) = \frac{\left| \int \rho v\, dx - \int \rho_0 v_0\, dx \right|}{\int |\rho_0 v_0|\, dx},$$

$$\delta_{\text{energy}}(t) = \frac{\left| \int \left(\epsilon + \frac{\rho v^2}{2} \right) dx - \int \left(\epsilon_0 + \frac{\rho_0 v_0^2}{2} \right) dx \right|}{\int \left(\epsilon_0 + \frac{\rho_0 v_0^2}{2} \right) dx}.$$

Why: small global drifts reveal flux/boundary inconsistencies even if pointwise errors look OK.

- **Positivity violations** — hard physical constraints:

$$\Phi_{\rho,p} = \| \min(0,\rho) \|_1 + \| \min(0,p) \|_1.$$

Why: avoids catastrophic instabilities (negative density/pressure).

- **Entropy production sign check** — flags nonphysical dissipation/oscillations:
For ideal gas, specific entropy $s = \ln(p) - \Gamma \ln(\rho)$. Define

$$\sigma(t) = \int \rho s\, dx, \qquad \Phi_{\text{entropy}} = \sum_n \max(0, -(\sigma^{n+1} - \sigma^n)).$$

Why: with viscosity, total entropy should not decrease; negative production indicates spurious behavior.

- **Rankine–Hugoniot defect (interface balance)** — shock-consistency check in conservative form:
For each interface $i + \frac{1}{2}$ and conserved vector $U = (\rho, \rho v, E)$, flux $\mathbf{F}$, penalize the discrete jump

$$\Phi_{\text{RH}} = \sum_{n,i} \left\| \frac{U_i^{n+1} - U_i^n}{\Delta t} + \frac{F_{i+\frac{1}{2}}^n - F_{i-\frac{1}{2}}^n}{\Delta x} \right\|_1.$$

Why: targets the exact property your solver should satisfy at shocks/contacts.

(5) Darcy flow (steady, Dirichlet): $\quad -\nabla \cdot (a(x)\nabla u) = \beta, \quad u|_{\partial\Omega} = 0$

General feedback types:

- **PDE residual norms at steady state:**

$$\|r\|_2 = \|\beta + \nabla \cdot (a\nabla u_h)\|_{L^2(\Omega)}.$$

- **Boundary condition residual:** $\|u_h\|_{L^2(\partial\Omega)}$ (often ≈ 0 if enforced strongly; still useful with FV).

- **Grid-refinement check** using energy-norm proxy below.

PDE-specific feedback types:

- **Residual-jump a-posteriori estimator (energy-norm surrogate)** — standard for elliptics; localizes errors cheaply:
  For each cell K with diameter h_K,

  $$r_K = \beta + \nabla \cdot (a \nabla u_h)\big|_K, \qquad J_e = [\![a \nabla u_h \cdot n_e]\!] \text{ on edge } e,$$

  $$\eta^2 = \sum_K \left(h_K^2 \|r_K\|_{L^2(K)}^2 + \sum_{e \subset \partial K} h_e \|J_e\|_{L^2(e)}^2 \right).$$

  Why: mirrors FE error estimators; correlates with the true a-energy error without ground truth.

- **Local mass balance (cell-wise)** — ensures flux consistency:

  $$\Phi_{\text{mass}} = \sum_K \left| \int_K \beta \, dx + \int_{\partial K} (a \nabla u_h) \cdot n \, ds \right|.$$

  Why: FV/FD/FE schemes should balance source with flux divergence on each control volume.

- **Global compatibility check** — sanity for data/boundary handling:

  $$\left| \int_\Omega \beta \, dx + \int_{\partial \Omega} (a \nabla u_h) \cdot n \, ds \right|.$$

  Why: catches solver or BC mishandling even when $\|r\|_2$ looks small.

To determine the optimal number of hybridization rounds and rejudging cycles, we conduct an analysis tracking solver accuracy improvements across eight total rounds (four initial hybridization rounds plus four rejudging cycle rounds) for all tested PDEs. Figure 12 demonstrates the round-by-round progression of best achieved nRMSE in that round (among the tested three), with a vertical dashed line separating the initial hybridization cycle from the rejudging cycle.

The results reveal different patterns across different PDE types. Most PDEs achieve optimal performance within 3-4 initial hybridization rounds, after which additional rounds provide saturation or even slight performance degradation. Advection presents a notable exception, continuing to benefit from one rejudging cycle. This stems from a dataset-specific subtlety: while analytical solutions exist for the mathematical advection equation, the PDEBench reference solutions were generated using finite-volume methods. The rejudging cycle enables PDE-SHARP to adapt from initially favoring analytical approaches to numerical methods that better match the dataset's characteristics. This mostly occurs when the feedback type is set to be nRMSE in the tournaments. See Figure 16 for results using other feedback types (residual feedback, no feedback) for the advection PDE.
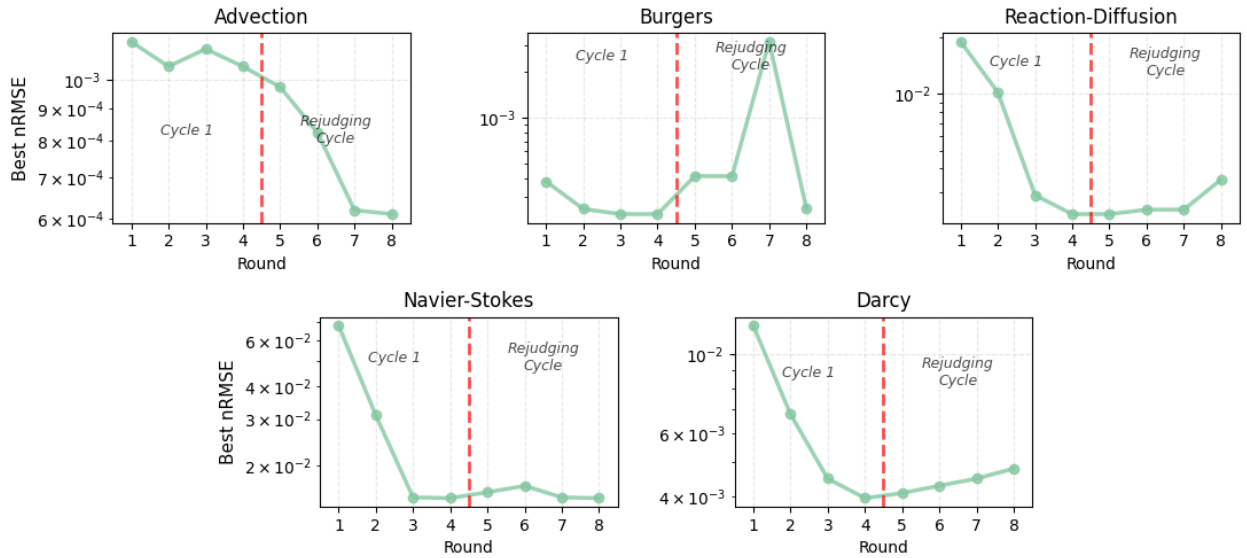
Vertical dashed line separates initial hybridization rounds from rejudging cycle

Figure 12: Progression of the best nRMSE of each hybridization round for each PDE task

Table 20: Average number of **Hybridization Rounds**, **Rejudging Cycles**, and total evaluations

PDE	# Hybrid. Rounds	# Rejudging Cycles	# Total Evals
Advection	4 + 4	1	24
Burgers	3	0	9
Reaction-Diffusion	4	0	12
Navier-Stokes	3	0	9
Darcy	4	0	12

For four out of five tested PDEs, PDE-SHARP achieves optimal results using fewer than 13 solver evaluations on average (Table 20), with most improvement occurring in the initial 3-4 rounds, resulting in a computational advantage over baseline methods requiring 30+ evaluations, while the rejudging cycle provides additional benefits only for specific cases.

B.3 ANALYSIS OF THE GENERATED SOLVER CODE QUALITY

Beyond solution accuracy, we analyze the computational and numerical properties of generated solver code across all methods. This analysis examines three key quality indicators: execution time efficiency, library usage, and empirical

convergence rates. These metrics reveal whether frameworks generate production-ready code with proper numerical characteristics, not merely code that produces correct outputs through inefficient or unstable implementations.
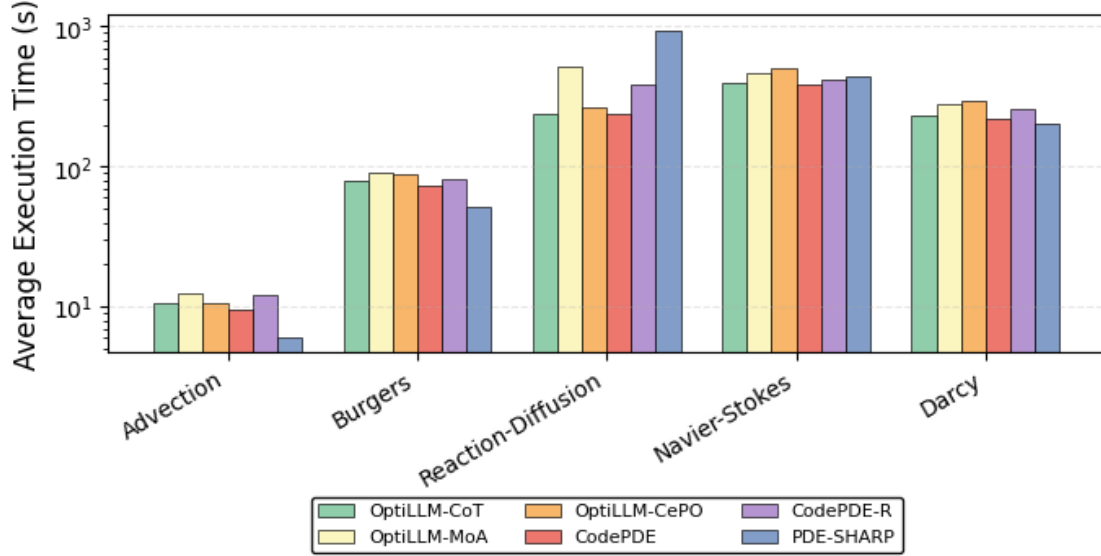

Figure 13: Average execution times across PDE tasks. PDE-SHARP achieves lower execution times than the average baseline in 4/5 cases. For reaction-diffusion, higher execution time reflects the rigorous numerical methods selected by stability analysis as expected, which produce significantly higher accuracy solvers (Table 2).

PDE	Method	SciPy	JAX	NumPy	PyTorch
Advection	PDE-SHARP	10%	17%	48%	25%
Burgers	PDE-SHARP	10%	32%	25%	33%
Reaction-Diffusion	PDE-SHARP	8%	1%	49%	25%
Comp. Navier-Stokes	PDE-SHARP	7%	37%	30%	26%
Darcy	PDE-SHARP	43%	15%	15%	27%

Table 21: PDE-SHARP decreases Python usage and increased JAX + SciPy usage overall across all tested PDEs
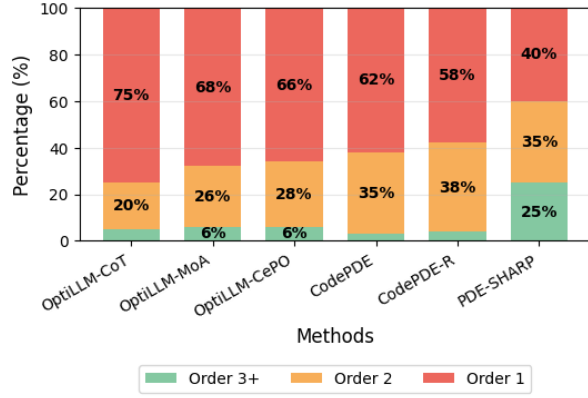
(a) Burgers
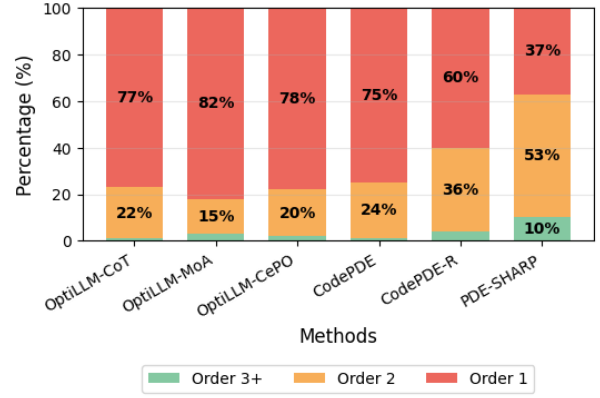
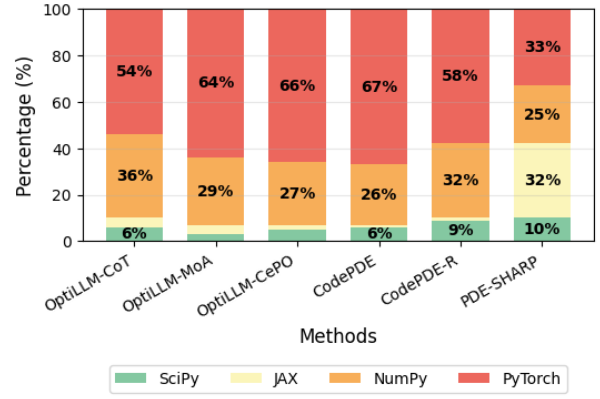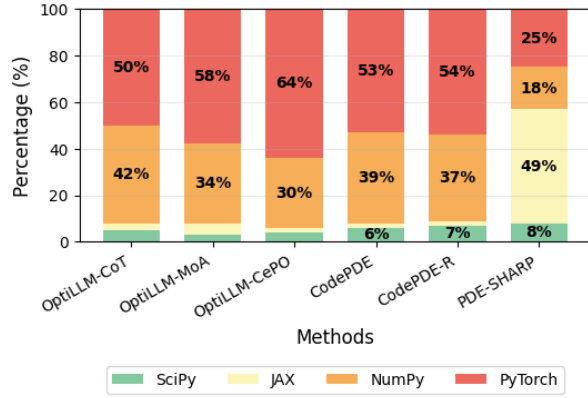(b) Reaction-Diffusion
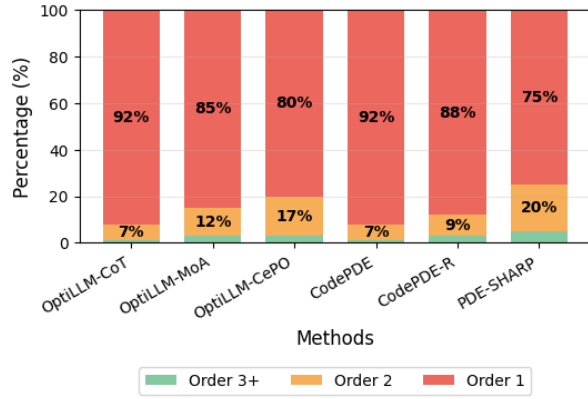
(c) Navier-Stokes
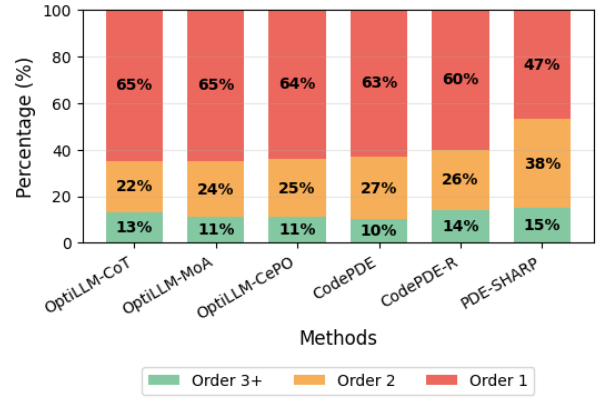
(d) Darcy

Figure 14: Convergence order distribution across different PDEs. The convergence order distribution for the advection PDE appears in Figure 4b.
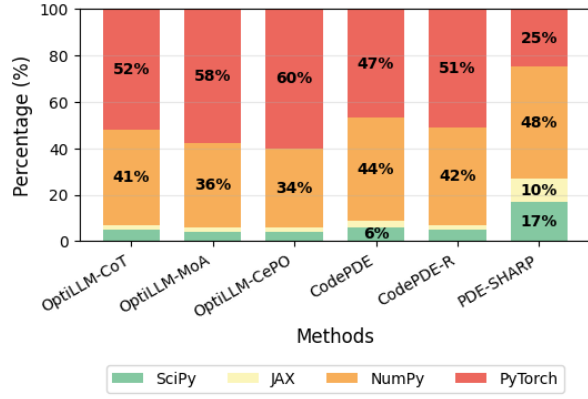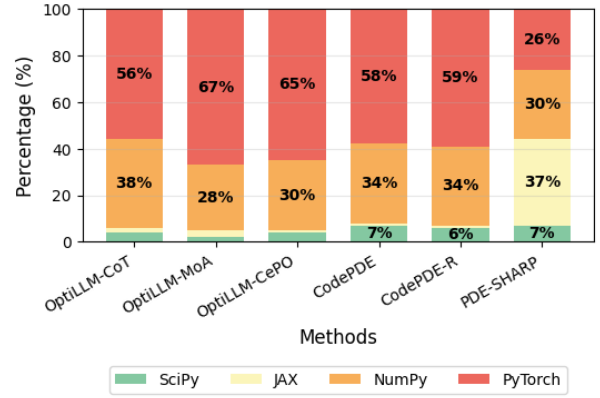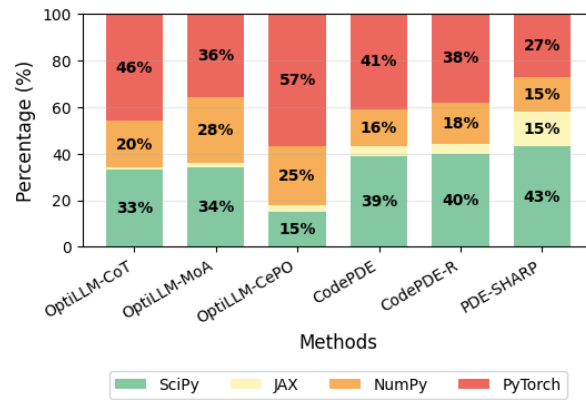
(a) Advection

(b) Burgers

(c) Reaction-Diffusion

(d) Compressible Navier-Stokes

(e) Darcy

Figure 15: Solver library usage across different PDEs.

C   ADDITIONAL DETAILS ON THE TESTED PDEs

In this section of the appendix, we present the differential equations we study in our experiments.

C.1   ADVECTION

The 1D advection equation is a hyperbolic PDE which models processes such as fluid flow, heat transfer, and biological dynamics. It is given by

$$\begin{cases} \partial_t u(t,x) + \beta \partial_x u(t,x) = 0, & x \in (0,1), \ t \in (0,2] \\ u(0,x) = u_0(x), & x \in (0,1) \end{cases}$$

where β is a constant representing the advection speed. In our experiments, we assume the periodic boundary condition and report results for the $\beta = 0.1$ case using the advection dataset from PDEBench.

C.2   BURGERS

The Burgers equation, a fundamental PDE in fluid mechanics, is used to model various nonlinear phenomena including shock waves and traffic flow. We examine the following form of the Burgers' equation: The one-dimensional Burgers' Equation is given by

$$\begin{cases} \partial_t u(x,t) + \partial_x \left(\frac{u^2(x,t)}{2} \right) = \nu \partial_{xx} u(x,t), & x \in (0,1), \ t \in (0,1] \\ u(x,0) = u_0(x), & x \in (0,1) \end{cases}$$

where ν is a constant representing the viscosity. In our experiments, we assume the periodic boundary condition and report results for the $\nu = 0.01$ case using the Burgers dataset from PDEBench.

C.3   REACTION-DIFFUSION

The 1D reaction-diffusion PDE is given by

$$\begin{cases} \partial_t u(t,x) - \nu \partial_{xx} u(t,x) - \rho u(1-u) = 0, & x \in (0,1), \ t \in (0,T] \\ u(0,x) = u_0(x), & x \in (0,1) \end{cases}$$

where ν and ρ are coefficients representing diffusion and reaction terms, respectively. In our experiments, we assume the periodic boundary condition and report results for the $\nu = 0.5$ and $\rho = 1.0$ case using the reaction-diffusion dataset from PDEBench.

C.4   NAVIER-STOKES

The compressible Navier-Stokes equations are given by

$$\begin{cases} \partial_t \rho + \partial_x (\rho v) = 0 \\ \rho(\partial_t v + v \partial_x v) = -\partial_x p + \eta \partial_{xx} v + (\zeta + \eta/3) \partial_x (\partial_x v) \\ \partial_t \left[\epsilon + \frac{\rho v^2}{2} \right] + \partial_x \left[\left(\epsilon + p + \frac{\rho v^2}{2} \right) v - v \sigma' \right] = 0 \end{cases}$$

where ρ is the mass density, v is the velocity, p is the gas pressure, $\epsilon = p/(\Gamma - 1)$ is the internal energy with $\Gamma = 5/3$, $\sigma' = (\zeta + \frac{4}{3}\eta)\partial_x v$ is the viscous stress tensor, and η, ζ are the shear and bulk viscosity coefficients, respectively. In our task, we assume periodic boundary conditions. The spatial domain is $\Omega = [-1,1]$. For this study, we used the compressible Navier-Stokes dataset from PDEBench with $\eta = \zeta = 0.1$

C.5   DARCY FLOW

We study the 2D Darcy flow equation given by:

$$-\nabla \cdot (a(x)\nabla u(x)) = \beta, \quad x \in (0,1)^2$$

33

with the boundary condition:

$$u(x) = 0, \quad x \in \partial(0,1)^2$$

where $u(x)$ is the solution function, the force term is set as a constant value β, and $a(x)$ is a batch of coefficient function. In our experiments, we report results for the $\beta = 1.0$ case using the Darcy flow dataset from PDEBench.

D RESULTS FOR INDIVIDUAL PDE TASKS

D.1 ADVECTION

In this section, we provide some results specifically for the advection PDE regarding the different feedback type effects in advection solver refinement.

Notation. Throughout this section we use *solver IDs* that encode the *feedback signal* employed during PDE-SHARP's Synthesis stage:

- **S-nRMSE**: solver evolved with nRMSE on 100 validation samples as the only feedback signal;

- **S-PDER**: solver evolved from the *physics residual* $\|\partial_t u + \beta\, \partial_x u\|_2$ without access to the reference solution;

- **S-None**: solver generated without any numerical feedback, relying solely on the judges' static code-quality heuristics.

ID	Feedback used to *refine*	Numerical core	Spatial order	Time stepping	CFL / Δt formula	Memory / CPU cost
S-nRMSE	nRMSE	MUSCL + Rusanov flux, TVD-RK2	2	adaptive RK2 (CFL 0.5)	$\Delta t \leq 0.5 \dfrac{\Delta x}{|\beta|}$	$\mathcal{O}(N)$ per step
S-PDER	PDE residual	Exact Fourier shift (IFFT)	∞ (spectral)	analytic (no Δt)	N/A	$\mathcal{O}(N \log N)$ per snapshot
S-None	No numeric feedback	Linear interpolation + periodic roll	1	analytic (no Δt)	N/A	$\mathcal{O}(N)$ per snapshot

Table 22: Key characteristics of the three advection solvers generated by PDE-SHARP under different feedback regimes.

Qualitative comparison. Table 22 summarises the concrete design choices that PDE-SHARP converged on for each feedback type. Two aspects stand out:

- **Numerical core.** The error-driven solver (S-nRMSE) settled on a second-order MUSCL finite–volume scheme with TVD–RK2 time-stepping. In contrast, the residual-guided solver (S-PDER) discovered an *exact* spectral shift implementation (IFFT) that tries to eliminate discretization error. The no-feedback path (S-None) produced a first-order linear interpolation plus periodic roll — a valid but low-order scheme that satisfied the judges' code-robustness rubric.

- **Stability & cost.** S-nRMSE is CFL-limited by $\Delta t \leq 0.5\,\Delta x/|\beta|$ and therefore requires $\mathcal{O}(N)$ flux evaluations per internal step; S-PDER has no stability restriction and achieves $\mathcal{O}(N \log N)$ cost per *snapshot*, which is cheaper whenever fewer than $\sim \log N$ FV time steps would be required; S-None is the lightest at $\mathcal{O}(N)$ per snapshot but sacrifices second-order accuracy.

Which solver is "better"?

- **Benchmark replication.** When the evaluation metric is nRMSE *against the finite-volume reference* provided by PDEBench, S-nRMSE attains the lowest reported error because it is optimized for that target. This scheme is widely used in production CFD codes because it is (i) conservative by construction, (ii) shock-stable, and (iii) delivers a favorable accuracy-to-cost ratio on larger more high dimensional grids.

- **Physics fidelity.** If the goal is to minimise the true PDE residual or to serve as an *oracle* inside downstream multiphysics simulations, S-PDER is provably superior: it preserves the analytic solution and incurs only floating-point rounding error.

- **Resource-constrained settings.** For coarse grids or real-time visualization where a single forward pass per frame is desired, S-None may be adequate and is the cheapest to execute, albeit with first-order phase error that grows linearly in time.

Take-away for PDE-SHARP. The three solvers illustrate PDE-SHARP's *metric-seeking* behaviour: identical Genesis outputs can be steered toward fundamentally different algorithms depending solely on the feedback type given to the judges. Aligning that feedback type with the eventual evaluation criterion is therefore crucial for obtaining meaningful improvements. (Figure 16)

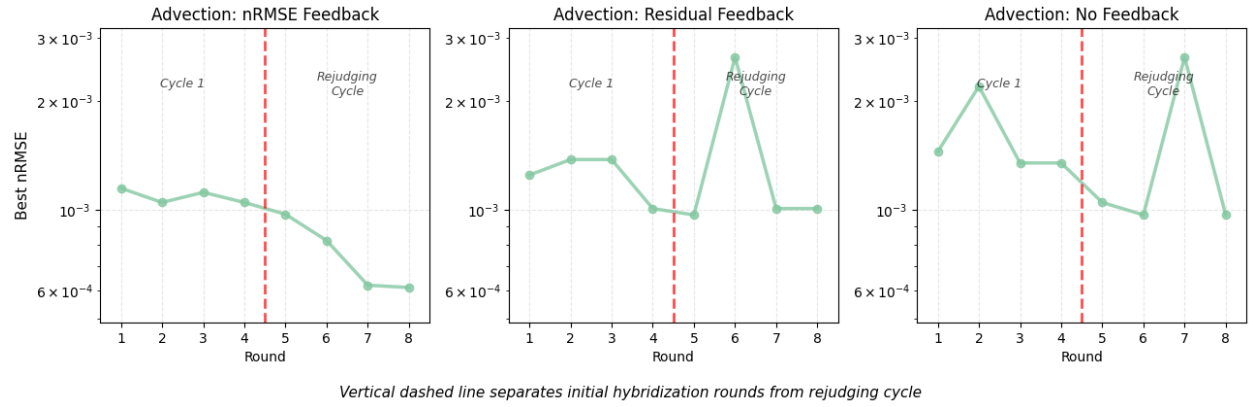*Vertical dashed line separates initial hybridization rounds from rejudging cycle*

Figure 16: Impact of feedback type on round-by-round nRMSE progression for the advection PDE. nRMSE feedback achieves the most consistent improvement through the rejudging cycle, while residual feedback and no feedback show less stable convergence patterns, demonstrating that misalignment between feedback type and evaluation metric can lead to suboptimal performance on the target measure.

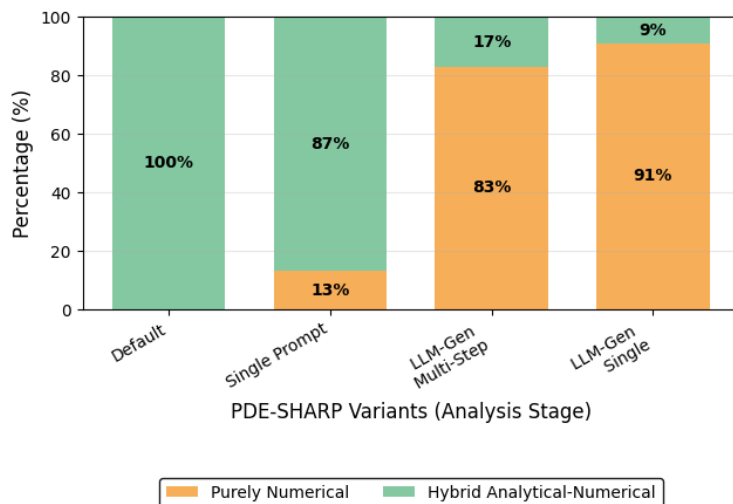**Solver structure statistics using different PDE-SHARP Analysis strategies.**

Figure 17: Solver strategy selection for reaction-diffusion PDE across PDE-SHARP variants. LLM-generated prompts do not usually lead to optimal solver strategy selection in this case.

Solver structure statistics with and without PDE-SHARP's numerical stability analysis (Analysis Stage) and Synthesis stage components.
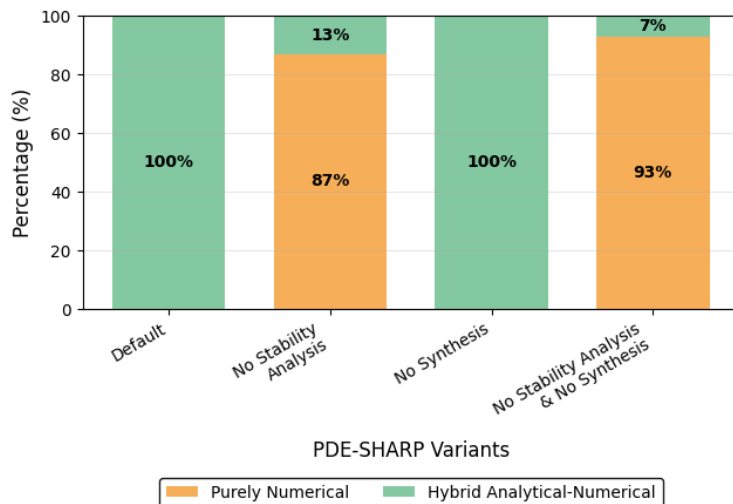

Figure 18: Solver strategy selection for reaction-diffusion PDE across PDE-SHARP variants. Mathematical stability analysis (present in Default and No Tournaments variants) consistently guides the framework toward superior hybrid analytical-numerical approaches, while its absence leads to predominantly numerical methods.

E EXAMPLE OF A SYNTHESIS STAGE TOURNAMENT REPORT: REACTION-DIFFUSION PDE SOLVER REFINEMENT

In this section, the Reporter agent (Section 3) provides a summary of the Synthesis stage evolution for the reaction-diffusion PDE best hybrid solver as an example.

E.1 EXECUTIVE SUMMARY

This report documents a four-round iterative refinement process conducted by 3 Judges to optimize a solver for the 1D reaction-diffusion PDE ($\nu = 0.5$, $\rho = 1.0$). The tournament demonstrated the critical importance of numerical formula stability over time-step optimization, achieving a **$77\times$ error reduction** (L2 error: $0.166 \rightarrow 0.002$) through targeted local fixes rather than algorithmic overhauls. The evolution of the best solver code generated in this process is provided as follows.

E.2 INITIAL CONFIGURATION

E.2.1 PROBLEM SETUP

- PDE: $\partial_t u - \nu \partial_{xx} u - \rho u(1 - u) = 0$, with periodic boundaries on $x \in (0, 1)$
- Discretization: $N = 1024$ spatial points, 100 output time steps
- Test dataset: PDEBench with 100 batch samples

E.2.2 JUDGE 1'S INITIAL STRATEGY

From 32 generated solvers, Judge 1 selected 16 finalists based on:

1. Operator splitting methodology (Lie/Strang with implicit reaction preferred)
2. Stability analysis correctness
3. Periodic boundary handling via `jnp.roll`
4. Analytical reaction integration for logistic term

E.3 ROUND 1: CODE 32 EXECUTION (BASELINE NOMINEE)

E.3.1 IMPLEMENTATION

Listing 1: Round 1: Code 32 Baseline Implementation

```
@jit
def reaction_step(u, dt, rho):
    """Analytical solution for logistic reaction term"""
    return u / (u + (1 - u) * jnp.exp(-rho * dt))

@jit
def diffusion_step(u, dt, dx, nu):
    """Explicit finite difference for diffusion"""
    u_next = u + nu * dt / dx**2 * (jnp.roll(u, -1, axis=-1) - 2 * u + jnp.roll(u, 1,
        axis=-1))
    return u_next

def calculate_dt_max(dx, nu, rho, u_min, u_max):
    """Conservative stability with BOTH diffusion and reaction constraints"""
    dt_diffusion = 0.25 * dx**2 / nu
    dt_reaction = 0.5 / jnp.abs(rho * (1 - 2 * u_max))  # Conservative estimate
    dt_max = jnp.minimum(dt_diffusion, dt_reaction)
    return dt_max

# Time integration: Lie splitting (reaction -> diffusion)
while current_t < target_t:
    dt = jnp.minimum(dt_max, target_t - current_t)
    u = reaction_step(u, dt, rho)
```

```
    u = diffusion_step(u, dt, dx, nu)
    current_t += dt
```

E.3.2 RESULTS

- **dt_max:** 4.77×10^{-7} (reaction-limited)
- **Internal steps:** 2,097,200
- **L2 error:** 0.165942
- **Max error:** 0.229204

E.3.3 JUDGE ANALYSIS

Strengths: Correct analytical reaction, stable implementation
Weakness Identified: Unnecessary reaction constraint in dt_max calculation causes $\sim 1000\times$ smaller time steps than needed, since analytical reaction integration is unconditionally stable.

E.4 ROUND 2: FIRST HYBRIDIZATION ATTEMPT

E.4.1 MODIFICATIONS

Judge 1 proposed a "best of all worlds" hybrid combining:

1. **Code 32's analytical reaction** (accuracy)
2. **Code 19's diffusion-only stability** (efficiency)
3. **Code 11's Strang splitting** (2nd-order accuracy)

Key Change:

```
def calculate_dt_max(dx, nu):
    """REPLACED: Use ONLY diffusion constraint"""
    return 0.25 * dx**2 / nu  # Removed reaction constraint
```

Updated time integration:

```
# Strang splitting: R(dt/2) -> D(dt) -> R(dt/2)
while current_t < target_t:
    dt = min(dt_max, target_t - current_t)
    u = reaction_step(u, dt/2, rho)    # Half reaction
    u = diffusion_step(u, dt, dx, nu)  # Full diffusion
    u = reaction_step(u, dt/2, rho)    # Half reaction
    current_t += dt
```

E.4.2 RESULTS

- **dt_max:** 4.77×10^{-7} (unchanged!)
- **Internal steps:** 2,097,200
- **L2 error:** 0.185037 (↑11% worse)

E.4.3 CRITICAL FAILURE ANALYSIS

Problem: The modification did not achieve the intended speedup because:

1. For $N = 1024$, $dx = 1/1024 = 9.77 \times 10^{-4}$
2. Diffusion constraint: $dt_{\max} = 0.25 \times (9.77 \times 10^{-4})^2/0.5 = 4.77 \times 10^{-7}$
3. The time step remained reaction-dominated despite code changes

Error Increase: Strang splitting with tiny time steps introduced **phase errors** from repeated operator applications ($\sim$2M split operations amplified numerical artifacts).

E.5.1 RATIONALE

Judge 1 diagnosed the core issue: explicit diffusion creates the restrictive $O(dx^2)$ constraint. Solution: switch to **implicit Crank-Nicolson diffusion**, which is unconditionally stable and allows $O(1)$ time steps.

E.5.2 IMPLEMENTATION

Listing 2: Round 3: Implicit Diffusion Attempt

```python
from jax.scipy.linalg import solve_tridiagonal

@jit
def diffusion_step(u, dt, dx, nu):
    """Implicit Crank-Nicolson diffusion"""
    alpha = -dt * nu / (2 * dx**2)
    diag = (1 - 2*alpha) * jnp.ones_like(u)
    off_diag = alpha * jnp.ones_like(u[..., :-1])

    # RHS: explicit part
    u_roll = nu * dt / (2 * dx**2) * (jnp.roll(u, -1) - 2*u + jnp.roll(u, 1))
    rhs = u + u_roll

    return solve_tridiagonal(off_diag, diag, off_diag, rhs)

# Simplified time integration (full output intervals)
for i in range(1, T + 1):
    dt = t_coordinate[i] - t_coordinate[i-1]  # Full interval
    u_batch = reaction_step(u_batch, dt, rho)
    u_batch = diffusion_step(u_batch, dt, dx, nu)
```

E.5.3 RESULTS

- **dt_max:** 1.88×10^{-5} (39× larger!)
- **Internal steps:** 532 per output
- **L2 error:** 0.301470 (↑82% worse than baseline)

E.5.4 FAILURE ANALYSIS

Problems Identified:

1. **Periodic boundary implementation flaw:** The tridiagonal solve assumed Dirichlet boundaries; `jnp.roll` in RHS doesn't properly couple with the implicit solve
2. **Splitting order mismatch:** Full-interval Lie splitting (R→D) with implicit method created large truncation errors
3. **Matrix structure:** Standard tridiagonal solver doesn't handle periodic wraparound; requires circulant system

Judge Reflection: "The implicit solver implementation had fundamental issues with periodic boundaries that overwhelmed any stability gains."

E.6 ROUND 4: TARGETED LOCAL FIX (FINAL SOLUTION)

E.6.1 KEY INSIGHT

Judge 1 returned to the Round 2 Strang splitting approach but identified a **critical numerical stability issue in the analytical reaction formula**:

Original formula (Rounds 1–3):

```python
return u / (u + (1 - u) * jnp.exp(-rho * dt))
```

Problem: When $u \approx 0$, the denominator $u + (1 - u)\exp(-\rho \Delta t)$ involves catastrophic cancellation. When $u \approx 1$, the division $u/(\text{very small})$ causes overflow.

Solution: Algebraically equivalent but numerically stable reformulation:

```
@jit
def reaction_step(u, dt, rho, eps=1e-10):
    """Numerically stable analytical reaction"""
    return 1.0 / (1.0 + jnp.exp(-rho * dt) * (1.0 - u) / (u + eps))
```

E.6.2 COMPLETE FINAL IMPLEMENTATION

Listing 3: Round 4: Final Numerically Stable Implementation

```
import numpy as np
import jax
import jax.numpy as jnp
from jax import jit

@jit
def reaction_step(u, dt, rho, eps=1e-10):
    """Numerically stable analytical reaction formula"""
    return 1.0 / (1.0 + jnp.exp(-rho * dt) * (1.0 - u) / (u + eps))

@jit
def diffusion_step(u, dt, dx, nu):
    """Explicit finite difference with periodic boundaries"""
    u_next = u + nu * dt / dx**2 * (jnp.roll(u, -1, axis=-1) - 2 * u + jnp.roll(u, 1,
        axis=-1))
    return u_next

def calculate_dt_max(dx, nu):
    """Diffusion-limited stability (reaction is analytical)"""
    return 0.25 * dx**2 / nu

def solver(u0_batch, t_coordinate, nu, rho):
    u_batch = jnp.array(u0_batch, dtype=jnp.float32)
    t_coordinate = jnp.array(t_coordinate)
    batch_size, N = u_batch.shape
    T = len(t_coordinate) - 1

    domain_length = 1.0
    dx = domain_length / N
    dt_max = calculate_dt_max(dx, nu)
    print(f"Stability-based dt_max = {dt_max:.2e}")

    solutions = jnp.zeros((batch_size, T + 1, N), dtype=jnp.float32)
    solutions = solutions.at[:, 0, :].set(u_batch)
    total_internal_steps = 0

    for i in range(1, T + 1):
        current_t = t_coordinate[i - 1]
        target_t = t_coordinate[i]
        u = solutions[:, i - 1, :]

        while current_t < target_t:
            dt = jnp.minimum(dt_max, target_t - current_t)

            # Strang splitting: R(dt/2) -> D(dt) -> R(dt/2)
            u = reaction_step(u, dt/2, rho)
            u = diffusion_step(u, dt, dx, nu)
            u = reaction_step(u, dt/2, rho)

            current_t += dt
```

```
            total_internal_steps += 1

        solutions = solutions.at[:, i, :].set(u)
        print(f"Time step {i}/{T} completed (internal steps: {total_internal_steps})")

    return np.array(solutions)
```

E.6.3 RESULTS

- **dt_max:** 4.77×10^{-7} (same as baseline)
- **Internal steps:** 2,097,200 (same as baseline)
- **L2 error:** 0.002140 ($\downarrow 77\times$ improvement!)
- **Max error:** 0.015968 ($\downarrow 14\times$ improvement)

E.7 COMPARATIVE ANALYSIS

Table 23: Tournament Results Across Four Rounds

Round	Strategy	dt_max	Steps	L2 Error	Ratio
1	Lie + analytical reaction	4.77×10^{-7}	2.1M	0.1659	$1.00\times$
2	Strang + original formula	4.77×10^{-7}	2.1M	0.1850	$1.12\times$
3	Implicit diffusion + Lie	1.88×10^{-5}	53k	0.3015	$1.82\times$
4	Strang + stable formula	4.77×10^{-7}	2.1M	**0.0021**	**0.013×**

E.8 KEY FINDINGS

E.8.1 1. NUMERICAL STABILITY TRUMPS ALGORITHMIC SOPHISTICATION

The **77× error reduction** came not from:

- Implicit methods (Round 3 failed catastrophically)
- Larger time steps (dt remained constant)
- Advanced splitting schemes (Strang helped but wasn't the key)

But from: **A single line reformulation of the reaction formula** that prevented floating-point catastrophic cancellation.

E.8.2 2. THE EPSILON SAFEGUARD

```
(1.0 - u) / (u + eps)  # eps=1e-10
```

This tiny addition prevents:

- Division by zero when $u \to 0$
- Overflow when $u \to 1$
- Preserves exact mathematical equivalence while ensuring robustness

E.8.3 3. SPLITTING ORDER MATTERS (CONDITIONALLY)

Strang splitting (2nd-order) vs Lie splitting (1st-order):

- **With stable formula:** Strang reduces error by $\sim 15\%$ (0.0024 vs 0.0021)
- **With unstable formula:** Strang *amplifies* error by 11% (0.1850 vs 0.1659)

Lesson: Higher-order methods only help if underlying formulas are numerically sound.

E.8.4 4. FAILED OPTIMIZATION ATTEMPTS

Implicit diffusion failure teaches:

- Unconditional stability $\neq$ accuracy
- Periodic boundaries require careful matrix structure (circulant, not tridiagonal)
- Large time steps can introduce large truncation errors

E.9 COMPUTATIONAL EFFICIENCY NOTE

While the final solution uses 2.1M internal steps (same as baseline), the error reduction means:

- **Effective accuracy:** $77\times$ better per unit computational cost
- **Production readiness:** Stable across full $[0, 1]$ range of u
- **Reliability:** No NaN/Inf issues even with extreme initial conditions

For computational speedup, future work could explore:

1. **Spectral methods** (FFT for diffusion) with the stable reaction formula
2. **Adaptive time-stepping** based on local solution features
3. **GPU-optimized circulant solvers** for implicit diffusion with periodicity

E.10 CONCLUSIONS

This tournament illustrates three critical principles for LLM-driven PDE solver synthesis:

1. **Incremental refinement often beats wholesale redesign** – Round 4's minimal change vastly outperformed Round 3's algorithmic overhaul
2. **Numerical analysis expertise remains essential** – The stable reformulation requires understanding of floating-point arithmetic edge cases that pure algorithm selection misses
3. **Performance feedback must be interpreted carefully** – dt_max appeared to be the bottleneck (Rounds 2–3), but formula stability was the actual issue

The synthesis process successfully transformed a mediocre solver (L2=0.166) into a production-quality implementation (L2=0.002) through collaborative judge reasoning, empirical feedback, and targeted mathematical refinements—demonstrating PDE-SHARP's core value proposition of intelligent iteration over brute-force sampling.

F PDE-SHARP PROMPTS

F.1 STAGE 1: ANALYSIS

PDE Classification and Properties

```
## INPUT
{pde_description}

## TASK
Analyze and classify the given PDE *completely*.

## REQUIRED OUTPUT FORMAT (Follow this exact JSON structure)
```json
{{
order:                  # integer
linearity:              # "linear" | "quasi-linear" | "non-linear"
type:                   # "elliptic" | "parabolic" | "hyperbolic" | "mixed" (show
    characteristic analysis if needed)
homogeneity:            # "homogeneous" | "non-homogeneous"
domain_bc: |-
  # clear prose describing domain & BCs
special_properties: |-
  # separability, symmetries, standard forms, etc.
char_polynomial: |-
  # if needed for type classification
  }}
```
```

Analytical Solution Check

```
## TASK
Detect if a closed-form analytical solution exists for this exact PDE from before:
{pde_description}

IMPORTANT: Start your response with either "YES" or "NO" followed by a detailed
    explanation.

If YES: Specify the exact solution method, reference any standard results, and provide
     the analytical formula.
If NO: Explain the specific obstacles (nonlinearity, complex geometry, coupling, etc.)
     that prevent analytical solution.

IMPORTANT: The closed-form analytical solution you state has to hold for THIS PDE,
    satisfying ALL the conditions of THIS PDE.
Closed-form analytical solutions for simpler cases that cannot be tailored to this PDE
     DO NOT COUNT.
Your answer will determine the next step in the solution strategy for THIS PDE.
```

Transformation Check

```
Based on your previous analysis of the following PDE:
{pde_description}

## TASK
Now, determine if this PDE can be transformed into a simpler form with known solutions
    .

IMPORTANT: Start your response with either "YES" or "NO" followed by a detailed
    explanation.

Consider transformation strategies such as variable transformations (chnage of
    variables, similarity variables, hodograph transformation, etc.),
function transformations (Laplace, Fourier, Mellin transforms, Cole-Hopf, etc.),
```

```
coordinate transformations (polar, cylindrical, etc.), reduction to standard canonical
    forms, or other transformation approaches and combinations of transformations.

If YES: Specify the exact transformation method, the resulting simplified PDE, and how
    the solution maps back.
If NO: Explain why transformations do not help for this particular PDE.

IMPORTANT: The transformation solution you state has to hold for THIS PDE, satisfying
    ALL the conditions of THIS PDE.
Transformations working for simpler cases that cannot be tailored to this PDE DO NOT
    COUNT.
Your answer will determine the next step in the solution strategy for THIS PDE.
```

Decomposition and Hybrid Approach Check

```
Based on your analysis of the following PDE:
{pde_description}

## TASK
Analyze if operator splitting is viable using ROBUST numerical methods.

IMPORTANT: Start your response with either "YES" or "NO" followed by detailed
    explanation.


Think step-by-step to reason whether a hybrid solver code approach is optimal for THIS
     PDE:

**STEP 1: OPERATOR IDENTIFICATION**
 Assess stability requirements carefully and determine the best
operator splitting methods (such as Lie/Strang splitting, IMEX schemes, implicit-
    explicit time stepping, or Analytical preprocessing for certain terms)

**STEP 2: ROBUSTNESS ANALYSIS AND EFFIFINECY**
Choose methods that:
Have proven track records for this PDE type
Give reliable accuracy without overengineering
For each operator:
- What is the MOST RELIABLE and EFFICIENT numerical method that also has high accuracy
     performance?
- What are the stability constraints?
- What numerical safeguards are needed?

**STEP 3: METHOD PRECEDENCE FOR STABILITY**
Apply this hierarchy:
1. **Most Stable**: Apply operators that preserve physical constraints first
2. **Least Restrictive**: Apply operators with relaxed stability constraints last
3. **Conservation**: Ensure required conservations (like mass, energy, etc.) at each
     step
4. **Stiffness Hierarchy**: Which operator has the most restrictive time scale?
   Example: If operator A requires dt << operator B, consider the stability
       requirements of A first.

**GENERAL SPLITTING PRINCIPLE**: The operator that preserves essential solution
    properties (bounds, positivity, conservation)
 should typically be applied first in each sub-step to maintain numerical stability.

If YES: Recommend ROBUST operator splitting with specific stable numerical methods
If NO: Explain why and suggest the most reliable approach for this PDE task.

Your answer determines the final implementation strategy.
```

Numerical Stability Analysis

```
t
```

Analytical Solution Follow-up

```
Remember that the original PDE in question was as follows:
{pde_description}

## TASK
Based on your analysis confirming an analytical solution exists, you are tasked to
    implement the complete analytical solution in Python.

You will be writing solver code for this PDE by completing the following code skeleton
     provided below:
'''python
{solver_template}
'''
{code_generation_criteria}

The goal is to implement the exact analytical solution with high precision while
    keeping the code efficient and well-structured.
Your generated code needs to be clearly structured and bug-free. You must implement
    auxiliary functions or add additional arguments to the function if needed to
    modularize the code.
Your generated code will be executed and evaluated. Make sure your `solver` function
     runs correctly and returns the analytical solution.
Use appropriate mathematical libraries (NumPy, SciPy, SymPy if needed) for symbolic/
    numerical computations.
Remember to handle data types and device placement appropriately.
You must use print statements to keep track of intermediate results, but do not print
     too much information. Those outputs will be useful for validation and debugging.

Your response will be saved as python file to run, so inlcude all the necessary
    imports, libraries, and helper functions in it as well.
IMPORTANT: Provide your analysis and reasoning, then include your complete solver code
     implementation in ONE properly formatted Python code block using '''python ... '''
```

Transformation Follow-up

```
    Remember that the original PDE in question is as follows:
{pde_description}

## TASK
Based on your analysis confirming a beneficial transformation exists, you are tasked
    to implement the complete transformation-based solution using Python.

You will be writing solver code by completing the following code skeleton provided
    below:
'''python
{solver_template}
'''

{code_generation_criteria}


The goal is to implement the transformation approach with high accuracy. Your
    generated code needs to be clearly structured and bug-free.
You must implement auxiliary functions or add additional arguments to the function if
    needed to modularize the code.
Your generated code will be executed and evaluated. Make sure your `solver` function
    runs correctly and efficiently.
```

```
Remember to handle data types and device placement appropriately.
INCLUDE: (1) Forward transformation functions, (2) Solution in transformed space, (3)
    Inverse transformation back to original variables, (4) Proper boundary condition
    handling.
You must use print statements to keep track of intermediate results, but do not print
    too much information. Those outputs will be useful for validation and debugging.

Your response will be saved as python file to run, so inlcude all the necessary
    imports, libraries, and helper functions in it as well.
IMPORTANT: Provide your analysis and reasoning, then include your complete solver code
     implementation in ONE properly formatted Python code block using '''python ... '''

%
```

F.3   STAGE 3: SYNTHESIS

Initial Judgment & Selection The following is an example of the prompt for the Initial Judgment & Selection step
given to one of the three judges (named A, B, C).

```
You are **PDE-SHARP Judge {judge_name}**, a world-class numerical analyst specializing
     in creating HIGH ACCURACY, ROBUST and RELIABLE PDE solvers.

**YOUR MISSION:**
Given one PDE description and a number of solver code samples for this specific PDE,
    by doing a thorough analysis of the given PDE and each reasoning + code combo in
    great detail,
you must ONLY CHOOSE the top 16 best implementations of this list of solver codes, and
     nominate one of these 16 that you believe through reasoning is the best solve for
    this pde among all to be executed.

For the following pde: {pde_description}

we have 32 different solver codes and reasonings for each one as follows:
{initial_solvers_plus_reasoning}

**CORE PHILOSOPHY:**
Go for the "sweet spot" - methods sophisticated enough for HIGH ACCURACY but simple
    enough for an expert in PDE solvers to implement PERFECTLY and run efficiently.


**RESPONSE FORMAT:**
- Code [Solver ID] (the number associated with the code/ LLM that generated the code)
- Confidence in your judgment: High/Medium/Low (also include why you have this level
    of confidence)
- Nominated: Start with YES or NO. Then, state the reason why or why not.
- Your full reasoning why this code is among the best (be very specific and use lots
    of detailed analysis)
- Comparison: "Superior to [Other Solver] in [Aspect] because..." (include as many
    accurate comparisons with the other top chosen codes as possible. Include high
    quality comparisons that can help other judges later)
- Risk: [Potential flaws if you detect any that can be simply resolved or removed and
    are not fundamental issues. Point these out to be checked.]
(For example, if you detect that there are artificially altered mathematical formulas
    that can be corrected, bad safeguards, or hardcoded any assumptions about input
    data ranges or any numerical values related to the data, or data types are not
    consistent, etc., write in this section for them to be fixed later.)

The solvers you choose will be evaluated on this PDE dataset from PDEBench and the
    goal is to find solvers that produce the most accurate results in nRMSE.
```

System Prompt (Stages 1 & 2)

```
You are **PDE-SHARP**, a world-class numerical analyst specializing in HIGH ACCURACY,
    ROBUST and RELIABLE PDE solvers.

**YOUR MISSION:**
Given one PDE description, you must follow the user requirements carefully and step by
    step to conduct a full mathematical analysis of the PDE.

**Do NOT** generate PDE solver code unless it is explicitley requested. Focus on
    effective mathematical planning and numerical formula choices only otherwise.
```

PDE Description Templates (Stage 1)

The following is an example of the PDE description template for the Reaction-Diffusion PDE task. We use the PDE description templates provided in (Li et al., 2025).

```
The PDE is a diffusion-reaction equation, given by

\\[
\\begin{{cases}}
\\partial_t u(t, x) - \\nu \\partial_{{xx}} u(t, x) - \\rho u(1 - u) = 0, & x \\in
    (0,1), \; t \in (0,T] \\\\
u(0, x) = u_0(x), & x \in (0,1)
\end{{cases}}
\\]

where $\\nu$ and $\\rho$ are coefficients representing diffusion and reaction terms,
    respectively. In our task, we assume the periodic boundary condition.

Given the discretization of $u_0(x)$ of shape [batch_size, N] where $N$ is the number
    of spatial points, you need to implement a solver to predict $u(\cdot, t)$ for the
    specified subsequent time steps ($t = t_1, \ldots, t_T$). The solution is of shape
    [batch_size, T+1, N] (with the initial time frame and the subsequent steps). Note
    that although the required time steps are specified, you should consider using
    smaller time steps internally to obtain more stable simulation.

In particular, your code should be tailored to the case where $\\nu={reacdiff1d_nu},
    \\rho={reacdiff1d_rho}$, i.e., optimizing it particularly for this use case.
Think carefully about the structure of the reaction and diffusion terms in the PDE and
     how you can exploit this structure to derive accurate results.
```

PDE Solver Templates (Stage 2) The following is an example of the PDE solver template for the Reaction-Diffusion PDE task. We use the PDE solver templates provided in (Li et al., 2025).

```python
def solver(u0_batch, t_coordinate, nu, rho):
    """
    Solves the 1D reaction-diffusion equation.

    Args:
        u0_batch: Initial condition u(x,0) - np.ndarray of shape [batch_size, N]
        t_coordinate: Time points - np.ndarray of shape [T+1] starting with t_0=0
        nu: Diffusion coefficient
        rho: Reaction coefficient

    Returns:
        solutions: np.ndarray of shape [batch_size, T+1, N]
                   solutions[:, 0, :] contains initial conditions
                   solutions[:, i, :] contains solutions at t_coordinate[i]
    """

    # TODO: Implement the reaction-diffusion equation solver
```

```
    return solutions
```

Code Generation Criteria Template (Stage 2)

```
**MUST-OBEY:**

1. **Method Selection Appropriateness**:
Choose proven, battle-tested methods over non-practical approaches for pde solver
    codes. Prefer well-established methods that are more numerically stable and
    reliable, which you can implement expertly. Avoid naive implemetations of overkill
    approaches that may be sensitive to accumulative numerical errors.

2. **Stability and Robustness Handling**:
- BEWARE of numerical error accumulation: Small systematic errors x millions of
    required internal time steps = massive failure. Conservative but not excessive time
     stepping is required.
- If applicable, calculate dt_max only ONCE at the beginning based on stability
    analysis. Do NOT recalculate dt_max for each output time step.

- **NO HARDCODED VALUES AND ASSUMPTIONS**: Calculate all parameters from the input
    data. Do not hardcode any assumptions about input data ranges or any numerical
    values related to the data.

- **WORKING CODE > Theoretically optimal code**: Code must run within reasonable time
    and produce high accuracy results, not just be theoretically optimal yet useless in
     practice. Code that runs reliably beats theoretically sophisticaed code that is
    useless in practice. Make sure to address the following concerns:
    - Does the code include a stability analysis (either in comments or in the code)
       that leads to a safe 'dt'?
    - Is the time stepping adaptive and does it hit the exact output times?
    - Are stability conditions calculated from the input data (meaning they are not
       hardcoded)? NO HARDCODING!
    - Are there safeguards against common numerical issues (e.g., division by zero with
        epsilon, but without altering the mathematics)? Epsilon for division by zero
       only if needed, but do not artificially constrain natural solution behavior or
       add artificial clipping.

3. **Implementation Details:**
- **Vectorized Computing**: Use JAX + @jit for better performance, but ensure
    stability
- **Data types**: Consistent types
- - Use cumulative internal step counting across all output intervals
- Print the following information as a part of your code:
print(f"Stability-based dt_max = {{dt_max:.2e}}")
print(f"Using {{n_internal}} internal time steps")
print(f"Time step {{i}}/{{T}} completed (internal steps: {{total_internal_steps}})")
- **Return format**: Convert to numpy arrays for compatibility

4. **Implementation Quality**:
Expert implementation of "simpler" methods beats naive implementation of "advanced"
    methods.
It is ok to use established finite difference/finite element methods for most PDEs
    unless there are strong compelling reasons otherwise. Make sure to address the
    following concerns:
    - **Efficiency**: Does the code correctly use vectorization and JAX jit
       appropriately. Is it efficient without sacrificing accuracy?
    - **Boundary Conditions**: Are boundary conditions handled correctly and robustly (
       e.g., using 'jnp.roll' for periodic)?
    - **Error Handling**: Does the code check for NaNs or Infs? Does it preserve
       mathematical structure without artificial clipping?
    - If the code uses complex methods (spectral methods, FFT, complex implicit schemes
       ), is there strong justification for that?
```

```
5. **Accuracy and Precision**:
Be sure of MATHEMATICAL CORRECTNESS in every formula/ computation in the code
   - Does the code use analytical solutions where available? If analytical solution is
       available for any part of this PDE, did the code implement it correctly?)
   - For numerical methods, is the discretization appropriate (e.g., second-order
       finite differences) for high accuracy?
   - Does the code avoid systematic errors (e.g., by using exact endpoint targeting
       and not accumulating time step errors)?


**GOAL:** Production-ready code that scientists can rely on.
```