WEB-PDE-LLM: From Mathematical PDE Models to Interactive Real-Time GPU Simulations in a Web Browser with LLM Agents

Anonymous Author(s)
Affiliation
Address
email

Abstract

Simulation of large systems of partial differential equations (PDEs) underlies the modeling of many physical and biological systems, yet translating complex mathematical models into accurate, efficient, and usable simulations remains technically demanding. We present WEB-PDE-LLM, a multi-agent framework that transforms mathematical PDE descriptions into validated, interactive, real-time GPU simulations that execute directly in a web browser. Starting from a LaTeX-like PDE specification, specialized LLM agents parse the model and numerical parameters, generate WebGL GPU solver code, execute and verify the simulation in a browser, and iteratively repair the implementation using shader-compilation errors and numerical-error feedback. After repair, the shader is frozen and evaluated against held-out reference trajectories never exposed to the agents. Unlike existing LLM-based scientific simulation frameworks that primarily generate Python or domain-specific solver code, WEB-PDE-LLM directly generates GPU computational kernels for browser execution, producing self-contained simulations requiring no Python environment, compilation, or external numerical libraries. We evaluate the framework across five PDE families of increasing complexity, from one-dimensional advection and Burgers' equations to nonlinear cardiac reaction-diffusion models with up to 19 differential equations per cell, using four LLM backbones and comparisons with one-shot LLM generation and existing agent-based PDE frameworks. WEB-PDE-LLM achieves the highest overall accuracy and solver pass rate and is the only evaluated framework to generate valid solvers for the most complex cardiac model across all four LLM backbones. These results demonstrate how LLM agents can transform mathematical models directly into accessible, high-performance scientific simulations that can be executed and explored interactively on consumer devices.

1 Introduction

PDEs govern transport, diffusion, wave propagation, and many biological and physical processes. In cardiac electrophysiology, reaction-diffusion PDEs couple membrane-voltage propagation with nonlinear ionic kinetics[1], producing wave fronts, spiral waves, and fibrillatory dynamics that are costly to simulate at high resolution [2–4]. Even with mature numerical methods, the workflow is demanding: discretization, boundary conditions, stable time steps, and a compute environment. LLMs have begun to lower it by generating scientific code from natural language, part of a broader shift toward agentic workflows that combine reasoning, tool use, execution feedback, and iterative self-correction [5–8]. Recent systems such as CodePDE, LLM-PDEveloper, AutoNumerics, and MCP-SIM show that LLMs can generate, debug, and refine simulation code [9–12]. However, generating scientific

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

code is not equivalent to producing an immediately usable scientific simulation. Existing approaches typically assume a conventional execution environment such as Python or a domain-specific codebase, requiring compatible software, numerical libraries, and computational infrastructure. An important next step is therefore to determine whether LLMs can translate mathematical models directly into self-contained, high-performance simulations that can be executed and explored interactively by the user. Separately, WebGL systems have demonstrated real-time browser-based PDE and cardiac simulation on commodity devices [13–19]. These systems, however, rely on models and shader implementations supplied by developers; existing LLM-based solver-generation methods do not target the browser execution layer, where physical fields are maintained as GPU textures and numerical updates are executed via fragment shaders [20, 21].

WEB-PDE-LLM closes this gap by treating the browser itself as the simulation platform, converting a LaTeX-like PDE description into a WebGL application whose fragment shader advances the state in real time, a self-contained simulation that needs no Python or local hardware setup. Because encoding PDEs directly into WebGL textures is non-trivial, WEB-PDE-LLM adopts a multi-agent framework (Section 3) that decouples equation parsing, code generation, verification, and automated debugging [22, 23].

The main contributions of this work are:

From Mathematical Models to Executable GPU Simulations: We introduce, to our knowledge, the first automated LLM-agent framework that translates mathematical PDE specifications directly into GPU-accelerated, browser-executable scientific simulations.
Zero-Setup In-Browser Interactive Scientific Computing: We enable direct synthesis of interactive WebGL solvers from natural language, executing client-side on personal hardware, via its browser, without dependencies on Python runtimes, compilations, or external numerical libraries and specialized local software infrastructure.
Specialized Multi-Agent Generation and Validation: We develop a modular agent architecture tailored to shader synthesis that decouples mathematical equation parsing and verification, and automated debugging using compilation and numerical-error feedback.
Rigorous Benchmark from Simple to Complex PDE Dynamics: We evaluate WEB-PDE-LLM across five PDE families and four LLM backbones, with an emphasis on benchmarking numerical accuracy, solver pass rate, and token efficiency against one-shot LLM generation and existing PDE-generation frameworks (CodePDE, MCP-SIM, PINNsAgent). WEB-PDE-LLM is the only evaluated framework to generate valid solvers for the 19-variable TNNP cardiac electrophysiology model across all four LLM backbones.

2 Related Work

LLM-based PDE solver generation. CodePDE is the closest baseline in spirit, asking LLMs to generate, execute, debug, and refine PDE solvers at inference time [9]. LLM-PDEveloper instead extends PDE software libraries by translating mathematical and algorithmic descriptions into new solver modules, while AutoNumerics uses a multi-agent, coarse-to-fine pipeline to select, implement, and verify classical numerical schemes [10, 11]. WEB-PDE-LLM shares their code-as-reasoning premise but targets WebGL fragment shaders and complete browser applications, changing the failure modes and required abstractions. OpInf-LLM instead learns a data-driven surrogate offline via operator inference, emitting no inspectable solver [24]. PDEBench and the more recent Well collection benchmark learned solvers across increasingly diverse physical systems [25, 26]. Physics-informed neural networks and neural operators learn solution maps from data or physics constraints [27, 28]; recent foundation models such as Poseidon and PDEformer-1 broaden this direction through pretraining across PDE families or symbolic equation specifications [29, 30].

Browser-based PDE simulation. Browser GPU computing predates LLM-based solver generation. Abubu.js demonstrated large-scale, interactive WebGL simulations, including cardiac electrophysiology, on personal computers and mobile devices [13]. VisualPDE provides an interactive browser environment for exploring reaction–diffusion and other PDE systems [14], while CardioFit uses WebGL to accelerate interactive parametrization of cardiac action-potential models [15]. These systems establish the feasibility and value of client-side scientific simulation, but their equations and numerical kernels are hand-authored. WEB-PDE-LLM addresses the complementary problem of automatically synthesizing and repairing those kernels from mathematical descriptions.

Multi-agent systems for scientific simulation. Multi-agent LLM systems decompose complex tasks into specialized planning, coding, execution, and repair roles [8, 31–35]. MCP-SIM builds, executes, and diagnoses simulation code via a memory-coordinated agent set [12]; CFDAgent automates zero-shot CFD simulation with preprocessing, solver, and postprocessing agents [36]; related work applies similar coordination to scientific hypothesis generation [37].

3 Methodology

Problem formulation. The input to WEB-PDE-LLM is a LaTeX-like description of a time-dependent PDE system, $\partial_t \mathbf{q}(t, \mathbf{x}) = \mathcal{F}(\mathbf{q}, \nabla \mathbf{q}, \nabla^2 \mathbf{q}; \boldsymbol{\theta})$, where $\mathbf{q}$ denotes one or more state variables and $\boldsymbol{\theta}$ physical/numerical parameters, together with the spatial domain, grid resolution, time horizon, time step, and boundary conditions. The output is a WebGL simulation artifact that advances $\mathbf{q}$ on a texture grid and, when requested for evaluation, exports sampled solution values for comparison with reference data. The implementation covers one-dimensional advection and Burgers', two-dimensional heat, Fenton–Karma (FK), and Ten Tusscher-Noble-Noble-Panfilov (TNNP/TP06) [2–4] equations: the simpler PDEs test stable transport/diffusion stencils, while the cardiac models test coupled nonlinear reaction terms, piecewise kinetics, and multi-channel state layouts.

System overview. WEB-PDE-LLM is organized as a parse/code/verify/debug loop (Figure 1), following the agentic pattern of interleaving reasoning, action, external execution, and feedback-driven revision [5–7]. The framework is model-provider agnostic and routes through OpenAI, Anthropic, Gemini, DeepSeek, or cloud/local Ollama models. Parse validation matters most for smaller models, which may hallucinate fields or choose unstable time steps, motivating explicit checker and resolver roles [22, 35].

Agent details. The **Parse Agent** converts the description into a JSON object with the fields needed downstream: governing equations, state-variable count, texture size, spatial and time step, domain size, time horizon, boundary conditions, parameters, and implementation notes. A **Parsed-Content Checking Agent** then reviews it for completeness, type correctness, and numerical plausibility (especially CFL-like time-step stability), correcting unstable metadata [7, 22, 8]. Based on the state-variable count, **Skeleton Preparation** selects a WebGL texture layout (a single channel for scalar PDEs, three channels for the FK, 19 channels for the TNNP) and injects Δt, Δx, texture size, and parameters. This narrows the **Code Agent** to the mathematically critical kernel: raw GLSL implementing finite-difference Laplacians, upwind updates, and thresholded gating kinetics for the cardiac models [23]. The **Verify Agent** launches the browser simulation and monitors the WebGL shader-compilation output with the best solver (lowest train nRMSE) out of 10 trials from Code Agent. It also computes nRMSE on a training set whose initial conditions are disjoint from those used for final evaluation. If shader compilation fails or the train nRMSE exceeds a predefined acceptance threshold, the **Debug Agent** receives the corresponding compiler message or train error feedback and revises the shader. After repair, the shader code is frozen and evaluated on the held-out test set.

4 Experiments

PDEs of Interest. Our benchmark suite comprises five PDE families, chosen to stress spatial dimension, operator type (transport, diffusion, mixed), linearity, state-variable count (one to 19), and boundary conditions (periodic, no-flux/Neumann). For every case the generated solver advances the state from a prescribed initial condition to a fixed horizon T and exports sampled trajectories for comparison with reference solutions; discretization and parameter values are released with our code.
Advection. The one-dimensional linear advection equation $\partial_t u = -\beta\, \partial_x u$, $x \in (0, 1)$, $t \in (0, T]$, with periodic boundaries and a smooth initial condition, tests stable transport stencils on a 1024-point grid to $T = 2$ with $\beta = 0.1$.
Burgers'. The one-dimensional viscous Burgers' equation in conservative form, $\partial_t u + \partial_x(\frac{1}{2}u^2) = \frac{\nu}{\pi}\, \partial_{xx} u$, $x \in (0, 1)$ with periodic boundary condition, 1024-point grid and $T = 2$, $\nu = 0.001$.
Heat. The two-dimensional heat equation $\partial_t u = \alpha \nabla^2 u$, $(x, y) \in (0, 1)^2$, with periodic boundaries and diffusivity α, tests an isotropic diffusion stencil on a 128×128 grid to $T = 2$, with $\alpha = 0.01$.
FK. The FK model is a two-dimensional, three-variable reaction-diffusion system for a diffusive field $u(t, \boldsymbol{x})$ and non-diffusive gating variables v, w, coupled through three piecewise ionic currents gated by a Heaviside excitation threshold (full equations in Section A). We solve on $\Omega = [0, 10]^2$ with no-flux (Neumann) boundaries on a 512×512 grid, $T = 100$, $\Delta t = 2.5 \times 10^{-2}$, $D = 10^{-3}$,

using the standard FK parameter set [2].

TNNP. The TNNP model couples a diffusive transmembrane potential $V(t, \boldsymbol{x})$ to 19 non-diffusive state variables through a total ionic current that sums 12 nonlinear components (complete equations and standard TP06 parameters in Section A) [3, 4]. We integrate on $\Omega = [0, 12]^2$ with no-flux (Neumann) boundaries on a 512×512 grid, $T = 100$, $\Delta t = 2.5 \times 10^{-2}$, $D = 10^{-3}$.

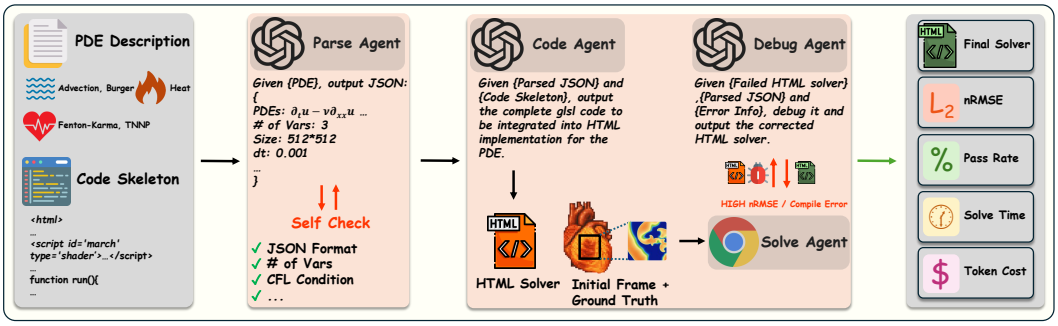

Figure 1: The WEB-PDE-LLM pipeline. A PDE description is parsed into structured metadata, checked, injected into a WebGL skeleton, completed by the Code Agent, and executed in a browser by the Verify Agent. Shader-compilation errors and train nRMSE are returned to the Debug Agent. The resulting shader is then frozen and evaluated on held-out test trajectories.

Results. We evaluate every framework on these five PDE families across four LLM backbones (Gemini 3.5 Flash, GPT-5.5, Claude Opus 4.8, and DeepSeek V4-Pro) along three metrics: accuracy (normalized Root Mean Square Error, nRMSE), pass rate, and API token cost. Throughout the result tables, the best value in each PDE column is shaded gray and the second best is underlined. The main text reports accuracy. The full pass rate, token cost and ablation experiment results are provided in Section B. Table 1 compares WEB-PDE-LLM qualitatively against state-of-the-art LLM-based PDE solving frameworks [9, 12, 38, 24] and LLM-only generation: it is the only method that is simultaneously cross-platform, zero-configuration, real-time-visualized, autonomously debugged, and training-free.

Table 1: Qualitative evaluation of generated solver capabilities.

Method	Cross-Platform Compatibility	Zero-Configuration	Real-Time Visualization	Autonomous Debugging	No Training Required
WEB-PDE-LLM	✓	✓	✓	✓	✓
CodePDE	✓	✗	✗	✓	✓
LLM-only	✓	✗	✗	✗	✓
MCP-SIM	✗	✗	✗	✓[a]	✓
PINNsAgent	✓	✗	✗	✗	✗
Op-Inf LLM	✓	✗	✗	✓	✗

[a] MCP-SIM resolves system runtime bugs but cannot correct numerical accuracy issues (e.g., high nRMSE).

Evaluation Metrics. Because several methods failed on the harder PDEs, a geometric mean over only the successful cells would reward a method for reporting results on easy problems while ignoring its failures. We therefore summarize each method with a coverage-aware Score that grades every PDE relative to the five competing methods (WEB-PDE-LLM, LLM-only, CodePDE, MCP-Sim, PINNsAgent) so a method cannot score well by solving only the easy PDEs. PDE solving accuracy uses the normalized RMSE, $\mathrm{nRMSE} = \frac{1}{S}\sum_{s=1}^{S} \|\boldsymbol{u}^{(s)} - \hat{\boldsymbol{u}}^{(s)}\|_2 / \|\boldsymbol{u}^{(s)}\|_2$, averaged over the S number of samples of that PDE. Token cost counts every input and output token each framework sends to and receives from the LLM API, priced in USD at the providers' official rates. Pass rate enters directly on its raw 0–100 scale. The exact formulas are in Section A.4.

Overall comparison. Figure 2 summarizes the three metrics under this Score. WEB-PDE-LLM leads on accuracy (87.3) and pass rate (90.5), ahead of CodePDE (51.3, 78.5) and well ahead of MCP-Sim (19.2, 47.5). The gap comes mostly from coverage on the cardiac models, where the Python baselines often fail to produce a running solver. Its weakest metric is token efficiency (45.6): the parse-code-verify-debug loop calls LLM several times, although it still beats the MCP-Sim loop, the most token-hungry method on every PDE (0.9). One-shot LLM-only generation is the cheapest in

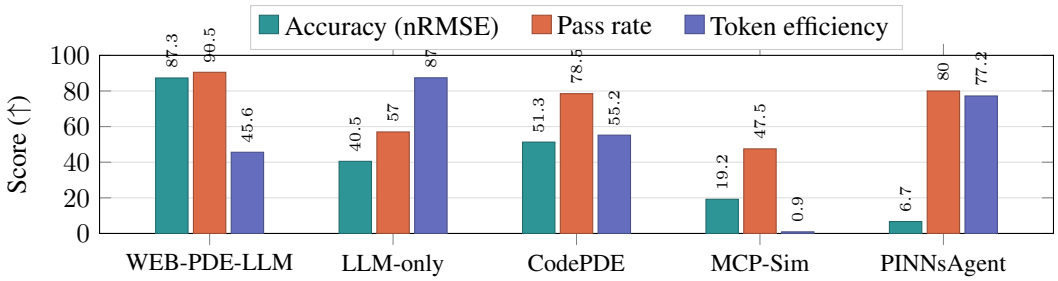

Figure 2: Coverage-aware Score per metric, averaged over four LLM backbones (Gemini 3.5 Flash, GPT-5.5, Claude Opus 4.8, DeepSeek V4-Pro). Each metric is graded per PDE relative to the competing frameworks (best $\to 100$, worst $\to 0$, fail $\to 0$) and averaged over the five PDEs; nRMSE and token cost use log-relative grading, pass rate is scored directly on its 0–100 scale (all: higher score is better).

tokens (87.4). PINNsAgent is close behind (77.2) but the least accurate (6.7), trading LLM calls for network training.

Accuracy. Table 2 reports test nRMSE per PDE and backbone. WEB-PDE-LLM is the only framework that produces a valid TNNP solver on all four backbones, down to 4.67×10^{-6} on GPT-5.5 and 3.35×10^{-6} on DeepSeek V4-Pro. Among the Python baselines, only MCP-Sim returns a TNNP success (a single DeepSeek V4-Pro run at 6.26×10^{-1}, five orders of magnitude worse), while CodePDE, LLM-only, and PINNsAgent fail on every backbone. For FK, WEB-PDE-LLM's best run reaches 2.80×10^{-4}, similar to the strongest CodePDE run (2.79×10^{-4}), while MCP-Sim never completes an FK case. On the 1D cases, the shader solvers are about an order of magnitude more accurate than the Python baselines (1.70×10^{-3} against 1.32×10^{-2} on advection, 7.24×10^{-4} against 1.67×10^{-2} on Burgers'), which is due to the parsed-metadata stage selecting a stable Δt and an upwind-consistent stencil. Heat is the exception: the one-shot Python solvers reach 2.12×10^{-7}, whereas WEB-PDE-LLM bottoms out at 1.86×10^{-6}, close to the single-precision-texture floor. PINNsAgent is competitive only on advection and otherwise trials every code-generating method by two to four orders of magnitude. Figure 3 in Section B.4 shows the predicted fields and pointwise errors behind these numbers for the Gemini 3.5 Flash backbone.

Pass Rate and Token Efficiency. Complete pass-rate and token-cost results for all PDEs, methods, and LLM backbones are reported in Section B. WEB-PDE-LLM achieves the strongest TNNP pass rate, ranging from 50–100% across backbones, and 70–90% for FK. In contrast, CodePDE never succeeds on TNNP and MCP-Sim fails on FK, with only one successful TNNP trial (10%). Although WEB-PDE-LLM uses 2–4 times more tokens than one-shot generation on simple cases, its end-to-end cost ($1.39–$13.76 per backbone) remains below CodePDE ($2.15–$22.40) and MCP-Sim ($2.49–$36.57).

Ablation Experiment. To isolate WEB-PDE-LLM's two agents, we remove them cumulatively on GPT-5.5: dropping parse-validation degrades accuracy on four of five PDEs and drops mean pass rate from 96.0% to 84.0%; further removing the Debug Agent costs another order of magnitude on advection and 4 more points (Section B.5).

5 Conclusion

We introduced WEB-PDE-LLM, a multi-agent framework that transforms mathematical PDE descriptions directly into validated, interactive GPU simulations running in a web browser. Across five PDE families and four LLM backbones, WEB-PDE-LLM achieves the highest overall accuracy and solver pass rate among the evaluated methods, with its strongest advantage on complex cardiac PDE models with many state variables. These results demonstrate that LLM agents can bridge mathematical model specification and accessible, high-performance scientific simulation.

The framework has several limitations. WebGL textures use single precision, limiting achievable accuracy, and the current implementation targets explicit time-stepping on regular grids. Globally coupled problems such as Poisson equations requiring a linear solve at each step are therefore not easily supported. Irregular geometries and three-dimensional domains also remain to be explored.

References

[1] Flavio H Fenton and Elizabeth M Cherry. Models of cardiac cell. *Scholarpedia*, 3(8):1868, 2008.

[2] Flavio H. Fenton and Alain Karma. Vortex dynamics in three-dimensional continuous myocardium with fiber rotation: Filament instability and fibrillation. *Chaos*, 8(1):20–47, 1998.

[3] K. H. W. J. ten Tusscher, D. Noble, P. J. Noble, and A. V. Panfilov. A model for human ventricular tissue. *American Journal of Physiology-Heart and Circulatory Physiology*, 286(4):H1573–H1589, 2004.

[4] K. H. W. J. ten Tusscher and A. V. Panfilov. Alternans and spiral breakup in a human ventricular tissue model. *American Journal of Physiology-Heart and Circulatory Physiology*, 291(3):H1088–H1100, 2006. doi: 10.1152/ajpheart.00109.2006.

[5] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

[6] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[7] 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. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[8] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed H. Awadallah, Adam White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[9] Shanda Li, Tanya Marwah, Junhong Shen, Weiwei Sun, Andrej Risteski, Yiming Yang, and Ameet Talwalkar. CodePDE: An inference framework for LLM-driven PDE solver generation. *arXiv preprint arXiv:2505.08783*, 2025.

[10] Haoyang Wu, Xinxin Zhang, and Lailai Zhu. Automated code development for PDE solvers using large language models. *arXiv preprint arXiv:2509.25194*, 2025.

[11] Jianda Du, Youran Sun, and Haizhao Yang. AutoNumerics: An autonomous, PDE-agnostic multi-agent pipeline for scientific computing. *arXiv preprint arXiv:2602.17607*, 2026.

[12] Donggeun Park, Hyeonbin Moon, and Seunghwa Ryu. A self-correcting multi-agent LLM framework for language-based physics simulation and explanation. *npj Artificial Intelligence*, 2:10, 2026.

[13] Abouzar Kaboudian, Elizabeth M Cherry, and Flavio H Fenton. Real-time interactive simulations of large-scale systems on personal computers and cell phones: Toward patient-specific heart modeling and other applications. *Science advances*, 5(3):eaav6019, 2019.

[14] Benjamin J. Walker, Adam K. Townsend, Alexander K. Chudasama, and Andrew L. Krause. VisualPDE: Rapid interactive simulations of partial differential equations. *Bulletin of Mathematical Biology*, 85(11): 113, 2023. doi: 10.1007/s11538-023-01218-4.

[15] Darby I. Cairns, Maxfield Roth Comstock, Flavio H. Fenton, and Elizabeth M. Cherry. CardioFit: a WebGL-based tool for fast and efficient parametrization of cardiac action potential models to fit user-provided data. *Royal Society Open Science*, 12(8):250048, 2025. doi: 10.1098/rsos.250048.

[16] John P Berman, Abouzar Kaboudian, Ilija Uzelac, Shahriar Iravanian, Tinen Iles, Paul A Iaizzo, Hyunkyung Lim, Scott Smolka, James Glimm, Elizabeth M Cherry, et al. Interactive 3d human heart simulations on segmented human mri hearts. In *2021 Computing in Cardiology (CinC)*, volume 48, pages 1–4. IEEE, 2021.

[17] Abouzar Kaboudian, Elizabeth M Cherry, and Flavio H Fenton. Large-scale interactive numerical experiments of chaos, solitons and fractals in real time via gpu in a web browser. *Chaos, Solitons & Fractals*, 121:6–29, 2019.

[18] Abouzar Kaboudian, Elizabeth M Cherry, and Flavio H Fenton. Gpu load balancing using sparse cartesian grids: Making interactive webgl simulations of complex ionic models even faster on 3d heart structures. In *2023 Computing in Cardiology (CinC)*, volume 50, pages 1–4. IEEE, 2023.

[19] Abouzar Kaboudian, Richard A Gray, Ilija Uzelac, Elizabeth M Cherry, and Flavio H Fenton. Fast interactive simulations of cardiac electrical activity in anatomically accurate heart structures by compressing sparse uniform cartesian grids. *Computer methods and programs in biomedicine*, 257:108456, 2024.

[20] Khronos Group. WebGL 2.0 specification, 2017.

[21] Khronos Group. OpenGL ES Shading Language 3.00.6 specification, 2016. Accessed 2026-05-04.

[22] Varun Nair, Elliot Schumacher, Geoffrey Tso, and Anitha Kannan. DERA: Enhancing large language model completions with dialog-enabled resolving agents. In *Proceedings of the 6th Clinical Natural Language Processing Workshop*, 2024.

[23] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

[24] Zhuoyuan Wang, Hanjiang Hu, Xiyu Deng, Saviz Mowlavi, and Yorie Nakahira. OpInf-LLM: Parametric PDE solving with LLMs via operator inference. *arXiv preprint arXiv:2602.01493*, 2026.

[25] Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Daniel MacKinlay, Francesco Alesiani, Dirk Pflueger, and Mathias Niepert. PDEBench: An extensive benchmark for scientific machine learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[26] Ruben Ohana, Michael McCabe, Lucas Meyer, Rudy Morel, Fruzsina J. Agocs, Miguel Beneitez, Marsha Berger, Blakesley Burkhart, Keaton Burns, Stuart B. Dalziel, Drummond B. Fielding, Daniel Fortunato, Jared A. Goldberg, Keiya Hirashima, Yan-Fei Jiang, Rich R. Kerswell, Suryanarayana Maddu, Jonah Miller, Payel Mukhopadhyay, Stefan S. Nixon, Jeff Shen, Romain Watteaux, Bruno Régaldo-Saint Blancard, François Rozet, Liam H. Parker, Miles Cranmer, and Shirley Ho. The well: A large-scale collection of diverse physics simulations for machine learning. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, volume 37, 2024.

[27] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[28] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. In *International Conference on Learning Representations*, 2021.

[29] Maximilian Herde, Bogdan Raonić, Tobias Rohner, Roger Käppeli, Roberto Molinaro, Emmanuel de Bézenac, and Siddhartha Mishra. Poseidon: Efficient foundation models for PDEs. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

[30] Zhanhong Ye, Xiang Huang, Leheng Chen, Zining Liu, Bingyang Wu, Hongsheng Liu, Zidong Wang, and Bin Dong. PDEformer-1: A foundation model for one-dimensional partial differential equations. *arXiv preprint arXiv:2407.06664*, 2024.

[31] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[32] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[33] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

[34] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. *arXiv preprint arXiv:2308.10848*, 2023.

[35] Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. Multi-agent collaboration mechanisms: A survey of LLMs. *arXiv preprint arXiv:2501.06322*, 2025.

[36] Zhaoyue Xu, Long Wang, Chunyu Wang, Yixin Chen, Qingyong Luo, Hua-Dong Yao, Shizhao Wang, and Guowei He. CFDAgent: A language-guided, zero-shot multi-agent system for complex flow simulation. *Physics of Fluids*, 37(11):117124, 2025.

[37] Alireza Ghafarollahi and Markus J. Buehler. SciAgents: Automating scientific discovery through multi-agent intelligent graph reasoning. *arXiv preprint arXiv:2409.05556*, 2024.

[38] Qingpo Wuwu, Chonghan Gao, Tianyu Chen, Yihang Huang, Yuekai Zhang, Jianing Wang, Jianxin Li, Haoyi Zhou, and Shanghang Zhang. PINNsAgent: Automated PDE surrogation with large language models. *arXiv preprint arXiv:2501.12053*, 2025.

NeurIPS Paper Checklist

1. **Claims**

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

   Answer: [Yes]

   Justification: The abstract and introduction claim a multi-agent framework that turns PDE specifications into browser-executable WebGL solvers, and the highest accuracy and pass rate among the evaluated methods. Those results are supported by Section 3 and the results in Tables 2 to 4.

2. **Limitations**

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

   Answer: [Yes]

   Justification: The final paragraph of the Conclusion discusses single-precision WebGL textures, the restriction to explicit time stepping on grids, the solving difficulty with globally coupled problems such as Poisson equations.

3. **Theory assumptions and proofs**

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

   Answer: [N/A]

   Justification: The paper presents an empirical framework and benchmark study: it contains no theoretical result requiring proof. The equations given are the definitions of the PDE families and the scoring metric, not derived results.

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: Section 3 specifies the agent roles and the parse/code/verify/debug loop. Section A gives the full equations, domains, grids, time steps, and parameters for all five PDE families. Section A.5 gives the trial times, acceptance threshold, and debug budget. Sections C to E reproduce every agent and baseline prompt verbatim.

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

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

   Answer: [Yes]

   Justification: The code, WebGL skeletons, and reference-trajectory generation scripts are available at `https://anonymous.4open.science/r/web-pde-llm-anonymous-3B56/`.

6. **Experimental setting/details**

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

   Answer: [Yes]

   Justification: Section 3 states that train and test initial conditions are disjoint and that the shader is frozen before test evaluation. Section A.5 gives the ten independent trials per case, the debug trigger threshold $\epsilon = 0.1$, the revision cap $R = 5$, and the lowest-train-nRMSE selection rule, applied identically to all baselines. Section A.3 describes the 50 test initial conditions per PDE family and their sources.

7. **Experiment statistical significance**

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

Answer: [No]

Justification: Following the CodePDE baseline paper, each reported nRMSE is the solver with the lowest train error out of ten trials, evaluated on 50 held-out initial conditions, so a spread over trials would not be meaningful for that number. Run-to-run variability is instead reported as the pass rate over the same ten trials (Table 3).

8. **Experiments compute resources**

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

Answer: [Yes]

Justification: Table 4 reports input and output tokens per PDE and the end-to-end API cost per backbone at official prices, including failed runs, which is the dominant cost for all code-generating methods. Local execution is a consumer GPU through the browser, with solve times annotated in Figure 3 (for example 1.66 s for a Fenton–Karma run to $T = 100$); PINNsAgent additionally requires network training on a single GPU.

9. **Code of ethics**

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

Answer: [Yes]

Justification: The work uses only synthetic and published simulation data, involves no human subjects and no personal data, and the submitted version preserves anonymity. The authors have reviewed the NeurIPS Code of Ethics and find no deviation.

10. **Broader impacts**

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

Answer: [Yes]

Justification: The positive impact is lowering the setup barrier to scientific simulation, since generated solvers run in a browser on consumer hardware without a Python environment or specialized infrastructure, which helps teaching and low-resource settings. The corresponding risk is that an automatically generated solver can look convincing while being numerically wrong, which is why the framework validates against reference trajectories and reports pass rate alongside accuracy. The users should not treat generated solvers as verified without such checks, particularly for the cardiac models, which are research models and not clinical tools.

11. **Safeguards**

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

Answer: [N/A]

Justification: No pretrained model, generative model, or scraped dataset is released. The released artifacts are solver-generation code and synthetic PDE reference trajectories, which carry no such misuse risk.

12. **Licenses for existing assets**

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

Answer: [Yes]

Justification: PDEBench is cited as the source of the advection and Burgers' reference data [25], and the Fenton–Karma and TP06 model formulations are cited to their original publications [2–4]. The baseline frameworks are cited to their papers [9, 12, 38], and the license of each asset is recorded in the released code.

13. **New assets**

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

    Answer: [Yes]

    Justification: The new assets are the WEB-PDE-LLM framework and the five-family evaluation suite with its reference trajectories. Their documentation is in the paper (Sections A, A.3 to A.5 and C) and in the anonymized code release, which includes the discretizations, parameter files, and instructions for regenerating the reference solutions.

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

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

    Answer: [N/A]

    Justification: The paper involves no crowdsourcing and no human subjects. All evaluation is automated against numerical reference solutions.

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

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

    Answer: [N/A]

    Justification: The paper involves no human subjects and no participant data, so no IRB review or equivalent approval was required.

16. **Declaration of LLM usage**

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

    Answer: [Yes]

    Justification: LLMs are the core of the method: they act as the parse, checking, code, verify, and debug agents that produce and repair the WebGL shaders. Section 3 describes each role, Section C reproduces every prompt verbatim, and the four evaluated backbones (Gemini 3.5 Flash, GPT-5.5, Claude Opus 4.8, DeepSeek V4-Pro) are named throughout the results.

A Full PDEs and Score Formulas

This appendix gives the complete state equations for the two cardiac models, the data source, and the exact coverage-aware Score formulas.

A.1 Fenton–Karma

The Fenton–Karma (FK) model is a two-dimensional, three-variable nonlinear reaction–diffusion system consisting of the normalized transmembrane potential $u(t, \boldsymbol{x})$ and two non-diffusive gating variables $v(t, \boldsymbol{x})$ and $w(t, \boldsymbol{x})$. Only u contains a diffusion term. The model is

$$\begin{aligned}
\frac{\partial u}{\partial t} &= D\nabla^2 u - \frac{I_{\text{fi}}(u,v) + I_{\text{so}}(u) + I_{\text{si}}(u,w)}{C_m}, \\
\frac{\partial v}{\partial t} &= \frac{[1-\mathcal{H}(u-u_c)]\,(1-v)}{\tau_v^-(u)} - \frac{\mathcal{H}(u-u_c)v}{\tau_v^+}, \\
\frac{\partial w}{\partial t} &= \frac{[1-\mathcal{H}(u-u_c)]\,(1-w)}{\tau_w^-} - \frac{\mathcal{H}(u-u_c)w}{\tau_w^+}.
\end{aligned} \tag{1}$$

The three ionic currents are

$$\begin{aligned}
I_{\text{fi}}(u,v) &= -\frac{v\,\mathcal{H}(u-u_c)(u-u_c)(1-u)}{\tau_d}, \\
I_{\text{so}}(u) &= \frac{u\,[1-\mathcal{H}(u-u_c)]}{\tau_0} + \frac{\mathcal{H}(u-u_c)}{\tau_r}, \\
I_{\text{si}}(u,w) &= -\frac{w\left[1+\tanh\left(k(u-u_c^{\text{si}})\right)\right]}{2\tau_{\text{si}}}.
\end{aligned} \tag{2}$$

The voltage-dependent recovery time scale is

$$\tau_v^-(u) = [1-\mathcal{H}(u-u_v)]\,\tau_{v1}^- + \mathcal{H}(u-u_v)\tau_{v2}^-, \tag{3}$$

where $\mathcal{H}(\cdot)$ denotes the Heaviside function.

A.2 Ten Tusscher–Noble–Noble–Panfilov

The ten Tusscher–Noble–Noble–Panfilov (TNNP) model couples a diffusive transmembrane potential $V(t, \boldsymbol{x})$ to nineteen non-diffusive state variables. The potential evolves as

$$\partial_t V = D\,\nabla^2 V - \frac{1}{C_m}\,I_{\text{ion}}, \tag{4}$$

where the total ionic current sums twelve nonlinear components,

$$\begin{aligned}
I_{\text{ion}} = {} & I_{\text{Na}} + I_{\text{K1}} + I_{\text{to}} + I_{\text{Kr}} + I_{\text{Ks}} + I_{\text{CaL}} \\
& + I_{\text{NaCa}} + I_{\text{NaK}} + I_{\text{pCa}} + I_{\text{pK}} + I_{\text{bNa}} + I_{\text{bCa}}.
\end{aligned} \tag{5}$$

Writing $\phi = F/(RT)$, the Nernst reversal potentials are

$$\begin{aligned}
E_K &= \tfrac{1}{\phi}\ln\tfrac{K_o}{K_i}, \qquad E_{Na} = \tfrac{1}{\phi}\ln\tfrac{Na_o}{Na_i}, \\
E_{Ks} &= \tfrac{1}{\phi}\ln\tfrac{K_o + p_{KNa}Na_o}{K_i + p_{KNa}Na_i}, \qquad E_{Ca} = \tfrac{1}{2\phi}\ln\tfrac{Ca_o}{Ca_i},
\end{aligned} \tag{6}$$

and the individual currents are

$$
\begin{aligned}
I_{\text{Na}} &= G_{Na}\, m^3 hj\,(V - E_{Na}), \qquad I_{\text{to}} = G_{to}\, rs\,(V - E_K),\\
I_{\text{Kr}} &= G_{Kr}\sqrt{K_o/5.4}\; x_{r1} x_{r2}\,(V - E_K),\\
I_{\text{Ks}} &= G_{Ks}\, x_s^2\,(V - E_{Ks}), \qquad I_{\text{K1}} = G_{K1}\, x_{K1,\infty}(V)\,(V - E_K),\\
I_{\text{pK}} &= \frac{G_{pK}\,(V - E_K)}{1 + e^{(25-V)/5.98}}, \qquad I_{\text{pCa}} = \frac{G_{pCa}\, Ca_i}{K_{pCa} + Ca_i},\\
I_{\text{bNa}} &= G_{bNa}(V - E_{Na}), \qquad I_{\text{bCa}} = G_{bCa}(V - E_{Ca}),\\
I_{\text{NaK}} &= \frac{P_{NaK}\, K_o\, Na_i}{(K_o + K_{mK})(Na_i + K_{mNa})\,\Theta(V)},\\
I_{\text{CaL}} &= G_{CaL}\, d\, f\, f_2\, f_{cass}\, \Phi_{\text{GHK}}(V, Ca_{SS}),\\
I_{\text{NaCa}} &= \frac{k_{NaCa}\big(e^{\gamma V\phi} Na_i^3 Ca_o - \alpha\, e^{(\gamma-1)V\phi} Na_o^3 Ca_i\big)}{(K_{mNai}^3 + Na_o^3)(K_{mCa} + Ca_o)\big(1 + k_{sat}\, e^{(\gamma-1)V\phi}\big)},
\end{aligned}
\tag{7}
$$

where $\Theta(V) = 1 + 0.1245\, e^{-0.1V\phi} + 0.0353\, e^{-V\phi}$ and the L-type calcium driving term takes the Goldman–Hodgkin–Katz form

$$
\Phi_{\text{GHK}}(V, Ca_{SS}) = \frac{4(V-15)\,\phi F\,\big(0.25\, Ca_{SS}\, e^{2(V-15)\phi} - Ca_o\big)}{e^{2(V-15)\phi} - 1}. \tag{8}
$$

Each of the twelve voltage-dependent gates relaxes toward a voltage-dependent steady state under Hodgkin–Huxley kinetics and is advanced by an exponential (Rush–Larsen) update, while the SR release gate R and the five ion concentrations (Na_i, K_i, Ca_i, Ca_{SS}, Ca_{SR}) evolve from these currents together with SR release, uptake, leak, transfer, and rapid calcium buffering. The complete gate rate functions, concentration balances, and standard TP06 parameter values are released with our code.

A.3 Code and Data Source

Each PDE family is evaluated against 50 samples with different initial conditions. The reference solutions come from three sources. Advection and Burgers' are taken from PDEBench [25], so the initial conditions and reference trajectories are the published ones. Heat uses the exact solution: the periodic heat kernel is applied in Fourier space, which is analytic rather than numerical and explains why the LLM-only Python solvers reach 2.12×10^{-7} on this case. Fenton–Karma and TNNP have no closed form, so their references are generated from the published implementations of the two cardiac models [2–4]. The full codes and data are available at `https://anonymous.4open.science/r/web-pde-llm-anonymous-3B56/`.

A.4 Coverage-aware Score

With N_{pde} PDEs (here $N_{\text{pde}} = 5$), method m scores on PDE pde

$$
Q_{m,\text{pde}} = 100 \times \frac{\log e_{\text{worst,pde}} - \log e_{m,\text{pde}}}{\log e_{\text{worst,pde}} - \log e_{\text{best,pde}}}, \tag{9}
$$

where $e_{m,\text{pde}}$ is the nRMSE (or input token + output token) of method m on PDE pde, and $e_{\text{best,pde}} = \min_m e_{m,\text{pde}}$ and $e_{\text{worst,pde}} = \max_m e_{m,\text{pde}}$ are the lowest and highest values achieved by any method on that PDE. The best method on a PDE scores 100, the worst 0, and a failed ("–") result is assigned $Q_{m,\text{pde}} = 0$. The Score is the mean over all PDEs,

$$
\text{Score}_m = \frac{1}{N_{\text{pde}}} \sum_{\text{pde}=1}^{N_{\text{pde}}} Q_{m,\text{pde}}, \tag{10}
$$

where the method m ranges over the five compared frameworks (WEB-PDE-LLM, LLM-only, CodePDE, MCP-Sim, and PINNsAgent).

A.5 Experimental Protocol

Every PDE family, for every PDE solving framework and every LLM backbone, is run for 10 independent trials, and the solver with the lowest train nRMSE is the one carried forward to evaluation.

During the training phase of each trial, the debug agent, where the framework has one, is invoked when the solver fails to compile or when its train nRMSE exceeds $\epsilon = 0.1$, and is allowed at most $R = 5$ revision rounds. All baselines and WEB-PDE-LLM share the same experimental protocol with the same parameters.

B Additional Results

This appendix reports the full nRMSE, pass-rate and token-cost measurements, together with field comparisons and the full ablation table.

B.1 Test nRMSE

Table 2 reports test nRMSE for every combination of PDE, method, and backbone. On the four simpler PDEs the methods stay within an order of magnitude of each other, and on Heat several one-shot Python solvers agree to the reported precision. On FK and TNNP most baseline methods simply fail, which is what the coverage-aware Score is built to capture.

Table 2: Test nRMSE by PDE type, method, and LLM backbone. Each entry is averaged over the 50 test initial conditions of that PDE family. The best value in each column is shaded and the second best underlined, with cells that tie at the reported precision all shaded. "–" marks a case for which the method never produced a valid solver. Several LLM-only and CodePDE cells coincide to have similar nRMSE: CodePDE gives the model almost as much freedom as one-shot prompting, so a strong enough backbone should give similar solvers either way, for easy PDEs. Lower is better.

Test nRMSE (↓)	Advection	Burgers	Heat	FK	TNNP
WEB-PDE-LLM					
Gemini 3.5 Flash	2.56e-3	1.46e-3	3.89e-4	2.81e-4	1.32e-5
GPT-5.5	1.70e-3	7.24e-4	1.86e-6	2.80e-4	4.67e-6
Claude Opus 4.8	1.88e-3	1.46e-3	3.91e-4	9.57e-3	3.03e-3
DeepSeek V4-Pro	1.70e-3	1.16e-3	3.85e-5	6.95e-3	3.35e-6
LLM-only					
Gemini 3.5 Flash	1.36e-2	2.36e-2	2.12e-7	4.15e-4	–
GPT-5.5	1.36e-2	1.73e-2	2.12e-7	–	–
Claude Opus 4.8	1.32e-2	2.33e-2	2.12e-7	–	–
DeepSeek V4-Pro	1.32e-2	1.73e-2	2.13e-7	–	–
CodePDE					
Gemini 3.5 Flash	1.36e-2	2.36e-2	2.12e-7	4.15e-4	–
GPT-5.5	3.50e-2	1.74e-2	1.26e-6	3.12e-4	–
Claude Opus 4.8	1.32e-2	2.33e-2	1.27e-6	4.78e-3	–
DeepSeek V4-Pro	1.32e-2	1.72e-2	1.37e-6	2.79e-4	–
MCP-Sim					
Gemini 3.5 Flash	1.33e-2	3.75e-2	3.79e-4	–	–
GPT-5.5	1.36e-2	1.93e-2	1.79e-6	–	–
Claude Opus 4.8	1.36e-2	1.97e-2	3.79e-4	–	–
DeepSeek V4-Pro	9.59e-1	1.67e-2	7.81e-3	–	6.26e-1
PINNsAgent					
Gemini 3.5 Flash	1.89e-2	1.71e-1	1.24e-2	5.03e-1	–
GPT-5.5	1.01e-2	1.96e-1	7.85e-3	6.28e-1	–
Claude Opus 4.8	1.75e-2	2.41e-1	1.24e-2	7.27e-1	–
DeepSeek V4-Pro	2.72e-3	6.17e-1	8.69e-3	3.90e0	–

B.2 Pass rate

Accuracy is only meaningful on cases a framework can actually run, so Table 3 reports the fraction of trials for which the generated solver executes and produces output. The pattern follows the accuracy results: WEB-PDE-LLM is the only framework that runs on TNNP, CodePDE is reliable

everywhere else but never produces a TNNP solver, and MCP-Sim starts degrading as early as Burgers. PINNsAgent passes almost everywhere, but that records only that training completed, not that the result is accurate.

Table 3: Pass rate by PDE type, method, and LLM backbone, calculated over 10 independent trials per case. Trial counts as a pass if the generated solver compiles and runs to the full horizon T without crashing or producing NaN/Inf, regardless of its numerical accuracy. Higher is better.

Pass rate (↑)	Advection	Burgers	Heat	FK	TNNP
WEB-PDE-LLM					
Gemini 3.5 Flash	100.0%	100.0%	100.0%	90.0%	50.0%
GPT-5.5	100.0%	100.0%	100.0%	90.0%	90.0%
Claude Opus 4.8	80.0%	100.0%	100.0%	80.0%	100.0%
DeepSeek V4-Pro	90.0%	90.0%	100.0%	70.0%	80.0%
LLM-only					
Gemini 3.5 Flash	100.0%	80.0%	100.0%	90.0%	0.0%
GPT-5.5	100.0%	100.0%	100.0%	0.0%	0.0%
Claude Opus 4.8	100.0%	30.0%	80.0%	0.0%	0.0%
DeepSeek V4-Pro	100.0%	60.0%	100.0%	0.0%	0.0%
CodePDE					
Gemini 3.5 Flash	100.0%	90.0%	100.0%	100.0%	0.0%
GPT-5.5	100.0%	100.0%	100.0%	100.0%	0.0%
Claude Opus 4.8	100.0%	100.0%	100.0%	100.0%	0.0%
DeepSeek V4-Pro	100.0%	100.0%	100.0%	80.0%	0.0%
MCP-Sim					
Gemini 3.5 Flash	100.0%	80.0%	90.0%	0.0%	0.0%
GPT-5.5	80.0%	50.0%	100.0%	0.0%	0.0%
Claude Opus 4.8	80.0%	80.0%	70.0%	0.0%	0.0%
DeepSeek V4-Pro	90.0%	40.0%	80.0%	0.0%	10.0%
PINNsAgent					
Gemini 3.5 Flash	100.0%	100.0%	100.0%	100.0%	0.0%
GPT-5.5	100.0%	100.0%	100.0%	100.0%	0.0%
Claude Opus 4.8	100.0%	100.0%	100.0%	100.0%	0.0%
DeepSeek V4-Pro	100.0%	100.0%	100.0%	100.0%	0.0%

B.3 Token cost

Table 4 reports input/output tokens per PDE together with the total API cost of running each backbone over the whole suite, including failed runs. WEB-PDE-LLM spends 2–4× the tokens of one-shot generation on the simple cases, since parse validation, verification, and shader repair each issue additional calls, but ends up cheaper end-to-end than CodePDE or MCP-Sim, which burn roughly half a million tokens per backbone on TNNP without producing a working solver. PINNsAgent is cheapest in tokens because its LLM only proposes a training configuration; the cost moves to GPU training instead. Absolute figures depend heavily on the backbone: DeepSeek V4-Pro runs the whole suite for $1.39 despite emitting the most output tokens.

Table 4: Token cost by PDE type, method, and LLM backbone, as input/output tokens. The last column is the API cost in USD over all five PDEs, failed runs included, at official prices per 1M in/out (gemini-3.5-flash $1.50/$9.00, GPT-5.5 $5.00/$30.00, claude-opus-4-8 $5.00/$25.00, deepseek-v4-pro $0.435/$0.87). Counts are normalized to the 10-trial budget: PINNsAgent searches once and is scaled $\times 10$, and the three DeepSeek Heat runs that finished 5 of 10 trials are scaled $\times 2$. Lower is better.

Token cost (↓)	Advection	Burgers	Heat	FK	TNNP	Cost (USD)
WEB-PDE-LLM						
Gemini 3.5 Flash	19.3k/10.6k	20.1k/13.5k	20.6k/13.0k	65.1k/38.2k	553.0k/304.6k	$4.44
GPT-5.5	27.8k/13.0k	24.0k/14.2k	20.3k/10.6k	57.7k/33.8k	248.5k/184.3k	$9.56
Claude Opus 4.8	39.9k/38.3k	52.0k/37.9k	31.6k/34.8k	103.0k/78.4k	323.3k/251.1k	$13.76
DeepSeek V4-Pro	37.4k/141.7k	38.3k/201.4k	18.3k/69.9k	151.7k/308.7k	455.3k/520.0k	$1.39
LLM-only						
Gemini 3.5 Flash	6.2k/7.2k	6.3k/13.4k	6.5k/10.7k	14.5k/18.8k	56.2k/73.5k	$1.25
GPT-5.5	5.7k/7.2k	5.8k/9.7k	6.0k/34.8k	13.1k/14.4k	53.6k/60.5k	$4.22
Claude Opus 4.8	8.5k/16.3k	8.7k/17.9k	9.1k/13.7k	19.0k/21.2k	78.0k/96.6k	$4.76
DeepSeek V4-Pro	5.6k/33.1k	5.8k/83.3k	6.8k/388.6k	12.8k/98.2k	49.8k/235.2k	$0.76
CodePDE						
Gemini 3.5 Flash	9.3k/14.1k	25.7k/39.9k	9.7k/15.9k	17.7k/24.1k	477.4k/513.3k	$6.27
GPT-5.5	8.7k/19.7k	8.8k/22.8k	9.0k/78.0k	16.1k/26.7k	471.8k/513.8k	$22.40
Claude Opus 4.8	13.4k/23.7k	13.6k/27.1k	14.0k/25.0k	23.9k/38.9k	553.1k/627.4k	$21.64
DeepSeek V4-Pro	10.0k/39.1k	17.3k/93.9k	9.8k/344.7k	44.3k/206.4k	343.7k/1570.7k	$2.15
MCP-Sim						
Gemini 3.5 Flash	52.0k/39.3k	66.6k/50.8k	58.8k/47.5k	184.7k/167.5k	543.1k/511.5k	$8.71
GPT-5.5	76.4k/62.6k	72.0k/60.3k	66.2k/156.4k	175.9k/164.4k	565.1k/616.0k	$36.57
Claude Opus 4.8	97.7k/75.9k	125.0k/103.7k	114.9k/96.5k	252.5k/232.0k	549.2k/525.7k	$31.54
DeepSeek V4-Pro	47.6k/164.4k	76.5k/347.9k	60.7k/776.5k	136.2k/462.5k	390.7k/754.9k	$2.49
PINNsAgent						
Gemini 3.5 Flash	39.0k/4.2k	39.2k/4.3k	40.9k/4.2k	38.7k/4.2k	38.7k/4.1k	$0.48
GPT-5.5	33.9k/3.3k	34.0k/3.4k	35.4k/10.3k	33.6k/11.9k	33.6k/3.2k	$1.82
Claude Opus 4.8	49.8k/4.5k	49.9k/4.6k	52.2k/4.6k	49.6k/4.5k	49.3k/4.4k	$1.82
DeepSeek V4-Pro	33.5k/31.4k	33.5k/26.4k	37.3k/49.1k	35.5k/86.0k	33.2k/23.1k	$0.26

B.4 Field Comparisons

Figure 3 compares the solution fields themselves rather than the scalar nRMSE, using the Gemini 3.5 Flash backbone on a same initial condition. Every block pairs the method's field at the final time with its absolute error against the reference solution, drawn on a log color scale so that diverged runs remain readable side by side, and is annotated with its nRMSE and wall-clock solve time.

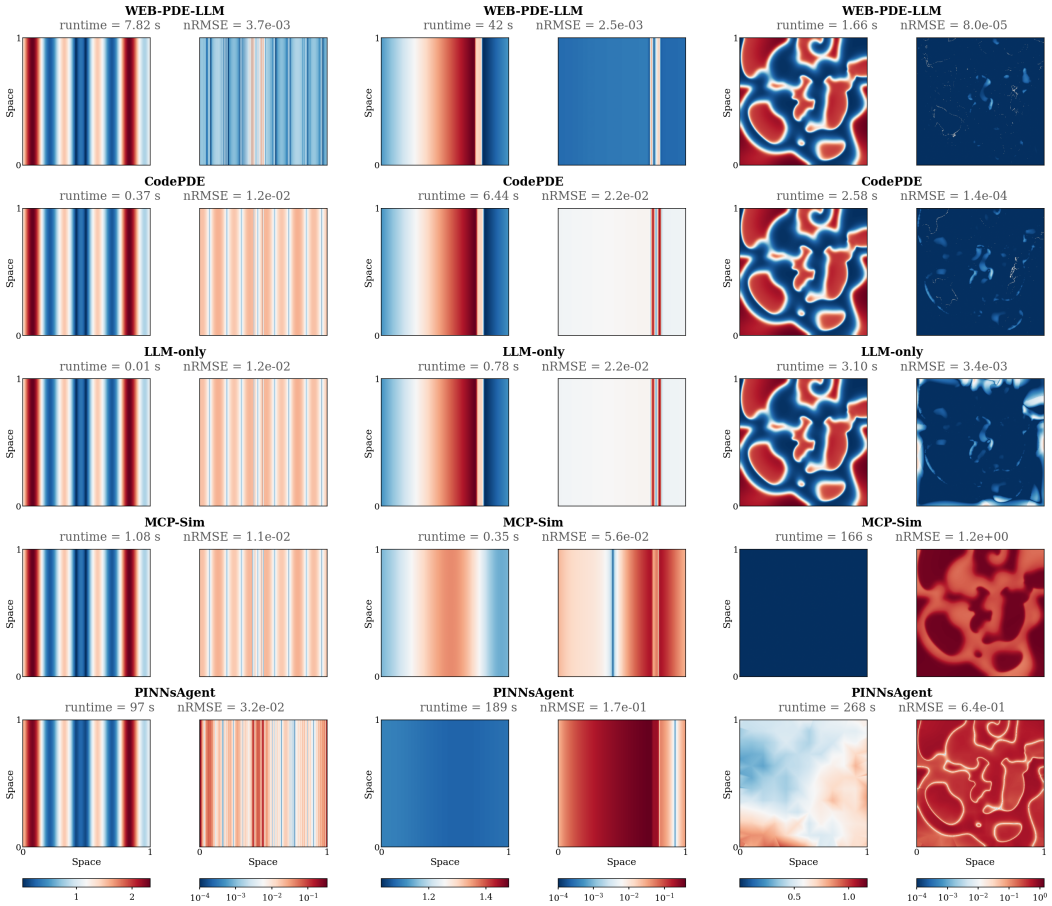

Figure 3: Solution fields on the Gemini 3.5 Flash backbone for one initial condition, with one column per PDE (Advection, Burgers', Fenton–Karma) and one block per method. Each block shows the final-time field beside its absolute error against the reference on a log scale, annotated with nRMSE and solve time. MCP-Sim smooths out Burgers' shock and returns an all-zero Fenton–Karma field, and PINNsAgent recovers neither the shock nor the wavefront. On FK the WEB-PDE-LLM solver reaches the horizon in 1.66 s (Section B.4), achieving real-time simulation.

B.5 Ablations

Dropping the parse-validation stage degrades accuracy on four of the five PDEs, most sharply on Heat, where nRMSE rises from 1.86×10^{-6} to 7.82×10^{-3}, and on FK, where it rises from 2.80×10^{-4} to 7.77×10^{-3}; the mean pass rate falls from 96.0% to 84.0%, confirming that explicitly checking the extracted Δt, boundary conditions, and parameter set catches unstable configurations the Code Agent would otherwise commit to. Additionally removing the Debug Agent leaves the Heat, FK, and TNNP cells unchanged but costs another order of magnitude on advection ($2.56 \times 10^{-3} \rightarrow 3.95 \times 10^{-2}$) and a further 4 points of pass rate, indicating that shader repair mostly recovers cases that compile-fail or diverge outright rather than refining already-correct kernels. The one cell that does not follow the trend is TNNP, where the ablated variants report a marginally lower nRMSE (3.82×10^{-6} versus 4.67×10^{-6}); with a 12-point lower pass rate on the same configuration, this reflects averaging over a smaller and easier set of surviving runs rather than a genuine accuracy gain.

Table 5: Ablation on the GPT-5.5 backbone, removing parse validation and the Debug Agent cumulatively. Lower nRMSE is better.

Test nRMSE (↓)	Advection	Burgers'	Heat	FK	TNNP	Mean Pass Rate (↑)
GPT-5.5						
WEB-PDE-LLM (full)	1.70e-3	7.24e-4	1.86e-6	2.80e-4	4.67e-6	96.0%
− Parse	2.56e-3	9.57e-4	7.82e-3	7.77e-3	3.82e-6	84.0%
− Parse − Debug	3.95e-2	1.33e-3	7.82e-3	7.77e-3	3.82e-6	80.0%

C WEB-PDE-LLM Prompts

The prompts for each agent in the parse/code/verify/debug loop, reproduced verbatim. Names in braces are placeholders filled in at call time: `{PDEs}` and `{bc}` come from the parsed JSON, and `{context_info}` carries the console log, WebGL error, or last-measured nRMSE. No held-out test trajectory, test reference value, or test nRMSE is included in the agent context. The ablation variants use the same prompts with the parsed fields replaced by the raw PDE text, so they are not repeated here.

System Prompt (all agents)

```
    You are an intelligent AI researcher for coding, numerical algorithms, and scientific
computing.
    Your goal is to conduct cutting-edge research in the field of PDE solving by
leveraging and creatively improving existing algorithms to maximize performances based on
feedbacks.
    Follow the user's requirements carefully and make sure you understand them.
    Always use Markdown cells to present your code.
```

Parse Agent

```
    You are given a clarified simulation specification intended for webGL.
    Your task is to parse this paragraph into a structured JSON object that captures all
key attributes needed for code generation.
    Output only the JSON. No talk, no markdown.


    ### JSON Schema Requirements:
    1. "PDEs": String. Full PDEs using LaTeX. IMPORTANT: Use double backslashes (e.g., \
nabla) for all LaTeX commands to ensure valid JSON escaping. When there is a new line, use
 \n.
    2. "number_of_state_variables": Integer.
    3. "texture_size": Integer (e.g., 512).
    4. "spatial_step": Float (the value of dx).
    5. "domain_size": Float (the size of the spatial domain, e.g., 20 for [-10,10]).
    6. "temporal_step": Float (the value of dt).
    7. "time_horizon": Float (the total simulation time, e.g., 100.0).
    8. "boundary_conditions": String or Object describing the conditions.
    9. "parameter_values": Object mapping each parameter to its initial value.
    10. "notes": String (optional, use null if no Important implementation notes)
    ### Example Output Format:
    {
    "PDEs": "\frac{\partial u}{\partial t} = D \nabla^2 u \n",
    "number_of_state_variables": 3,
    "texture_size": 512,
    "spatial_step": 0.01,
    "domain_size": 20.0,
    "temporal_step": 0.001,
    "time_horizon": 100.0,
    "boundary_conditions": "Periodic",
    "parameter_values": {"D": 0.001, "C_m": 1.0, "tau_pv": 7.99},
```

```
    }

    Input:
    {user_input}

    Output:
```

Parsed-Content Checking Agent

```
    Given this PDE description:
    {pde_desc}

    And this JSON parameter set:
    {json}

    Check if the parameters (especially temporal_step with CFL condition) are reasonable
for this PDE.
    Modify any values if needed for stability and accuracy.
    Return the complete modified JSON (or the same JSON if no changes needed).
```

Code Agent

```
    Given the {PDEs}, {coding_skeleton}, boundary condition {bc} and notes {notes}, return
 the complete glsl code implementation for the PDE solver in WebGL (do not put #version
300 es in your codes).
    Output only the raw glsl code. No talk, no markdown, just code.
```

Debug Agent

```
    The code {codes} is producing errors or high RMSE in this context: {context_info},
with the following pdes {PDEs}, boundary conditions {bc}, and parameter values:

    {parameter_values}

    {return_format}
```

D Baseline Prompts

Prompts for the three baselines that ask an LLM to write a solver, reproduced verbatim. LLM-only makes one generation call. CodePDE generates once and then repairs on failure, picking one of two debugging prompts depending on whether the run returned high numerical error or crashed, with the previous solver sent to the call. MCP-Sim clarifies the request, parses it to JSON, generates FEniCS code from that JSON, and diagnoses errors when a run fails. In each case `pde_description` is filled with the corresponding description from Section E, the same one WEB-PDE-LLM receives.

PINNsAgent is not listed here because it never asks the LLM for a solver. The PDE is fixed in its training code, and the LLM instead picks a configuration (network width and depth, activation, optimizer, sampling, and loss weights) from a predefined search space over a small number of iterations, guided by the scores of earlier trials [38]. Its prompt is therefore a hyperparameter-selection prompt rather than a code-generation prompt, and the solver quality it reports comes from network training rather than from anything the LLM writes.

LLM-only: System

```
    You are a helpful and precise assistant for solving PDEs. Your task is to implement
the solver function based on the provided template and description.
```

LLM-only: Generation

````
    You are an expert in scientific computing. Your task is to write a complete, efficient
 implementation of a PDE solver. Implement the `solver` function to solve the PDE. You
must not modify the function signature.

    ### 1. PDE Description
    {pde_description}

    ### 2. Solver Template
    ```python
    {solver_template}
    ```
````

CodePDE: System

```
    You are an intelligent AI researcher for coding, numerical algorithms, and scientific
computing.
    Your goal is to conduct cutting-edge research in the field of PDE solving by
leveraging and creatively improving existing algorithms to maximize performances based on
feedbacks.
    Follow the user's requirements carefully and make sure you understand them.
    Always document your code as comments to explain the reason behind them.
    Always use Markdown cells to present your code.
```

CodePDE: Generation

````
    Your task is to solve a partial differential equation (PDE) using Python in batch mode.

    {pde_description}

    You will be completing the following code skeleton provided below:

    ```python
    {solver_template}
    ```

    Your tasks are:
    1. Understand the above code samples.
    2. Implement the `solver` function to solve the PDE. You must not modify the function
signature.

    The 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 to evaluated. Make sure your `solver` function
runs correctly and efficiently.
    You can use PyTorch or JAX with GPU acceleration.
    You must use print statements for to keep track of intermediate results, but do not
print too much information. Those output will be useful for future code improvement and/or
 debugging.

    Your output must follow the following structure:

    1. A plan on the implementation idea.
    2. Your python implementation (modularized with appropriate auxiliary functions):

    ```python
    [Your implementation (do NOT add a main function)]
    ```

    You must use very simple algorithms that are easy to implement.
````

CodePDE: Debug (execution error or high numerical error)

````
    Thank you for your implmentation! When running the code, I got the following output
and error message:

    Code output: {code_output}

    Error message: {error_message}

    Can you think step-by-step to identify the root cause of the error and provide a
solution to fix it? Please provide a detailed explanation of the error and the solution
you propose. You can refer to the code implementation you provided earlier and analyze the
 error message to identify the issue.

    Your response should be in the following format:

    [Your rationale for debugging (think step-by-step)]

    ```python
    [Your bug-free implementation]
    ```
````

CodePDE: Debug (NaN or inf)

````
    Thank you for your implmentation! After running the code, the error becomes NaN or inf.


    The followings are the output and error message:

    Code output: {code_output}

    Error message: {error_message}

    Can you think step-by-step to identify the root cause of the error and provide a
solution to fix it? You should check if any computation in the code is numerically stable
or not. You should also consider to use smaller step sizes.

    Your response should be in the following format:

    [Your rationale for debugging (think step-by-step)]

    ```python
    [Your bug-free implementation]
    ```
````

MCP-Sim: Input Clarifier

```
    You are given a user's simulation request in natural language. Based on the following
guidelines, generate a **single-paragraph, fully specified simulation description**
suitable for direct use in FEniCS.

    Carefully consider the following aspects when clarifying:

    Problem Type and Structure
    - Identify the PDE type: heat, fluid (Navier-Stokes), elasticity, fracture, reaction-
diffusion, etc.
    - Specify whether the problem is steady or time-dependent.
    - Determine if it is multiphysics (e.g., thermo-elasticity, fluid-structure
interaction, electro-mechanics).
    - Identify spatial dimension: 2D or 3D.
    - Describe the domain shape: rectangle, circle, cylinder, presence of notch, etc.
```

```
    Field Variables and Conditions
    - Define field variables such as u, p, T, d, k, etc.
    - Infer and describe boundary and initial conditions clearly.
    - Estimate required material properties: E, nu, k, rho, cp, mu, Gc, etc.

    Numerical Settings
    - Suggest appropriate time step dt (if transient).
    - Recommend solver structure (e.g., nonlinear Newton solver, staggered scheme).
    - Mention output format (e.g., whether to store results in .xdmf).

    Format your output as one complete paragraph in clear technical English.
    Do NOT include section headings or bullet points.

    [User Request]
    {raw_input}

    [Refined Simulation Specification]
```

MCP-Sim: Parsing

```
    You are given a clarified simulation specification intended for FEniCS.
    Your task is to parse this paragraph into a structured JSON object that captures all
key attributes needed for code generation.

    The output must be a valid JSON with the following fields:
    - problem_type (e.g., heat, fluid, elasticity, hyperelasticity, fracture)
    - pde_description (a short descriptive sentence)
    - solver_template (a code skeleton with placeholders for the solver function)
    - dimension (1, 2, or 3)
    - domain (shape description)
    - domain_geometry_file (optional, use null if not needed)
    - mesh (object with fields: nx, ny)
    - variables (e.g., ["u"], ["u", "p"], ["u", "d"])
    - time_dependent (true/false)
    - nonlinear (true/false)
    - coupled (true/false)
    - boundary_conditions (list of Dirichlet or Neumann conditions)
    - initial_conditions (initial values for each variable)
    - source_terms (list, or empty array [])
    - material_properties (dictionary of physical parameters)
    - notes (optional field for special considerations)

    Output only the JSON object. Do not include any explanation, markdown, or code block.

    Input:
    {clarified_input}

    Output:
```

MCP-Sim: Code Builder

```
    You are an expert in generating FEniCS-based simulation code.

    You are given a parsed JSON object describing the problem, along with the full
clarified natural language description.

    You must generate valid, executable Python code using classical FEniCS (do not use
dolfinx), referencing BOTH the parsed JSON and the full_text. Implement the `solver`
function to solve the PDE. You must not modify the function signature.

    If any important information is missing in the parsed JSON (e.g., complex geometry,
fiber arrangements, custom material properties), infer and supplement it from the
full_text.
```

```
    Always prioritize correctness, physical realism, and FEniCS best practices.

    Supported problem types:
    - 'fluid': Navier-Stokes
    - 'heat': heat transfer
    - 'elasticity': linear elasticity
    - 'hyperelasticity': hyperelastic materials
    - 'phase_field_fracture': nonlinear coupled system with damage variable d (range 0-1)

    [Code generation requirements]
    - Do NOT use Markdown formatting. Return plain Python code only.
    - Do not print anything during module import or solver execution. Do not include
progress logs, status messages, shape prints, timing prints, or "simulation complete"
messages. The solver must be silent unless it raises an exception.
    - Follow this structure:
      1. Import libraries
      2. Define geometry and mesh (construct square shape)
      3. Define function spaces (combine or separate for u/d in phase-field)
      4. Apply boundary and initial conditions
      5. Define weak form F and Jacobian J; use `solve()`
         - For time-dependent heat problems:
           Use Backward Euler by default
           Weak form: F = u*v*dx + dt*k*dot(grad(u), grad(v))*dx - u_n*v*dx
           Do NOT include grad(u_n) in any term
      6. If problem is time-dependent, include a time-stepping loop:
         - For phase_field_fracture: alternate solving u and d
         - Save result each step using XDMF format
      7. Save outputs (.xdmf files per field, with timestep info if applicable)

    [Code guidelines]
    - Output must be a fully executable Python script (.py), no markdown or explanation
    - Use `solve(F == 0, ...)` and include `solver_parameters` (e.g., max iterations,
relaxation)
    - Prefer `MUMPS` as linear solver for nonlinear problems
    - Use `+ eps` for denominators to improve numerical stability (e.g., `epsilon + eps`)

    Input Parameters:
    {parsed_data}
```

MCP-Sim: Error Diagnosis

```
    You are a diagnostic agent specialized in identifying and resolving errors in Python
code for FEniCS-based simulations.

    Your task is to:
    1. Analyze the simulation output and error logs.
    2. Accurately identify the root cause of failure in the original code.
    3. Return the corrected full version of the code as plain Python (no markdown).

    Output Format (must follow this JSON schema):
    {{
      "fix_type": "parsing" or "code",
      "hint": "Explanation of the error and how it was fixed",
      "after_code": "Modified full Python code",
      "confidence": float between 0.0 and 1.0
    }}

    Input Data:
    [Error Message]
    {error_message}

    [Simulation Output Log]
    {simulation_output}

    [Original Code]
```

```
    {code}
```

E PDE Descriptions

Every framework receives the same natural-language PDE description except for necessary channel setup in WebGL textures, reproduced below. Each description states the governing equations, the domain, the required solver signature, the parameter values, and the space-time steps. The TNNP description is too long to reproduce in full.

PDE Description: Advection

```
    The PDE system is a 1D advection model:

    \begin{equation}
    \begin{cases}
    \partial_t u(t, x) = -\beta \partial_x u(t,x) \\
    \end{cases}
    \end{equation}

    where $x \in (0,1)$ and $t \in (0,T]$. The spatial domain is $\Omega = [0,1]$.

    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 the state variables
for the specified subsequent time steps ($t = t_1, \dots, t_T$). The solver outputs $u$,
each of shape $[batch\_size, T+1, N]$ (including the initial time frame and subsequent
steps).

    To maintain stability, internal time-stepping smaller than the required output
intervals should be considered.

    Model Parameters are: $\beta=0.1$

    **Important implementation notes**:
    - Use 1024 interior spatial points
    - The domain is $x \in [0,1]$ with $dx \approx 9.765625 \times 10^{-4}$
    - Time horizon is $T=2.0$ with $dt < 1.0 \times 10^{-3}$
    - Use periodic boundary conditions.
```

PDE Description: Burgers'

```
    The PDE system is a 1D Burgers' model:

    \begin{equation}
    \begin{cases}
    \partial_t u(t, x) + \partial_x (u(t,x)^2) / 2 = \nu / 3.1415926 * \partial_{xx} u(t,x
) \\
    \end{cases}
    \end{equation}

    where $x \in (0,1)$ and $t \in (0,T]$. The spatial domain is $\Omega = [0,1]$.

    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 the state variables
for the specified subsequent time steps ($t = t_1, \dots, t_T$). The solver outputs $u$,
each of shape $[batch\_size, T+1, N]$ (including the initial time frame and subsequent
steps).

    To maintain stability, internal time-stepping smaller than the required output
intervals should be considered.

    Model Parameters are: $\nu=0.001$
```

```
    **Important implementation notes**:
    - Use 1024 interior spatial points
    - The domain is $x \in [0,1]$ with $dx \approx 9.765625 \times 10^{-4}$
    - Time horizon is $T=2.0$
    - Use periodic boundary conditions.
```

PDE Description: Heat

```
    The PDE system is a 2D heat equation for the temperature field $u(t,x,y)$:

    \begin{equation}
    \partial_t u(t,x,y) = \alpha \left(\partial_{xx} u(t,x,y) + \partial_{yy} u(t,x,y)\
right)
    \end{equation}

    where $(x,y) \in (0,1)^2$ and $t \in (0,T]$. The spatial domain is $\Omega = [0,1]^2$.

    Given the discretization of $u(0,x,y)$ of shape $[batch\_size, N, N]$, where $N$ is
the number of spatial points in each direction, implement a solver to predict the
temperature field for the specified subsequent time steps ($t = t_1, \dots, t_T$). The
solver outputs $u$ with shape $[batch\_size, T+1, N, N]$ including the initial frame.

    Model Parameters are: $\alpha=0.01$

    **Important implementation notes**:
    - Use 128 spatial points in each direction.
    - The domain is $[0,1)^2$ with $dx = dy = 1/128 = 7.8125 \times 10^{-3}$.
    - Time horizon is $T=2.0$.
    - Use a stable internal temporal step $dt < 1 \times 10^{-3}$.
    - Boundary conditions: periodic in both spatial directions.
```

PDE Description: Fenton–Karma

```
    The PDE system is a 2D reaction-diffusion model, describing the evolution of the
normalized transmembrane potential $u(t, x)$ and two gating variables $v(t, x)$ and $w(t,
x)$:

    \begin{equation}
    \begin{cases}
    \partial_t u(t, x) = D \nabla^2 u(t, x) - \frac{I_{fi}(u, v) + I_{so}(u) + I_{si}(u, w
)}{C_m} \\
    \partial_t v(t, x) =
        \begin{cases}
        \frac{1-v(t, x)}{\tau_{mv}(u)} & u(t, x) < V_c \\
        \frac{-v(t, x)}{\tau_{pv}} & u(t, x) \ge V_c
        \end{cases} \\
    \partial_t w(t, x) =
        \begin{cases}
        \frac{1-w(t, x)}{\tau_{mw}} & u(t, x) < V_c \\
        \frac{-w(t, x)}{\tau_{pw}} & u(t, x) \ge V_c
        \end{cases}\\
    \tau_{mv}(u) =
        \begin{cases}
        \tau_{v1} & u(t, x) < V_v \\
        \tau_{v2} & u(t, x) \ge V_v
        \end{cases}\\
    I_{fi}(u, v) = \frac{-v(t, x) \cdot H(u(t, x) - V_c) \cdot (u(t, x) - V_c) \cdot (1 -
u(t, x))}{\tau_d} \\
    I_{so}(u) = \frac{u(t, x)(1 - H(u(t, x) - V_c))}{\tau_0} + \frac{H(u(t, x) - V_c)}{\
tau_r} \\
    I_{si}(u, w) = \frac{-w(t, x)(1 + \tanh(k(u(t, x) - V_{csi})))}{2\tau_{si}}\\
    H(x) =
        \begin{cases}
```

```
        0 & x < 0 \\
        1 & x \ge 0
        \end{cases}
    \end{cases}
    \end{equation}

    where $x \in (0,10)^2$ and $t \in (0,T]$. In our task, we assume No-Flux (Neumann)
boundary conditions. The spatial domain is $\Omega = [0,10]^2$.

    Given the discretization of $u(0, x), v(0, x), w(0, x)$ each of shape $[batch\_size, N,
 N]$ where $N$ is the number of spatial points, you need to implement a solver to predict
the state variables for the specified subsequent time steps ($t = t_1, \dots, t_T$). The
solver outputs $u, v, w$, each of shape $[batch\_size, T+1, N, N]$ (including the initial
time frame and subsequent steps).

    To maintain stability, internal time-stepping smaller than the required output
intervals might be considered.

    Model Parameters are: $D = 0.001$, $C_m = 1.0, \tau_{pv}=7.99, \tau_{v1}=9.8, \tau_{v2
}=312.5, \tau_{pw}=870.0, \tau_{mw}=41.0, \tau_{0}=12.5, \tau_{r}=33.83, \tau_{si}=29.0, k
=10.0, V_{csi}=0.861, V_c=0.13, V_v=0.04, \tau_{d}=0.5714$

    **Important implementation notes**:
    - Use a $512 \times 512$ spatial grid
    - The domain is $x \in [0,10]$ with $dx \approx 1.953125 \times 10^{-2}$
    - Time horizon is $T=100.0$ with $dt = 2.5 \times 10^{-2}$
    - Use No-Flux (Neumann) boundary conditions.
```

PDE Description: TNNP

```
    The PDE system is a 19-state-variables model representing the Ten Tusscher-Noble-Noble-
Panfilov (TP06) human ventricular action potential with 19 state variables.

    \begin{equation}
    \begin{split}
    AM &= \frac{1}{1 + \exp\left(\frac{-60 - V}{5}\right)} \\
    BM &= \frac{0.1}{1 + \exp\left(\frac{V + 35}{5}\right)} + \frac{0.10}{1 + \exp\left(\
frac{V - 50}{200}\right)} \\
    minft &= \frac{1}{\left(1 + \exp\left(\frac{-56.86 - V}{9.03}\right)\right)^2} \\
    \tau_M &= AM \cdot BM, \qquad exptaumt = \exp\left(-\frac{dt}{\tau_M}\right) \\
    sm &= minft - (minft - sm) \cdot exptaumt
    \end{split}
    \end{equation}

    [... 137 lines omitted ...]

    Model Parameters are: $Ko = 5.4$, $Cao = 2.0$, $Nao = 140.0$, $Vc = 0.016404$, $GNa =
14.838$, $GbNa = 0.00029$, $KmK = 1.0$, $KmNa = 40.0$, $knak = 2.724$, $GCaL = 0.00003980$,
 $GbCa = 0.000592$, $knaca = 1000.0$, $KmNai = 87.5$, $KmCa = 1.38$, $ksat = 0.1$, $n =
0.35$, $GpCa = 0.1238$, $KpCa = 0.0005$, $GpK = 0.0146$, $diffCoef = 0.001$.

    - Use 512 interior spatial points.
    - Spatial domain $x \in [0,12]^2$ with $dx = 0.0234375$.
    - Time horizon is $T=100.0$ with $dt = 2.5 \times 10^{-2}$
    - Boundary conditions: No-Flux (Neumann)

    **Important implementation notes for WEB-PDE-LLM only**:
    Texture 1: ($V$, $sRR$, $Nai$, $Ki$); Texture 2: ($Cai$, $CaSS$, $CaSR$, $ISumCa$);
Texture 3: ($sm$, $sh$, $sj$, $sxs$); Texture 4: ($sd$, $sf$, $sf2$, $sfcass$); Texture 5:
 ($sr$, $ss$, $sxr1$, $sxr2$).
    You must translate the entire PDE mathematical description into the WebGL shader, step-
by-step, with absolutely NO simplifications or structural omissions.
```