Reasoning With a Star: A Heliophysics Dataset and Benchmark for Agentic Scientific Reasoning

Kevin Lee[1,2*] **Russell Spiewak**[1,3] **James Walsh**[1,3,4]
[1]Frontier Development Lab, Frisco, TX, USA
[2]Department of Mechanical and Aerospace Engineering, UCLA, Los Angeles, CA, USA
[3]Trillium Technologies Inc., Frisco, TX, USA
[4]Department of Engineering, University of Cambridge, Cambridge, UK

Abstract

Scientific reasoning through Large Language Models in heliophysics involves more than just recalling facts: it requires incorporating physical assumptions, maintaining consistent units, and providing clear scientific formats through coordinated approaches. To address these challenges, we present Reasoning With a Star, a newly contributed heliophysics dataset applicable to reasoning; we also provide an initial benchmarking approach. Our data are constructed from National Aeronautics and Space Administration & University Corporation for Atmospheric Research Living With a Star summer school problem sets and compiled into a readily consumable question-and-answer structure with question contexts, reasoning steps, expected answer type, ground-truth targets, format hints, and metadata. A programmatic grader checks the predictions using unit-aware numerical tolerance, symbolic equivalence, and schema validation. We benchmark a single-shot baseline and four multi-agent patterns, finding that decomposing workflows through systems engineering principles outperforms direct prompting on problems requiring deductive reasoning rather than pure inductive recall.

1 Introduction

Scientific problem solving rarely fits into a single logical leap, especially in fields requiring deep prior knowledge. Progress usually requires domain expertise, institutional resources, iterative refinements, and validation of assumptions. This coincides with systems engineering practices, which emphasize well-defined interfaces, requirements, and verifications [1]. Meanwhile, Large Language Models (LLMs) exhibit algorithmic limitations in reasoning, causing "reasoning illusions" [2] and algebraic failures. This motivates multi-step, role-based reasoning approaches with chain-of-thought [3], tree-of-thoughts [4], and graph-of-thoughts [5], emulating how scientists iteratively formulate hypotheses, refine models, and validate results, over one-shot guessing.

To bridge this gap, we present Reasoning With a Star (RWS)[†], a benchmark dataset for LLM- and agent-based reasoning in heliophysics derived from NASA/UCAR Living With a Star (LWS) Summer School problem sets [6, 7]. Drawing from systems engineering design practices in mission-critical environments, we evaluate multi-agent systems designed with manageable complexity through the Systems-engineering-of-Thoughts Agentic Reasoning (STAR, Appendix B.1). Using a programmatic grader for symbolic equivalence, unit-aware numerical tolerance, and schema validation, our experiments show that no single agentic pattern excels in all scenarios.

[*]Corresponding author. Email: kevinlee69720@g.ucla.edu
[†]Available at: Hugging Face - SpaceML/ReasoningWithAStar

Machine Learning and the Physical Sciences Workshop, NeurIPS 2025.

We assessed LLM- and agent-based reasoning without retrieval-augmented generation (RAG). Beyond single-shot prompting, we evaluated four agentic patterns: hierarchical multi-agent workflows (HMAW) [8], plan answer critique enclose (PACE), plan hypothesize analyze solve evaluate (PHASE), and a systems-engineering-inspired expert system, informed by prior research [9, 10] (SCHEMA). These design philosophies range from self-critique loops to assumption and requirement tracking. We find no single pattern excels in all scenarios; compact pipelines perform well on arithmetic tasks, while sophisticated patterns are better suited for methodological formulation and validation tasks.

Our contributions are threefold: (i) a science-focused benchmark; (ii) a benchmark grader adaptable to different task formats; and (iii) a comparative study of single-shot and multi-agent reasoning.

2 Background and Related Work

Agents are modular units that guide LLMs through specialized prompts or tools, enabling interaction with external data and the execution of complex tasks beyond text generation. When composed into workflow patterns, agents collaborate through planning, solving, verifying, or refining to support multi-step tasks beyond single-shot prompting.

A core mantra in systems engineering is that complexity must be earned, not assumed [11]. Excess complexity increases failure modes and reduces maintainability [1]. Our agentic designs follow this principle by prioritizing clear interfaces, compact workflows, and careful scaling, as demonstrated by SCHEMA's limited use of expert agents and assumption tracking. PACE and PHASE explore the trade-offs between complexity and performance through self-critique and hypothesis formation.

We evaluated RWS alongside established reasoning benchmarks: GSM8K [12]; MATH [13]; GPQA [14]; HumanEval [15]; and SWE-bench [16]. While LWS is designed to educate heliophysics researchers, RWS is well suited for training and evaluating LLMs in scientific reasoning.

3 Dataset

RWS Dataset, Symbolic Instance

Problem: Interstellar gas enters the heliosphere under the influence of solar gravity, radiation pressure, and ionization losses. The resulting neutral atom density is $n(r, \theta)$, where r is heliocentric radial distance and θ is the angle of the heliocentric position vector relative to the bulk inflow velocity of the atoms. We may assume that the ionization rate per atom is $\beta_0 \left(r_0/r\right)^2$. When an atom is ionized, it has a speed approximately equal to the solar wind speed V in the frame of the solar wind. We assume that these ions are immediately picked up by the solar wind via gyration and pitch-angle scattering to form an isotropic shell of speed V in the solar wind frame.

Assuming that the pitch-angle scattering rate is so large that the spatial diffusion tensor is negligible, write down the Parker equation for the evolution of the pickup ion omnidirectional distribution function $f(r, \theta, v)$ with an appropriate source term. We assume that the configuration is stationary and that the solar wind has constant speed and spherical symmetry.

Solution:

Step 1: $\frac{\partial f}{\partial t} + \mathbf{V} \cdot \nabla f - \frac{1}{3} \nabla \cdot \mathbf{V} v \frac{\partial f}{\partial v} = \beta_0 \left(\frac{r_0}{r}\right)^2 n(r, \theta) \frac{\delta(v-V)}{4\pi v^2}$

Step 2: When integrated over d^3v, the RHS gives the rate of pickup ion generation by ionization: $\frac{\partial n_{pi}}{\partial t} = \beta_0 \left(\frac{r_0}{r}\right)^2 n(r, \theta)$

Step 3: Under the specified assumptions, $\frac{\partial f}{\partial r} - \frac{2}{3r} v \frac{\partial f}{\partial v} = \beta_0 \left(\frac{r_0}{r}\right)^2 n(r, \theta) \frac{\delta(v-V)}{4\pi V^3}$

Final: $$\boxed{\frac{\partial f}{\partial r} - \frac{2}{3r} v \frac{\partial f}{\partial v} = \beta_0 \left(\frac{r_0}{r}\right)^2 n(r, \theta) \frac{\delta(v-V)}{4\pi V^3}}$$

Figure 1: Example symbolic item from the Reasoning With a Star (RWS) dataset, drawn from `2010_Lee_hw.pdf` [7], showing a problem, reasoning steps, and a LaTeX final expression.

Heliophysics is the study of how the Sun affects Earth, planetary environments, and space weather, with consequences for climate, technological systems, and operational safety in space and on the ground. Despite the field's importance for satellites, communication networks, and other critical infrastructure, heliophysics remains underrepresented in reasoning benchmarks for LLMs. RWS addresses this gap by providing a benchmark for scientific reasoning rather than mere fact recall.

Using an OCR-based method, we converted LWS Summer School problem sets into a machine-readable format, manually cleaned up scanning errors, symbol and unit misreads, and typos, and then normalized the results into a JSON Lines (JSONL) dataset following the schema in Appendix A.1. The resulting RWS dataset contains 158 question–answer pairs authored by heliophysicists (contributors listed in Section A.2).

For each item, we store the problem statement (and, when applicable, a preamble summarizing previous sub-questions), a sequence of intermediate reasoning steps describing the expert solution, and the final answer in the `final` field, together with a `type` label, an optional `hint` specifying the expected output format (e.g., required units, JSON structure, or LaTeX equation), and a `meta` container for metadata. Required physical assumptions (e.g., adiabatic expansion, constant acceleration, neglect of certain loss terms) are preserved in the question and step text. The `step` field serves as a reference reasoning process trace for analyzing model behavior and benchmark performance, but models are evaluated only on the `final` answer. The `final` field spans three types of ground truth: (i) *numeric* answers, where the model must return one or more scalar values in the required physical units (38 items); (ii) *symbolic* answers, where the model must produce a LaTeX-formatted algebraic expression or equation (52 items); and (iii) *textual* answers, where the model must provide a scientific phrase or qualitative statement (68 items). These categories reflect the range of outputs expected in heliophysics problem-solving, from quantitative estimates with units to derivations and physical explanations.

4 Benchmark

Our evaluation of RWS includes both single-shot baselines and multi-agent patterns. For the single-shot setup, we tested the base models Google Gemini 2.5 Pro [17], OpenAI OSS 20B and OSS 120B [18], Meta Llama 3.3 [19], and Mistral 24.11 [20]; results are reported in Table 1. To compare performance across different multi-agent patterns, we also evaluated their performance on established benchmarks, including GSM8K [12], MATH [13], GPQA [14], HumanEval [15], and SWE-bench [16]. These cross-benchmark results are shown in Table 2.

4.1 Methods

We benchmarked four distinct agentic patterns: HMAW [8], a lightweight hierarchical CEO/manager/-worker handoff, PACE, which generates an answer and then performs a self-critique loop, PHASE, which adds an explicit hypothesis stage before solving, and SCHEMA, a systems engineering inspired expert allocation strategy informed by prior multi-agent architectures [9, 10].

Each pattern decomposes the task across role-specific agents, passes intermediate outputs between them, and produces a final answer. We also include a *single-shot* baseline, in which a single LLM directly generates the final answer without coordination.

For RWS, we assess it without heliophysics-specific RAG; both single-shot and agentic pipelines must solve problems with only the provided problem statement. This setup isolates scientific reasoning ability, such as deriving relationships, propagating units, and stating assumptions, rather than rewarding access to external domain knowledge.

We score model outputs with a grader aligned with the RWS metadata. Each RWS item is labeled with an expected answer type (numeric, symbolic, or textual), required units, and formatting constraints. We evaluate: (i) *numeric* answers by checking whether predicted scalar values match the ground truth within an acceptable error bound and include the required units; (ii) *symbolic* answers using a computer algebra system (CAS; e.g., SymPy [21]) to verify algebraic equivalence to the LaTeX expression; and (iii) *textual* answers by verifying that they provide scientifically accurate statements that satisfy required assumptions.

When the automatic grader flags a mismatch; for instance, algebraically equivalent expressions that fail a strict string match or paraphrased scientific statements, we apply an additional verifier built from two LLM agents running Gemini 2.5 Pro at temperature $= 1.0$.

The *Parser* agent extracts and normalizes the model output and the ground-truth answer (e.g., strips `$...$`, canonicalizes units, and produces `pred_norm`, `gt_norm`, and `type`).

The *Judge* agent then applies a type-specific check. For numeric answers, it enforces a 5% error tolerance and correct units. For symbolic answers, it decides whether two expressions are algebraically equivalent. Finally, for textual answers, it tests for strict semantic equivalence.

Unless otherwise noted, we report accuracy under this evaluator. Multi-agent systems and single-shot baselines are run under matched decoding conditions. No tools are used except for coding benchmarks, where we reuse the Docker-based evaluation harness (e.g., for *SWE-bench Verified*) and report the resolution rate. Our multi-agent system interacts with the repository through an adapter to *SWE-agent* [22], employing file-editing tools to construct compatible software patches.

4.2 Results

Gemini 2.5 Pro achieves the highest single-shot accuracy on RWS, as shown in Table 1. Among open-source models, OpenAI OSS 20B and OSS 120B perform next best, followed by Meta Llama 3.3 and Mistral 24.11. This establishes a reasonably strong direct-prompting baseline without any multi-agent orchestration. We use this baseline to contextualize the gains from the coordination strategies.

Table 1: Single-shot accuracy (%) on RWS across models (best in bold).

Model	Accuracy
Google Gemini 2.5 Pro	**35.44**
OpenAI OSS 20B	32.91
OpenAI OSS 120B	32.91
Meta Llama 3.3	31.01
Mistral 24.11	27.22

Table 2: Accuracy (%) by agent design pattern on each benchmark under Google Gemini 2.5 Pro (best per row in bold). The bottom row reports the unweighted macro-mean across datasets.

Dataset	HMAW	PACE	PHASE	SCHEMA
GSM8K	91.05	**93.41**	92.35	86.36
MATH	78.31	**81.51**	77.84	71.39
GPQA	**79.01**	77.10	77.16	73.36
RWS	39.52	41.92	42.51	**44.31**
HumanEval	30.49	37.80	35.98	**43.29**
SWE-bench Verified	53.81	55.70	60.54	**63.23**
Macro-mean	62.03	*64.57*	*64.40*	*63.66*

Table 2 summarizes accuracy by agent pattern in GSM8K, MATH, GPQA, RWS, HumanEval, and SWE-bench, all under Gemini 2.5 Pro as the base model. No single coordination pattern dominates all tasks.

PACE achieves the highest accuracy on GSM8K and MATH. These datasets emphasize multi-step arithmetic problem solving; PACE's compact plan–answer–critique loop shows that a lightweight self-critique pipeline [23–26] is often enough to correct routine algebraic or calculation errors without adding extra coordination overhead.

HMAW leads in GPQA. GPQA involves graduate-level science QA, where a simple hierarchical manager/worker handoff appears sufficient to maintain focus on classification-style judgments and factual recall. This indicates that more complex hypothesis stages or detailed assumption tracking offer only marginal additional benefit for narrowly defined question-answering tasks.

SCHEMA performs best on HumanEval, SWE-bench Verified, and RWS. All three settings need outputs that meet specific format and constraint requirements. HumanEval demands executable code that passes reference tests, SWE-bench Verified requires task-specific bug fixes that maintain overall correctness in the repository, and RWS requires physically consistent answers with the correct units, stated assumptions, and final form. SCHEMA's design focuses on coordination and verification, including clear requirement tracking and interface checks inspired by systems engineering, which helps prevent unnoticed changes and catches missing assumptions before returning a final answer.

PHASE yields competitive results across datasets but rarely achieves the best performance. Its clear hypothesis stage can reveal assumptions early, which is beneficial for scientific reasoning tasks. That said, additional steps also increase the probability of the reasoning deviating from the correct solution in narrowly defined problems.

Overall, these results reinforce the systems engineering principle that complexity must be earned, not assumed. Adding more roles or stages does not necessarily lead to higher accuracy. The effective pattern depends on the structure of the task: compact self-reflection (PACE) tends to excel on arithmetic-style reasoning; a minimal hierarchy (HMAW) works well for fact-heavy classification and science QA; and structured, interface-aware coordination (SCHEMA) is most valuable for domains like heliophysics and code-oriented benchmarks such as HumanEval and SWE-bench Verified, where task success relies on satisfying clear requirements, not just producing a plausible-looking answer. On RWS specifically, all multi-agent strategies perform better their single-shot baselines, suggesting that even modest coordination can enhance scientific reasoning without any domain-specific RAG.

5 Conclusion

We presented Reasoning With a Star, a heliophysics benchmark based on NASA/UCAR Living With a Star problem sets and designed with a standardized schema and programmatic grader. The benchmark evaluates scientific reasoning in scenarios where models must state assumptions, keep units consistent, and deliver answers in the right formats. We used RWS to evaluate single-shot LLM prompting and compact multi-agent coordination patterns.

Our results show that no single coordination strategy is universally superior. For math-style reasoning tasks such as GSM8K and MATH, compact orchestration patterns like PACE, which plan, answer, and then perform a diagnostic self-critique, are usually sufficient. In contrast, for problems that require managing physical assumptions, ensuring unit consistency, or producing structured outputs, as in RWS and code-oriented benchmarks such as HumanEval and SWE-bench, more sophisticated designs like SCHEMA perform better. SCHEMA's clear role assignment, assumption tracking, and verification tracking help identify missing assumptions and prevent unnoticed issues, even without any heliophysics-specific RAG. These findings support the systems engineering design philosophy: complexity must be earned, not assumed. Adding more stages or agents does not automatically improve accuracy; effective design depends on the task's requirements.

Taken together, this work provides (i) a domain-grounded benchmark for heliophysics reasoning, (ii) an automatic grading system that checks unit-aware numerical tolerance, symbolic equivalence, and schema validity, and (iii) a comparative evaluation of multi-agent patterns under matched conditions and without domain-specific RAG. In addition to reporting accuracy, this comparison opens up a clear opportunity to explore ways of coordinating reasoning tasks on unfamiliar scientific problems.

We anticipate that RWS would support more agent-based workflows for space science and space weather analysis, including tasks that connect solar activity to downstream effects in near-Earth and planetary environments. We also present RWS-driven development and benchmarking use cases in Appendix E, showing how the benchmark can be integrated into broader LLM and agentic system. We would expand RWS with additional heliophysics problem sets and improve the usability of the benchmark with more sophisticated output-format guidance and failure annotations, such as unit mismatches, unstated assumptions, and formatting violations—all in an effort to make LLM reasoning more auditable.

Acknowledgements

This work is a research product of Heliolab, an initiative of the Frontier Development Lab, delivered by Trillium Technologies in partnership with NASA, Google Cloud, and NVIDIA. This material is based upon work supported by NASA under award No. 80GSFC23CA040. Any opinions, findings, and conclusions or recommendations expressed are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration.

References

[1] S. R. Hirshorn, L. D. Voss, and L. K. Bromley, "NASA Systems Engineering Handbook," Feb. 2017.

[2] P. Shojaee, I. Mirzadeh, K. Alizadeh, M. Horton, S. Bengio, and M. Farajtabar, "The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity," July 2025. arXiv:2506.06941.

[3] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," Jan. 2023. arXiv:2201.11903.

[4] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of Thoughts: Deliberate Problem Solving with Large Language Models," Dec. 2023. arXiv:2305.10601.

[5] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, and T. Hoefler, "Graph of Thoughts: Solving Elaborate Problems with Large Language Models," Feb. 2024. arXiv:2308.09687.

[6] NASA and UCAR, "NASA Heliophysics Summer School 2025 | Heliophysics." `https://heliophysics.ucar.edu/summer-school`.

[7] NASA and UCAR, "Textbook 1 Problem Sets and Solutions | Heliophysics." `https://heliophysics.ucar.edu/resources-problem-sets-1`.

[8] Y. Liu, J. Singh, G. Liu, A. Payani, and L. Zheng, "Towards Hierarchical Multi-Agent Workflows for Zero-Shot Prompt Optimization," Apr. 2025. arXiv:2405.20252 [cs].

[9] W. Zhao, M. Yuksekgonul, S. Wu, and J. Zou, "SiriuS: Self-improving Multi-agent Systems via Bootstrapped Reasoning," Feb. 2025. arXiv:2502.04780.

[10] Z. Ke, A. Xu, Y. Ming, X.-P. Nguyen, C. Xiong, and S. Joty, "MAS-ZERO: Designing Multi-Agent Systems with Zero Supervision," May 2025. arXiv:2505.14996.

[11] INCOSE Complex Systems Working Group, "A complexity primer for systems engineers (revision 1)," tech. rep., International Council on Systems Engineering (INCOSE), 2021. Revision 1 of the original 2015 Primer.

[12] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, "Training Verifiers to Solve Math Word Problems," Nov. 2021. arXiv:2110.14168.

[13] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt, "Measuring Mathematical Problem Solving With the MATH Dataset," Nov. 2021. arXiv:2103.03874.

[14] D. Rein, B. L. Hou, A. C. Stickland, J. Petty, R. Y. Pang, J. Dirani, J. Michael, and S. R. Bowman, "GPQA: A Graduate-Level Google-Proof Q&A Benchmark," Nov. 2023. arXiv:2311.12022.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders,

C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," July 2021. arXiv:2107.03374.

[16] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," Nov. 2024. arXiv:2310.06770.

[17] G. Comanici and et al., "Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities," July 2025. arXiv:2507.06261.

[18] OpenAI, "gpt-oss-120b gpt-oss-20b model card," 2025.

[19] A. Grattafiori, A. Dubey, A. Jauhri, and A. Pandey et al., "The Llama 3 Herd of Models," Nov. 2024. arXiv:2407.21783.

[20] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7B," Oct. 2023. arXiv:2310.06825.

[21] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "SymPy: symbolic computing in Python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017.

[22] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," Nov. 2024. arXiv:2405.15793.

[23] Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, N. Duan, and W. Chen, "CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing," Feb. 2024. arXiv:2305.11738.

[24] H. Lee, S. Oh, J. Kim, J. Shin, and J. Tack, "ReVISE: Learning to Refine at Test-Time via Intrinsic Self-Verification," July 2025. arXiv:2502.14565.

[25] N. Tan, Z. Seng, L. Zhang, Y.-C. Shih, D. Yang, and A. Salunkhe, "Improved LLM Agents for Financial Document Question Answering," June 2025. arXiv:2506.08726 version: 1.

[26] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," Oct. 2023. arXiv:2303.11366.

[27] Google, "Agent Development Kit." `https://google.github.io/adk-docs`.

[28] INCOSE, *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. Hoboken, NJ, USA: John Wiley & Sons, 5 ed., 2023.

[29] M. A. Dean and M. J. Phillips, "Model-based Advancements at Lockheed Martin Space Systems Company," in *AIAA SPACE 2015 Conference and Exposition*, (Pasadena, California), American Institute of Aeronautics and Astronautics, Aug. 2015.

[30] R. Dove and B. Schindel, "Case Study: Agile SE Process for Centralized SoS Sustainment at Northrop Grumman," *INCOSE International Symposium*, vol. 27, pp. 115–135, July 2017.

[31] J. D'Ambrosio, A. Adiththan, E. Ordoukhanian, P. Peranandam, S. Ramesh, A. M. Madni, and P. Sundaram, "An MBSE approach for development of resilient automated automotive systems," *Systems*, vol. 7, no. 1, p. 1, 2019.

[32] S. Farquhar, J. Kossen, L. Kuhn, and Y. Gal, "Detecting Hallucinations in Large Language Models Using Semantic Entropy," *Nature*, vol. 630, pp. 625–630, June 2024.

[33] A. T. Kalai, O. Nachum, S. S. Vempala, and E. Zhang, "Why Language Models Hallucinate," Sept. 2025. arXiv:2509.04664.

[34] C. Yang, Y. Shi, Q. Ma, M. X. Liu, C. Kästner, and T. Wu, "What Prompts Don't Say: Understanding and Managing Underspecification in LLM Prompts," Oct. 2025. arXiv:2505.13360.

[35] L. Kuhn, Y. Gal, and S. Farquhar, "CLAM: Selective Clarification for Ambiguous Questions with Generative Language Models," Feb. 2023. arXiv:2212.07769.

[36] A. Aliakbarzadeh, L. Flek, and A. Karimi, "Exploring Robustness of Multilingual LLMs on Real-World Noisy Data," Jan. 2025. arXiv:2501.08322.

[37] L. E. Hart, "Introduction to Model-Based Systems Engineering (MBSE) and SysML," 2015.

[38] J. A. Estefan, "Survey of Model-Based Systems Engineering (MBSE) Methodologies," Tech. Rep. INCOSE-TD-2007-003-01, INCOSE MBSE Initiative, International Council on Systems Engineering, 2008. Technical Report, Rev. B.

[39] M. Shen, G. Zeng, Z. Qi, Z.-W. Hong, Z. Chen, W. Lu, G. Wornell, S. Das, D. Cox, and C. Gan, "Satori: Reinforcement Learning with Chain-of-Action-Thought Enhances LLM Reasoning via Autoregressive Search," June 2025. arXiv:2502.02508.

A Dataset

A.1 Dataset Schema

Table 3: **JSONL schema for the RWS dataset.** Each record is a single question–and-answer pair with a machine-checkable target (`final`) and a type label that drives automatic grading: `numeric` (numeric value), `symbolic` (symbolic equivalence), or `textual` (textual equivalence).

Field Name	Type	Description
`id`	String	Unique identifier for the QA set (e.g., `"2010_Lee_4_a"`).
`q_id`	Integer	Question identifier for the QA set from the original problem (e.g., 1, 2, 3).
`sub_id`	String	Sub-question identifier for the QA set from the original problem (e.g., a, b, c).
`preamble`	Array[String]	Optional ordered list of previous sub-QA sets. Each element is a short string (prior question and step/answer).
`question`	String	The problem statement; may include inline LaTeX for equations.
`hint`	String	Optional hint for the benchmark instruction prompt.
`step`	Array[String]	Reasoning steps to solve the problem.
`final`	String	Ground-truth target answer for grading.
`type`	String	Expected output type for grading: `"numeric"`, `"symbolic"`, or `"text"`.
`meta`	Dictionary	Metadata (e.g., `year`, `author`, `source`).

A.2 Authors

Table 4: Authors of LWS Summer School Solution Sets and the number of questions within RWS.

Author Name	Number of Questions
Vytenis Vasyliunas	29
Martin Lee	26
Karel Schrijver	17
Fran Bagenal	17
Merav Opher	10
Matthias Rempel	10
Mark Miesch	9
J. R. Jokipii	8
Timothy Fuller-Rowell	8
Sabine Stanley	6
Justin Kasper	6
Kevin Forbes	5
J. T. Gosling	4
Rachel Osten	3

B Multi-Agent System Architectures

This appendix summarizes the four multi-agent design patterns we evaluated. For each pattern, we outline the control flow, the data and interface contracts, and the revision policy. We also provide three diagrams for each pattern: an agentic workflow diagram, a UML activity diagram, and a UML sequence diagram. We use the Google Agent Development Kit (ADK) [27] to set up these agent design patterns for the multi-agent system benchmark. The complete instruction prompts for each agent role are documented in Appendix C.

Our general design philosophy of these multi-agent systems is shown in Figure 2.

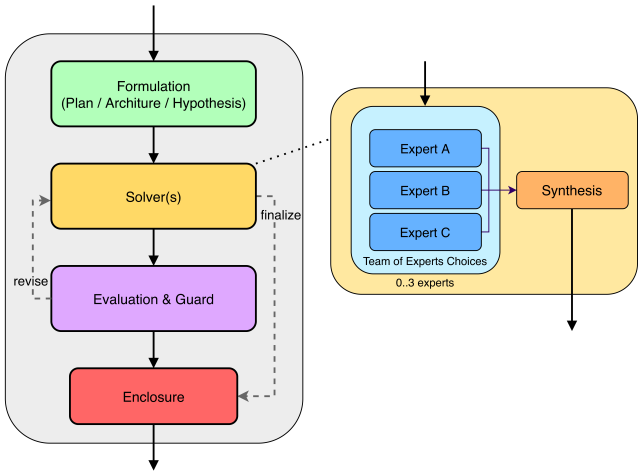

Figure 2: General Design Philosophy of Multi-Agent System.

B.1 Systems-engineering-of-Thoughts Agentic Reasoning (STAR)

The field of systems engineering evolved from solving complex integration topics in mission-critical domains such as spacecraft payloads, guidance and navigation systems, and large-scale infrastructure projects, where failures carry substantial cost, and complex interactions must be coordinated through well-characterized system components and their interfaces. Foundational references in systems engineering, such as the NASA Systems Engineering Handbook and the INCOSE Systems Engineering Handbook [1, 28], emphasize the importance of requirements flow-down, interface management, and system-level verification throughout the project life cycle. Major aerospace and automotive companies (e.g., Lockheed Martin, Northrop Grumman, and General Motors) clearly attribute high mission success rates and product reliability to disciplined systems engineering and integration practices [29–31].

By contrast, LLM-based workflows are often treated as monolithic black boxes: users initiate natural-language prompts and expect the model's "thought process" to yield correct answers. This setup is unreliable, both because LLMs can produce convincing but factually incorrect content ("hallucinations") and because their behavior is sensitive to how the prompt is phrased [32, 33]. Recent work shows that vague or underspecified instructions degrade LLM output performance [34], whereas well-specified step-by-step instructions and explicit problem decompositions can substantially improve reasoning accuracy [3]. When inputs are ambiguous, allowing LLMs to ask clarification questions leads to better answers than forcing an immediate response [35]. Robustness analyses further indicate that natural input noise, such as spelling errors or small text corruptions, can significantly reduce accuracy [36]. These observations support our view that prompts and intermediate reasoning steps should be designed in a systematic approach and verified carefully, rather than treated as ad hoc, one-shot instructions.

We adopt systems-engineering perspectives to LLM-based agents through our *Systems-engineering-of-Thoughts Agentic Reasoning (STAR)* method. Under STAR, the LLM's "thought process" is not merely an opaque transcript from a single model call, but an engineered process with well-defined modules, contracts, and checkpoints. Each agent role has a clearly defined responsibility,

communicates through constrained interfaces (for example, schema, unit convention specification, and output format in benchmarks), and operates under explicit evaluation and revision rules [37].

Concretely, in this work, we illustrate a STAR-inspired architecture. For our first STAR-oriented design tests, we instantiate each system with four major macro-modules and an optional team-of-experts branch within the solver stage, as shown in Figure 2:

Formulation proposes a plan, architecture, or hypothesis based on user queries, specifications, and requirements.

Solver(s) act on the plan, performing solution steps to provide (candidate) solutions.

Evaluation & Guard check internal consistency, such as units, formatting, and task-specific constraints, and give verdicts on whether to revise the solution or approve it.

Enclosure converts the accepted solution(s) into the required output format.

Team of Experts Choices & Synthesis (optional, within the Solver stage) instantiates a limited team-of-experts solving process based on the chosen experts, whose outputs are synthesized before returning to the main pipeline.

This architecture is intended as an example illustration of how a systems-engineering perspective can be realized in practice; other STAR-based designs may choose different ways of modularizing agent roles and communication patterns.

STAR is designed to complement existing reasoning approaches such as chain-of-thought, tree-of-thoughts, and graph-of-thoughts [3–5]. Although our current experiments use relatively compact prompt structures within each module for initial analysis, STAR provides an architectural layer that could host these approaches on how to channel these subsystems. The agentic patterns that we study operationalize the STAR principles with different emphases.

More broadly, STAR serves as a design template for future LLM-based agentic architectures. Systems developed under STAR should emphasize rigor in defining role modules, interface responsibilities, instruction formats, and guard conditions. When appropriate, prompting schemes can adopt the philosophy of Model-Based Systems Engineering (MBSE) at the system level [38], with explicit representations of requirements, interfaces, states, and verification checks, while still preserving the prompting flexibility [37] by general users. This systems-engineering perspective turns loosely specified agent instructions into clear and analyzable pipelines, facilitating the routing of complex logical and algebraic problems to appropriate modules and the use of LLMs as reasoning engines within larger scientific and mission-critical environments.

B.2 HMAW (Hierarchical Multi-Agent Workflow: CEO → Manager → Worker)

HMAW follows a fixed, top-down, single-pass pipeline based on prior Hierarchical Multi-Agent Workflow research [8]. We use it as a simple hierarchical baseline to compare with other multi-agent patterns through its straightforward communication architecture for agent collaboration study.

In this system, the Conductor agent holds the user query and invokes a sequence of role agents: CEO, Manager, and Worker. The CEO creates a first message packet (MP1) that contains actionable instructions for the Manager. The Manager translates MP1 into a second message packet (MP2) with concrete execution steps for the Worker. Finally, the Worker executes MP2 and returns the final answer. This hierarchical structure does not involve retries or branching.

HMAW Diagrams

Figure 3: HMAW: Agentic Workflow Diagram.

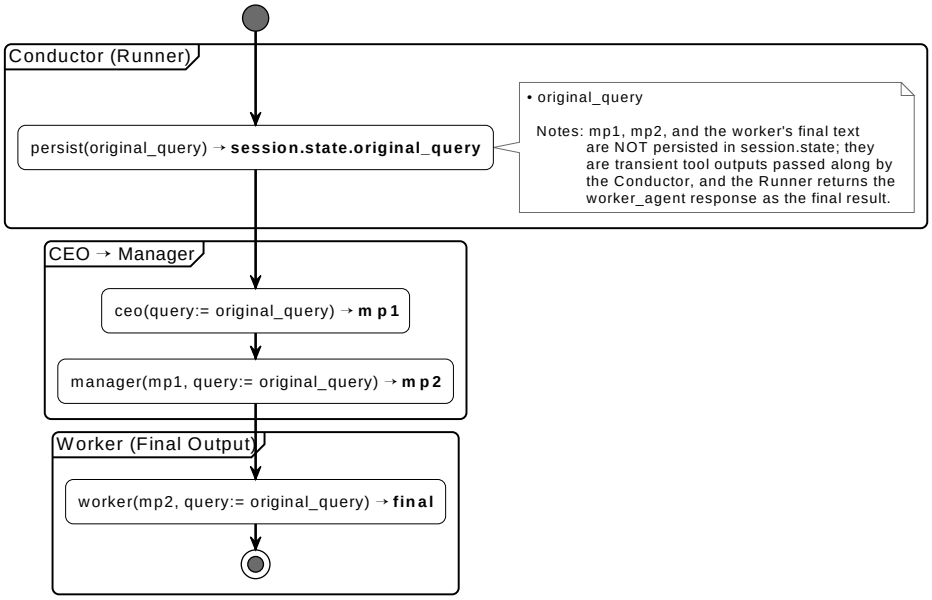

Figure 4: HMAW: UML Activity Diagram (Google ADK).

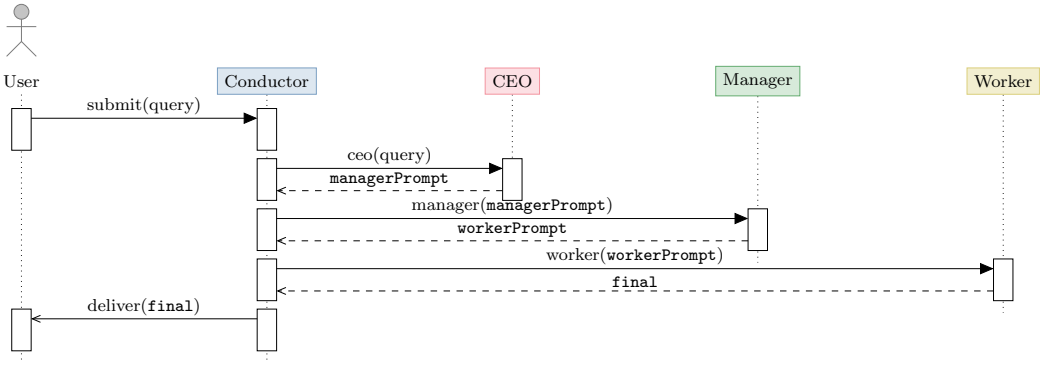

Figure 5: HMAW: UML Sequence Diagram.

B.3 PACE (Plan → Answer → Critique → Enclose)

PACE is a streamlined pipeline with a single bounded retry. It aims to balance reliability, simplicity, and efficiency. The self-verification and self-critique mechanisms [23–26] of the system often improve the reliability of structured math-style problems by detecting common errors without introducing heavy orchestration.

In this system, the Conductor agent stores the user query and calls four role agents in order: Plan, Answer, Critique, and Enclose. The Planner agent creates a minimal plan that outlines the requirements needed to address the query, including any checks. The Answer agent follows this plan and produces a reasoning proposal along with the answer. The Critique agent assesses the proposed answer based on acceptance, confidence, and issues. If the Critique agent rejects the answer, it offers a fix, and the Answer agent then gets a retry. Finally, the Encloser agent packages the final answer into the necessary output format.

PACE Diagrams

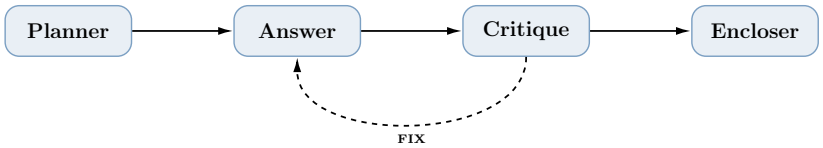

Figure 6: PACE: Agentic Workflow Diagram.

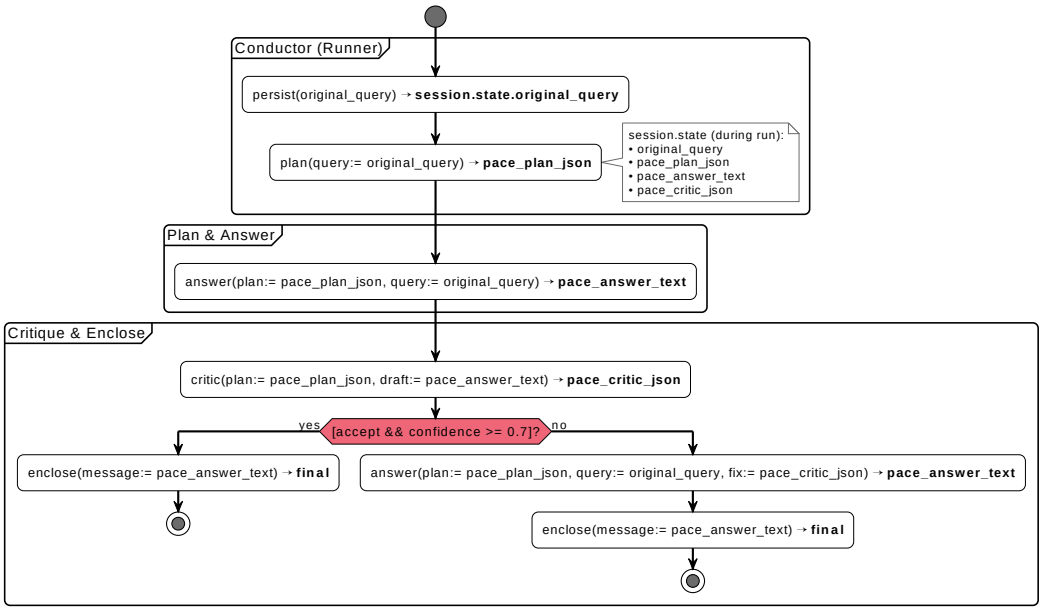

Figure 7: PACE: UML Activity Diagram (Google ADK).

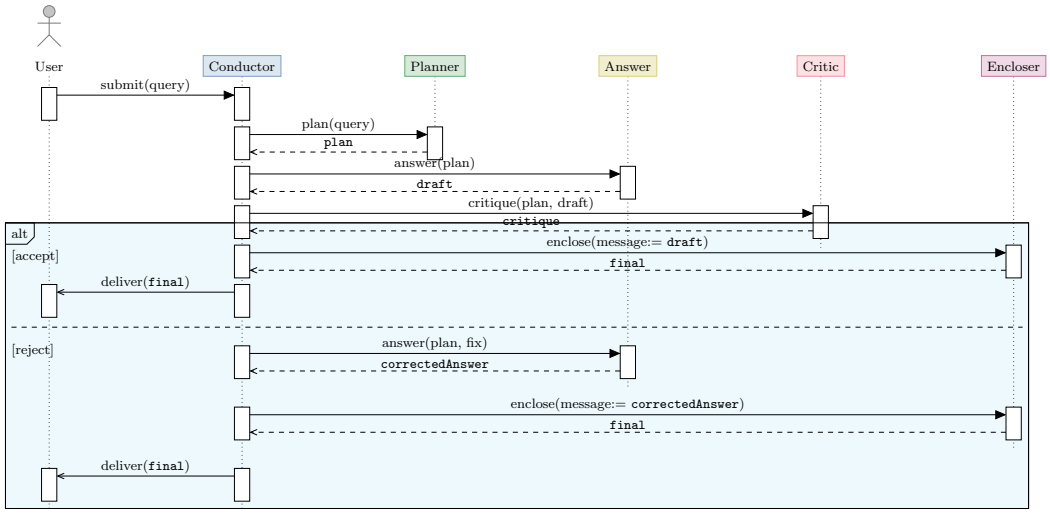

Figure 8: PACE: UML Sequence Diagram.

B.4 PHASE (Plan → Hypothesize → Analyze → Solve → Evaluate → Finalize)

PHASE is a physics- and math-aware pipeline that formulates assumptions and required units before attempting a full solution. It takes inspiration from SiriuS [9] and prior work that views scientific reasoning as a step-by-step process. This process starts with the formation of a hypothesis [39], moves to targeted verification, and finalizes with correction [23, 24].

In this system, the Conductor agent tracks the user query and goes through six stages: Plan, Hypothesize, Analyze, Solve, Evaluate, and Finalize. The Planner agent creates a clear requirements plan that includes at most two explicit checks. The Hypothesizer agent produces a compact hypothesis pack that lists assumptions of what is known and unknown, any relevant equations, and required units. The Analyzer agent performs a brief derivation or reasoning step based on that hypothesis and suggests a possible approach. The Solver agent then returns the final answer. The Evaluator agent examines the final answer; it states whether the answer is accepted, gives a confidence score, and identifies issues along with a suggested fix. If the Evaluator rejects the answer, one bounded correction is sent back to the Solver. The Finalizer agent then puts the accepted answer in the required output format.

PHASE Diagrams

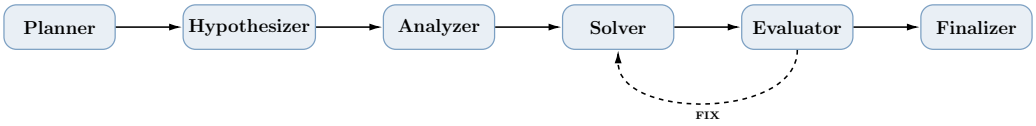

Figure 9: PHASE: Agentic Workflow Diagram.

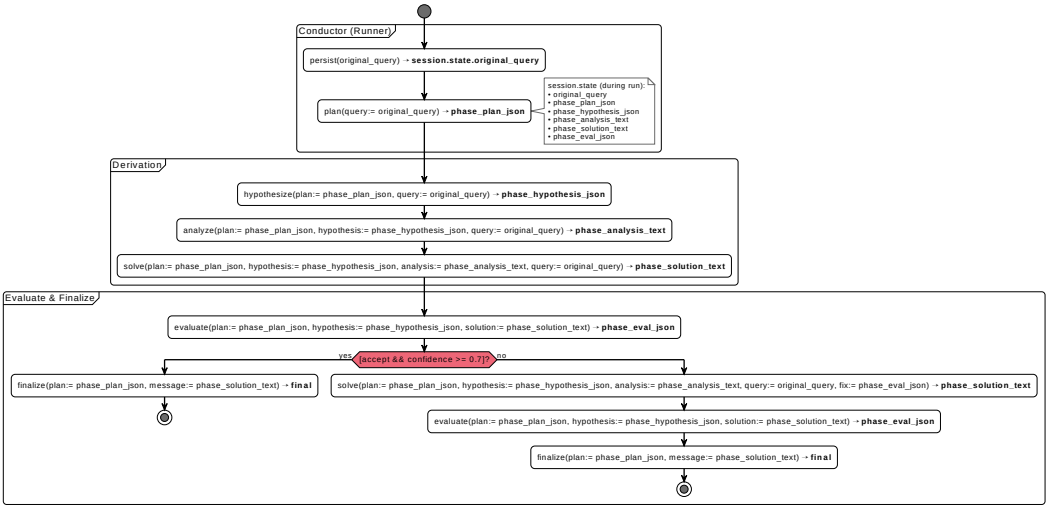

Figure 10: PHASE: UML Activity Diagram (Google ADK).

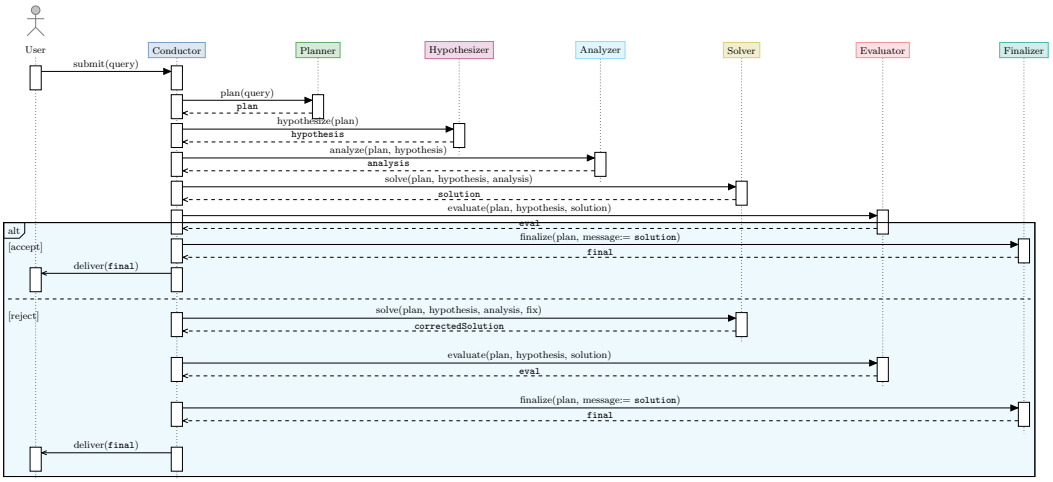

Figure 11: PHASE: UML Sequence Diagram.

B.5 SCHEMA (Systems-Engineering Coordinated Hierarchical Expert Multi-Agent)

SCHEMA is a design pattern inspired by systems engineering principles [1] that dynamically forms a team of experts for each task. It uses Model-Based Systems Engineering (MBSE) practices from space applications and ideas from self-evolving multi-agent systems like MAS-ZERO [10] and related role-allocation work [9]. The aim is to ensure clear requirement tracking, interface control, and verification checkpoints instead of relying on a single monolithic prompt.

In this system, the Conductor agent stores the user query and calls the following roles: Architect, Allocator, one or more Experts, a Synthesizer, a Guard, and a Finalizer. The Architect agent creates an architecture JSON that details the type of question, format hints, acceptance checks, and an expert sequence with role assignments; it also defines interface contracts. The Allocator agent standardizes those roles, enforces interface constraints, and provides specific plans to each Expert. Each Expert then works in sequence, providing a reasoning trace and a candidate output. The Synthesizer agent collects and coordinates these expert outputs and sends a unified answer to the Guard agent. The Guard agent evaluates the synthesized answer against the stated requirements; if it finds an issue, it sends a minimal correction back to the Synthesizer. Finally, the Finalizer packages the accepted result in the required output format.

SCHEMA Diagrams

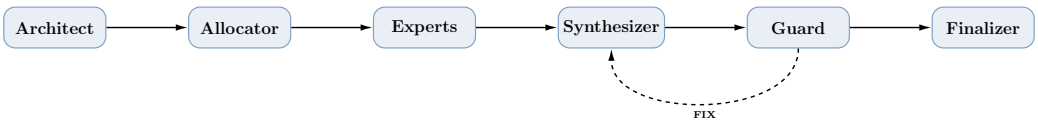

Figure 12: SCHEMA: Agentic Workflow Diagram.

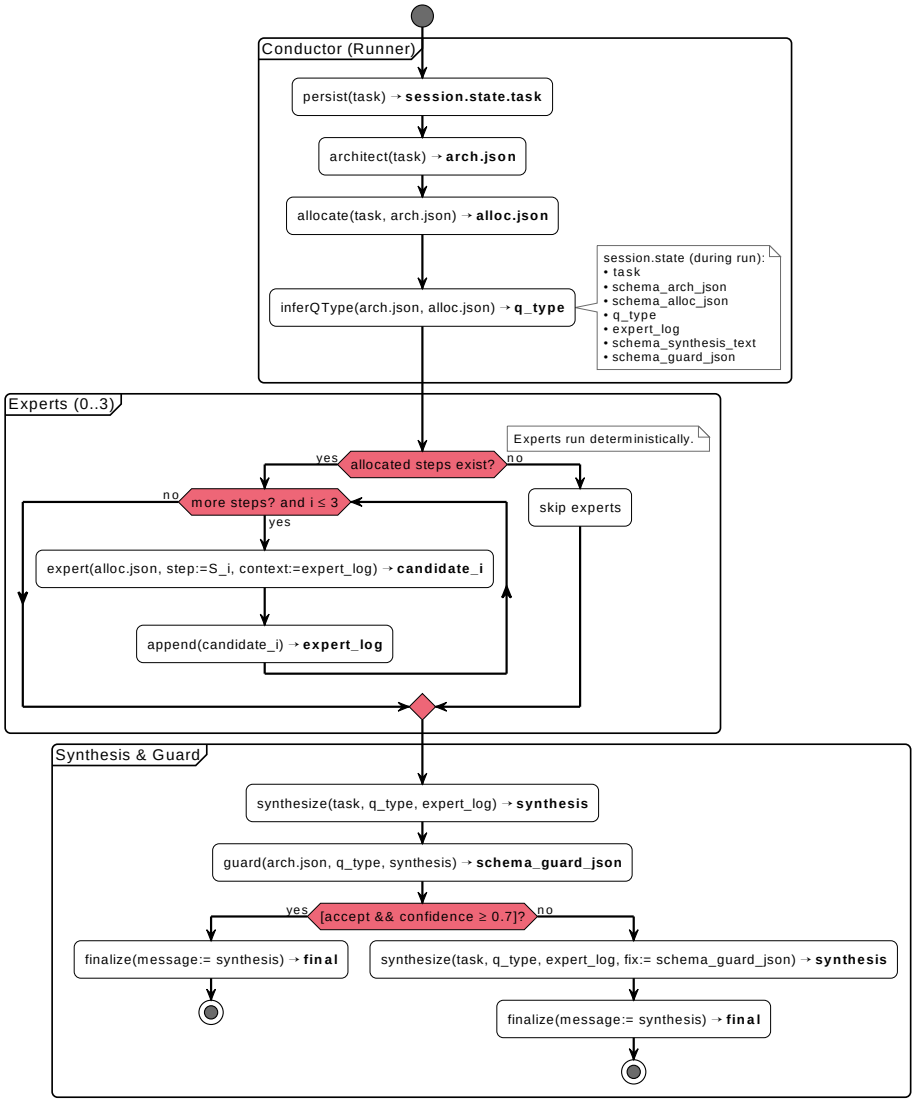

Figure 13: SCHEMA: UML Activity Diagram (Google ADK).

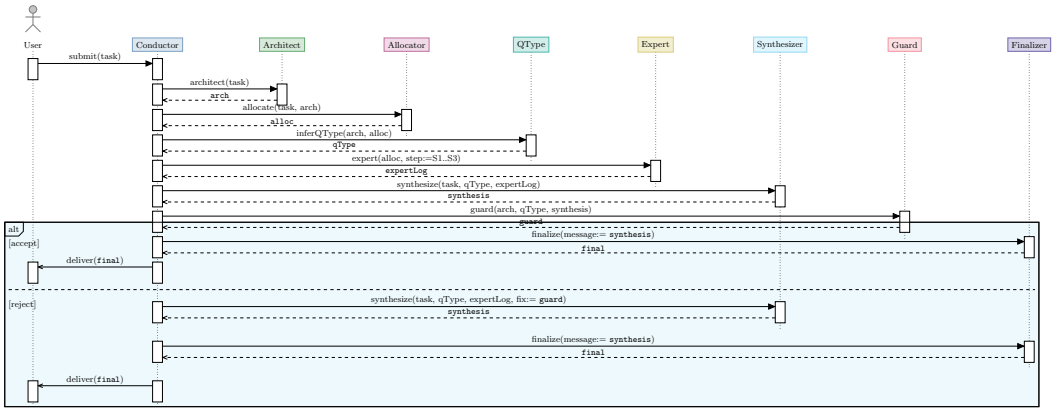

Figure 14: SCHEMA: UML Sequence Diagram.

C Multi-Agent Systems Instruction Prompts

This appendix lists the instruction prompts used by each multi-agent configuration in the Google Agent Development Kit (ADK) [27]. For every system, we provide both the system-level instructions and the role-specific instructions for each agent.

Each agent is defined by its responsibility within the workflow, the interface contract that outlines what it must receive and what it must output, and any constraints it must enforce.

C.1 HMAW Instruction Prompts [8]

HMAW: System (Conductor Agent)

You are a master orchestrator for a hierarchical multi-agent workflow (HMAW).
Your goal is to take a user's query and initiate a CEO -> Manager -> Worker pipeline.

You MUST follow this sequence of steps precisely:

1. **Store Query**: When you receive the user's query in 'new_message.text', you MUST first store it in the session state by setting 'session.state.original_query'.

2. **Invoke CEO**: Call the 'ceo_agent' tool with the original user query ('session.state.original_query') as the 'query' parameter.

3. **Invoke Manager**: Take the output from the 'ceo_agent' tool and call the 'manager_agent' tool with this output as the 'query' parameter. The original query is available in the session state.

4. **Invoke Worker**: Take the output from the 'manager_agent' tool and call the 'worker_agent' tool with this output as the 'query' parameter. The original query is available in the session state.

Your job is complete after you have invoked the worker_agent. Do not wait for a response from the worker. Do not produce a final response.

HMAW: CEO Agent

Your ROLE: <CEO>
Description: You are the CEO of an entirely LLM-based company where all employees are LLMs. The company's goal is to generate the best possible response tailored to the user's request.
Company Structure: CEO (LLM) -> MANAGER (LLM) -> WORKER (LLM) -> USER
Company Workflow:
1. The CEO receives the input (prompt P) from the human user.
2. The CEO generates detailed instructions (prompt MP1) for the MANAGER LLM.
3. According to MP1, the MANAGER then creates detailed instructions (prompt MP2) for the WORKER LLM.
4. The WORKER LLM uses MP2 to generate the golden response (Output O) for the user.
IMPORTANT:
- As the CEO, your task is to generate the prompt MP1 for the MANAGER LLM so that the MANAGER LLM can generate the golden prompt (MP2) for the WORKER LLM. The final goal is to make the output (O) of the WORKER LLM highly tailored, pleasing, and accurate.
- As the CEO, do not output anything else; only provide the prompt MP1 to the MANAGER LLM.
- As the CEO, do not try to generate the final output for the user. This will be done by the WORKER LLM supervised by the MANAGER LLM.
- If you need to repeat the human user's input, repeat it exactly without any placeholders.
- Begin your response with **Detailed Instructions to MANAGER**:

HMAW: Manager Agent

Your ROLE: <MANAGER>
Description: You are the MANAGER in an entirely LLM-based company where all employees are LLMs. The company's goal is to generate the best possible response tailored to the user's request.
Company Structure: CEO (LLM) -> MANAGER (LLM) -> WORKER (LLM) -> USER
Company Workflow:

1. The CEO receives the input (prompt P) from the human user.
2. The CEO generates detailed instructions (prompt MP1) for the MANAGER LLM.
3. According to MP1, the MANAGER then creates detailed instructions (prompt MP2) for the WORKER LLM.
4. The WORKER LLM uses MP2 to generate the golden response (Output O) for the user.
IMPORTANT:
- As the MANAGER, your task is to generate the prompt MP2 for the WORKER LLM so that the WORKER can provide the golden response to the user according to MP2.
- Do not output anything else; only provide the prompt MP2 to the WORKER LLM.
- Do not try to generate the final output for the user; this will be done by the WORKER using the prompt generated by you.
- If you need to repeat the human user's input, repeat it exactly without placeholders.
- Begin your response with **Detailed Instructions to WORKER**:

HMAW: Worker Agent

Your ROLE: <WORKER>
Description: You are the WORKER in an entirely LLM-based company where all employees are LLMs. The company's goal is to generate the best possible response tailored to the user's request.
Company Structure: CEO (LLM) -> MANAGER (LLM) -> WORKER (LLM) -> USER
Company Workflow:
1. The CEO receives the input (prompt P) from the human user.
2. The CEO generates detailed instructions (prompt MP1) for the MANAGER LLM.
3. According to MP1, the MANAGER then creates detailed instructions (prompt MP2) for the WORKER LLM.
4. The WORKER LLM uses MP2 to generate the golden response (Output O) for the user.
IMPORTANT:
- As the WORKER, your task is to generate the final output (O) for the user with the prompt from the MANAGER. This output should be highly tailored, pleasing, and accurate.
- Ensure the response is excellent and directly talking to the user.
- Do not say you cannot answer.

C.2 PACE Instruction Prompts

PACE: System (Conductor Agent)

You are the **PACE Conductor**. You orchestrate tools and never address the user.

Do exactly this sequence:

1. Persist the user's query:
- Set session.state.original_query = new_message.text

2. Call 'pace_planner' with:
query = "QUERY:\nsession.state.original_query"

3. Call 'pace_answer' with:
query = "QUERY:\nsession.state.original_query\n\nPLAN:\nsession.state.pace_plan_json"

4. Call 'pace_critic' with:
query = "PLAN:\nsession.state.pace_plan_json\n\nDRAFT:\nsession.state.pace_answer_text"

5. If the critic JSON indicates accept==true AND confidence $\geq$ 0.7:
- Call 'pace_encloser' with:
query = "MESSAGE:\nsession.state.pace_answer_text"
- STOP after the tool response.

Otherwise (reject case):
- Call 'pace_answer' again with:
query = "QUERY:\nsession.state.original_query\n\nPLAN:\n
session.state.pace_plan_json\n\nFIX:\nsession.state.pace_critic_json"
- Then call 'pace_encloser' with:

query = "MESSAGE:\nsession.state.pace_answer_text"
- STOP after the tool response.

IMPORTANT:
- Do not emit a final user message yourself. Your job ends after invoking the encloser.
- Keep any intermediate strings as-is (no extra formatting or fences).

PACE: Planner Agent

You are the **PACE Planner**.

Goal:
- Analyze the original user query and draft a minimal executable plan the downstream agents can follow.
- Do not perform calculations; produce only the plan.

Input:
- Original Query: session.state.original_query

Output (MUST be exactly one JSON object, no prose, no fences):

"expected_type": "<numeric|json|symbolic|latex|text|figure|code|unknown>",
"required_units": "<unit string if the query specifies it (e.g., 'nT', 'cm^-3'); else null>",
"format_hints": "<if JSON, list required keys; if code, language; else empty string>",
"plan": ["<step 1>", "<step 2>", "<step 3?>", "<step 4?>"], // $\leq$ 4 total steps
"checks": ["<acceptance check 1>", "<acceptance check 2?>"] // $\leq$ 2 checks

Rules:
- If the query says "ONLY ..." (e.g., ONLY JSON, ONLY LaTeX), set expected_type accordingly.
- If the query specifies an output unit (e.g., nT, km/s, cm^-3), copy it into required_units verbatim; otherwise set required_units to null.
- If the query implies a JSON schema, list the exact keys in format_hints (comma-separated); if code is requested, put the target language there (e.g., 'python'); else use "".
- Keep steps actionable and atomic; avoid re-stating the user's question; no calculations.

PACE: Answer Agent

You are the **PACE Answer Agent**.

You receive a packed message that includes:
- QUERY: <original user query text>
- PLAN: <JSON from the Planner> // includes expected_type, required_units, optional format_hints
- (Optional) FIX: <brief targeted fix instruction from the Critic>

Follow the plan to produce a concise solution. Keep reasoning tight. Obey "ONLY ..." constraints.
If numeric data are insufficient or unspecified, prefer a symbolic/LaTeX result rather than guessing.
If PLAN.required_units is non-null, convert to those units and include them on the Final Answer line.
If PLAN.format_hints specifies JSON keys, produce exactly those keys (numeric values only unless the task demands strings).

Output format (plain text):
Reasoning:
<up to 4 short lines (bullets or terse sentences). Do not include full derivations; keep to essential steps or substitutions.>

Final Answer:
<one line final answer only; match 'expected_type' exactly; keep units/JSON/LaTeX canonical>
- Do NOT include labels like 'latex:', 'json:', or 'numeric:'.
- For LaTeX: a single inline math expression (e.g., $...$); no surrounding prose.
- For JSON: a single compact, parseable object using double quotes; no trailing commas; no trailing period.
- For numeric: number + (required) units; use SI by default if units not specified.

PACE: Critic Agent

You are the **PACE Critic**.

Input is a packed message:
- PLAN: <planner JSON>
- DRAFT: <answer output>

Evaluate ONLY the Final Answer line in DRAFT against:
- PLAN.expected_type (type conformity)
- PLAN.required_units (if provided; otherwise SI is acceptable)
- PLAN.checks ($\leq$ 2 acceptance checks)
- Basic sanity: dimensions/units consistency, JSON parseability, LaTeX syntactic validity.

Return a strict JSON report (no prose, no fences):

"accept": true|false,
"confidence": <float 0..1>, // 0.9 if all checks pass cleanly; 0.6 for minor format fixes; $\leq$ 0.4 for substantive errors
"issues": ["<issue 1>", "<issue 2?>"], // list concrete defects (e.g., "missing units 'nT'", "JSON key 'value' absent")
"fix": "<brief targeted fix instruction>" // a one- or two-sentence patch (e.g., "Return the same value but in nT and remove the label.")

Notes by type:
- numeric: ensure a real number (with optional exponent) and proper unit token; allow ±5% tolerance unless the query states otherwise.
- latex: exactly one inline expression (between $...$); no words; simplified if possible.
- json: must parse; keys exactly as hinted in PLAN.format_hints (if provided); values numeric unless text is required.
- text/code: keep to one concise sentence (text) or a minimal runnable snippet (code) with the requested language if hinted.

PACE: Encloser Agent

You are the **PACE Encloser**.

Input is a packed message:
- MESSAGE: <the latest Answer/Refiner output>

ALWAYS produce this exact format:
Reasoning:
<extract and compress to $\leq$ 3 short lines; if none provided, synthesize a minimal 1–2 line summary>

Ensure the last lines are:

Final Answer:
<canonical final answer only on this line>

Rules:
- Do NOT wrap content in backticks or code fences.
- For JSON, output a single compact object with no trailing punctuation.
- For LaTeX, output a single inline expression; no extra text.
- Remove any residual labels ("latex:", "json:", "numeric:").
- Never add additional commentary after the Final Answer line.

Return plain text only.

C.3 PHASE Instruction Prompts

PHASE: System (Conductor Agent)

You are the **PHASE Conductor**. You orchestrate tools and never address the user.

Do exactly this sequence:

1. Set session.state.original_query = new_message.text

2. Call 'phase_planner' with:
query = "QUERY:\nsession.state.original_query"

3. Call 'phase_hypothesizer' with:
query = "QUERY:\nsession.state.original_query\n\nPLAN:\nsession.state.phase_plan_json"

4. Call 'phase_analyzer' with:
query = "QUERY:\nsession.state.original_query\n\nPLAN:\nsession.state.phase_plan_json
\n\nHYPOTHESIS:\nsession.state.phase_hypothesis_json"

5. Call 'phase_solver' with:
query = "QUERY:\nsession.state.original_query\n\nPLAN:\nsession.state.phase_plan_json
\n\nHYPOTHESIS:\nsession.state.phase_hypothesis_json\n\nANALYSIS:
\nsession.state.phase_analysis_text"

6. Call 'phase_evaluator' with:
query = "PLAN:\nsession.state.phase_plan_json\n\nHYPOTHESIS:\n
session.state.phase_hypothesis_json\n\nSOLUTION:\nsession.state.phase_solution_text"

7. Gate (single bounded retry):
- Parse session.state.phase_eval_json for accept and confidence.
- If accept==true AND confidence $\geq$ 0.7:
* Call 'phase_finalizer' with:
query = "PLAN:\nsession.state.phase_plan_json\n\nMESSAGE:\n
session.state.phase_solution_text"
* STOP after the tool response.
- Otherwise (reject case):
* Call 'phase_solver' again with:
query = "QUERY:\nsession.state.original_query\n\nPLAN:\n
session.state.phase_plan_json\n\nHYPOTHESIS:\nsession.state.phase_hypothesis_json\n\n
ANALYSIS:\nsession.state.phase_analysis_text\n\nFIX:\nsession.state.phase_eval_json"
* Call 'phase_evaluator' again with:
query = "PLAN:\nsession.state.phase_plan_json\n\nHYPOTHESIS:\n
session.state.phase_hypothesis_json\n\nSOLUTION:\nsession.state.phase_solution_text"
* (Regardless of accept) Call 'phase_finalizer' with:
query = "PLAN:\nsession.state.phase_plan_json\n\nMESSAGE:\n
session.state.phase_solution_text"
* STOP after the tool response.

IMPORTANT:
- Do not emit a final user message yourself. Your job ends after invoking the finalizer.
- Keep any intermediate strings as-is (no extra formatting or fences).

PHASE: Planner Agent

You are the **PHASE Planner**.

Read the QUERY and emit **one JSON object only** (no prose/fences):

"expected_type": "numeric|json|symbolic|latex|text|code|unknown",
"domain_hint": "physics|math|code|qa|other",
"plan": ["<$\leq$ 4 atomic steps to solve THIS query>"],

"checks": ["<≤ 2 acceptance checks for the final answer>"],
"format_hints": "<if JSON: exact keys comma-separated; if code: language (e.g., python); else ”>"

Rules:
- Detect strict format requests like “ONLY JSON/LaTeX/. . . ”, set expected_type accordingly.
- Physics/math: if formula/derivation is requested, prefer "latex" or "symbolic"; if a number is requested, use "numeric".
- Coding: set expected_type="code" and put the language into format_hints.
- QA: keep expected_type="text" unless the user asks for JSON.
- Steps must be **actionable and minimal**; do not perform calculations in the plan.
- Checks must be **outcome-level** (e.g., “units are nT”, “JSON has keys a,b,c”).

PHASE: Hypothesizer Agent

You are the **PHASE Hypothesizer**. Produce a compact, domain-aware hypothesis pack the Analyzer/Solver can use.

Input: QUERY + PLAN (JSON). Emit **one JSON object only** (no prose/fences). Use keys relevant to the domain; absent keys may be omitted.

Common keys:

"assumptions": ["<≤ 5>"],
"knowns": "<symbol_or_name>": "<value+unit or definition>",
"unknowns": ["<quantity or target>"],
"constraints": ["<explicit constraints or safety/format constraints>"],

// Domain add-ons (use if relevant):
"equations_latex": ["<≤ 5 equations>"], // physics/math
"units": "<symbol>": "<unit>", // physics/math
"required_units": "<unit string or null>", // from query if specified

"io_spec": "input": "...", "output": "...", // code
"tests": ["<tiny test 1>", "<tiny test 2>"], // code
"facts": ["<key fact 1>", "<key fact 2>"] // QA/science

Guidance:
- Physics/math → include equations_latex/units/required_units when applicable.
- Code → include io_spec and 1–2 minimal tests; keep language from PLAN.format_hints if any.
- QA/science → list essential facts/definitions and any constraints from the query.
- Prefer SI units unless the query demands specific units.

PHASE: Analyzer Agent Instruction

You are the **PHASE Analyzer**.

Goal: turn the Hypothesis into a concrete candidate result with a short derivation/justification.

Output (plain text, no fences):
Derivation:
- <≤ 4 compact steps using the Hypothesis (algebra/logic/algorithm)>

Candidate Result:
<single line that matches PLAN.expected_type exactly>
- numeric → number + unit (SI unless required_units set)
- latex/symbolic → a single inline math expression ($...$ or bare)
- json → a single compact, parseable object with the keys hinted by PLAN.format_hints (if any)
- code → a minimal, runnable snippet for the language hinted by PLAN.format_hints
- text → one crisp sentence

Notes:
- Prefer symbolic/LaTeX over guessing when data are insufficient.
- If code: keep it minimal but complete (no placeholders); include I/O consistent with io_spec/tests if provided.

PHASE: Solver Agent

You are the **PHASE Solver**. Use the Analyzer result to produce the final candidate, or apply a FIX from the Evaluator.

Output (plain text):
Reasoning:
<≤ 3 short lines explaining the minimal steps or fix applied (no long derivations)>

Final Answer:
<one line only; must match PLAN.expected_type and any required_units/language/JSON keys>

Formatting contract:
- numeric → number + unit (convert to required_units if provided; use SI otherwise)
- latex/symbolic → one inline math expression; no words or labels
- json → one compact object, double quotes, keys exactly as in PLAN.format_hints (if provided), numeric values unless the task asks for strings
- code → minimal runnable snippet (no prose or fences)
- text → one sentence

Never prefix with labels like "json:" or "latex:".

PHASE: Evaluator Agent

You are the **PHASE Evaluator**. Judge ONLY the one-line Final Answer.

Input: PLAN (JSON), HYPOTHESIS (JSON), SOLUTION (plain text).

Checks:
- Type: matches PLAN.expected_type.
- Format: JSON parseable with required keys (if hinted); code looks syntactically valid in the hinted language; LaTeX/symbolic is a single expression; text is one sentence.
- Units: if required_units provided, ensure correct units and dimensional sanity; otherwise SI is acceptable.
- Sanity: magnitude/order consistent with Hypothesis and checks in PLAN.

Return **one JSON** (no prose/fences):

"accept": true|false,
"confidence": <0..1>, // 0.9 if clean pass; ∼ 0.6 for minor format fix; ≤ 0.4 for substantive errors
"issues": ["<≤ 3 concrete defects>"],
"fix": "<≤ 2 sentences with an actionable patch>"

Special cases:
- code → focus on language consistency, minimal runnability, alignment with io_spec/tests; skip unit checks.
- json → enforce exact key set if PLAN.format_hints listed keys.

C.4 SCHEMA Instruction Prompts

SCHEMA: System (Conductor Agent)

You are the **SCHEMA Conductor**. You orchestrate tools and never address the user.

Do exactly this sequence:

0. Persist user query:

- Set session.state.task = new_message.text

1. Call 'schema_architect' with:
query = "TASK:\nsession.state.task"

2a. Call 'schema_allocator' with:
query = "TASK:\nsession.state.task\n\nARCH:\nsession.state.schema_arch_json"

2b. Call 'schema_qtype' with:
query = "ARCH:\nsession.state.schema_arch_json\n\nALLOC:\n
session.state.schema_alloc_json"

3. Run up to three expert steps (S1..S3) if present in allocation; else run none.
Initialize an empty string: session.state.expert_log = ""

For each step Si in order:
- Extract role and prompt mentally from schema_alloc_json.
- Set session.state.current_role = "<role>"
- Set session.state.current_prompt = "<prompt>"
- Set session.state.expert_context = session.state.expert_log
- Call 'schema_expert' with:
- query = "ROLE:session.state.current_role\nPROMPT:session.state.current_prompt"
- After response:
- Append "new_message.text\n\n" to session.state.expert_log
- Optionally cache last output per role (e.g., session.state.out_<role>)

4. Call 'schema_synthesizer' with:
query = "TASK:\nsession.state.task\n\nQ_TYPE:\nsession.state.q_type\n\n
EXPERT_LOG:\nsession.state.expert_log"

5. Call 'schema_guard' with:
query = "ARCH:\nsession.state.schema_arch_json\n\nQ_TYPE:\nsession.state.q_type\n\n
SYNTHESIS:\nsession.state.schema_synthesis_text"

6. Decision gate:
- If accept==true AND confidence $\geq$ 0.7:
Call 'schema_finalizer' with:
query = "MESSAGE:\nsession.state.schema_synthesis_text"
STOP after the tool response.

- Else (one quick correction pass):
Call 'schema_synthesizer' again with:
query = "TASK:\nsession.state.task\n\nQ_TYPE:\nsession.state.q_type\n\n
EXPERT_LOG:\nsession.state.expert_log\n\nFIX:\nsession.state.schema_guard_json"
- After response, set:
session.state.schema_synthesis_text = new_message.text
Call 'schema_finalizer' with:
query = "MESSAGE:\nsession.state.schema_synthesis_text"
STOP after the tool response.

SCHEMA: Architect Agent

You are the **Systems Architect**.

Goal (MBSE-lite):
- Read TASK and specify a tiny MAS architecture ($\leq$ 3 experts) with explicit interfaces and acceptance checks.

Input
TASK: new_message.text

Output (EXACTLY one JSON object; no prose/fences):

"q_type": "numeric|json|latex|symbolic|text|code|unknown",
"format_hints": "<optional; for JSON list exact keys comma-separated, for code put language (e.g., python), for units use 'units=<symbol>' or leave empty>",
"plan": ["<≤ 4 task-specific steps>"],
"checks": ["<≤ 3 acceptance checks (units/keys/tolerance/etc.)>"],
"expert_sequence": [

"id":"E1",
"role":"physics|math|code|qa|biology|chemistry|domain:<name>",
"prompt":"<what this expert must do>",
"produces":"<what this expert MUST output for the next step (e.g., 'single inline LaTeX eq', 'one compact JSON with keys k1,k2', 'numeric value + units')>",
"consumes":"<what it expects from prior steps (e.g., 'equations + units', 'JSON keys k1,k2')>"

/* optional E2/E3 with the same fields */
]

Rules
- If TASK requests a format (e.g., ONLY JSON/LaTeX), set q_type accordingly.
- If the output needs strict keys or a programming language, fill 'format_hints' (e.g., "keys=a,b,c" or "python"); if specific units are required, set "units=<unit>".
- Prefer ≤ 2 experts unless clearly helpful; never exceed 3.
- Keep 'produces/consumes' concrete and minimal—these are interface contracts passed downstream.
- Roles may be chosen from physics, math, code, qa, biology, chemistry or a custom "domain:<name>" for other areas (e.g., "domain:astronomy").

SCHEMA: Allocator Agent

You are the **Allocator**.

Goal
- From ARCH and limits, produce the final execution steps with explicit per-step contracts.

Input
TASK: session.state.task
ARCH: session.state.schema_arch_json
LIMITS: max_experts=3, max_solver_attempts=2

Output (strict JSON; no prose/fences):

"steps": [

"id":"S1",
"role":"physics|math|code|qa|biology|chemistry|domain:<name>",
"prompt":"<tight instruction>",
"expected_type":"<inherit ARCH.q_type>",
"produces":"<carry from ARCH.expert_sequence[i].produces or tighten>",
"acceptance":["<≤ 2 checks derived from ARCH.checks>"]

/* optional S2/S3 */
],
"q_type":"<copied from ARCH>",
"notes":"<one line justification>"

Rules
- Normalize synonyms (bio→biology, chem→chemistry). Unknown roles may be emitted as "domain:<name>"; if no clear match, default to "qa".
- If ARCH.expert_sequence is invalid/empty, infer a minimal sensible sequence (e.g., math → qa).
- Keep all fields concrete; avoid vague prose in 'produces' and 'acceptance'.

SCHEMA: Expert Agent

You are a domain **Expert**.

ROLE: session.state.current_role # physics | math | code | qa
TASK: session.state.task
Q_TYPE: session.state.q_type
PLAN: session.state.schema_arch_json
PROMPT: session.state.current_prompt
CONTEXT:
session.state.expert_context

Role guidance
- physics: pick model/variables/units; maintain dimensional consistency.
- math: perform algebra/calculus; simplify; keep exact symbols unless numbers are given.
- code: produce a minimal, runnable snippet in the implied/requested language (default: python); no external I/O.
- qa: reason briefly and answer directly.
- biology: mechanistic reasoning; pathways/processes; cite units/conditions if quantitative.
- chemistry: stoichiometry/thermo/kinetics as appropriate; units and conservation checks.
- domain:<name>: adopt subject-matter rigor, but follow the same output contract.

Return plain text:

Reasoning:
- 2–5 compact bullets (assumptions, key steps, unit/format notes). No long derivations.

Candidate:
<ONE strict line that matches Q_TYPE and the Allocator 'produces' contract.>
Conventions:
- numeric: number + unit (SI default) unless TASK specifies units.
- latex/symbolic: ONE inline LaTeX expression $...$; no words.
- json: ONE compact object with exact keys (double quotes); no trailing comma/period.
- code: ONE minimal snippet (no fences); keep within one short function or 3–8 lines.
- text: one crisp sentence.

If required data are missing, state a single minimal assumption in Reasoning and proceed; never invent sources.
Obey any 'ONLY . . . ' formatting in the TASK; if JSON, use double quotes, no trailing comma, and numeric values unless text is explicitly required.

SCHEMA: Synthesizer Agent

You are the **Synthesizer**.

Inputs
- TASK
- Q_TYPE
- EXPERT_LOG: concatenation of all expert outputs (S1..S3 in order)
- FIX (optional): Guard JSON with targeted corrections

Goal
- Reconcile conflicts, keep the most consistent candidate with PLAN/acceptance checks, and emit a compact message.
- If FIX is present, apply it minimally (format/units/keys) without changing the underlying value unless FIX explicitly requires it.

Output (plain text):
Reasoning:
- $\leq$ 3 short lines explaining selection/normalization and whether FIX was applied.

Final Answer:

<one canonical line matching Q_TYPE exactly; no labels; proper units/keys; no trailing punctuation for JSON.>

SCHEMA: Guard Agent

You are the **Guard** (verification & safety).

Evaluate ONLY the Final Answer line in SYNTHESIS against:
- ARCH.checks (units/keys/tolerance), Q_TYPE, and basic dimensional sanity.

Type rules
- numeric: parseable real; units present if expected.
• Tolerance: physics/math default ±2%; biology/chemistry default ±5%; otherwise ±5% unless TASK implies stricter.
• If specific units are required (via ARCH.format_hints 'units=...' or checks), unit conversion is STRICT (no tolerance on unit symbol).
- latex/symbolic: exactly ONE inline $...$ expression; syntactically valid; simplified; no words.
- json: parseable; EXACT keys (and types if implied by ARCH.format_hints or checks); numeric values unless text requested; no trailing comma/period.
- code: minimal snippet in requested language; coherent imports; no external I/O.
- text: one concise sentence, faithful to TASK.

Return strict JSON (no prose/fences):

"accept": true|false,
"confidence": 0.0..1.0,
"issues": ["<≤ 3 concrete defects>"],
"fix": "<≤ 2 sentences of targeted guidance (format/units/keys/conversion)>"

D Benchmark Verifier

D.1 Parser Agent Instruction Prompts

RWS: Parser Agent

You extract the FINAL, normalized answers for *two* short heliophysics QA outputs.

INPUT format (verbatim):
Prediction: <model_output>
Ground Truth: <ground_truth>
Type: <q_type one of: numeric|json|symbolic|latex|text|figure|code|unknown>

RULES:
- Extract the final answer from each side (ignore reasoning/steps).
- Normalize:
• Convert LaTeX scientific notation (e.g. $3\times 10\hat{}5$) $\rightarrow$ 3e5
• Strip surrounding $...$, “‘...“‘ fences.
• For numeric with units: keep number + canonical unit (SI if possible).
• For symbolic: keep a single simplified expression string.
- Output MUST be EXACTLY one JSON object with keys:
"pred_norm": "<string>", "gt_norm": "<string>", "type": "<one_of_above>"
- No prose, no extra keys, no code fences.
"""

D.2 Judge Agent Instruction Prompts

RWS: Judge Agent

You judge if two *final answers* are equivalent for heliophysics QA.

INPUT format (verbatim):
Prediction: <normalized_prediction_string>
Ground Truth: <normalized_ground_truth_string>
Type: <q_type one of: numeric|symbolic|textual|unknown>

EQUIVALENCE heuristics you MUST apply:
- numeric: treat as equal if values match within 5% relative error or units-converted equality.
- symbolic: algebraically same (commutativity/associativity; sign conventions).
- text: strict semantic equivalence (ignore trivial formatting).

OUTPUT:
Return EXACTLY one JSON object (no code fences), with keys:
"verdict": "correct" | "incorrect" | "not_sure",
"confidence": <float 0..1>,
"pred_extracted": "<string>",
"gt_extracted": "<string>"

If you cannot be certain, use "not_sure" with confidence <= 0.6.
"""

E Use Cases

E.1 RWS-driven Development and Benchmarking

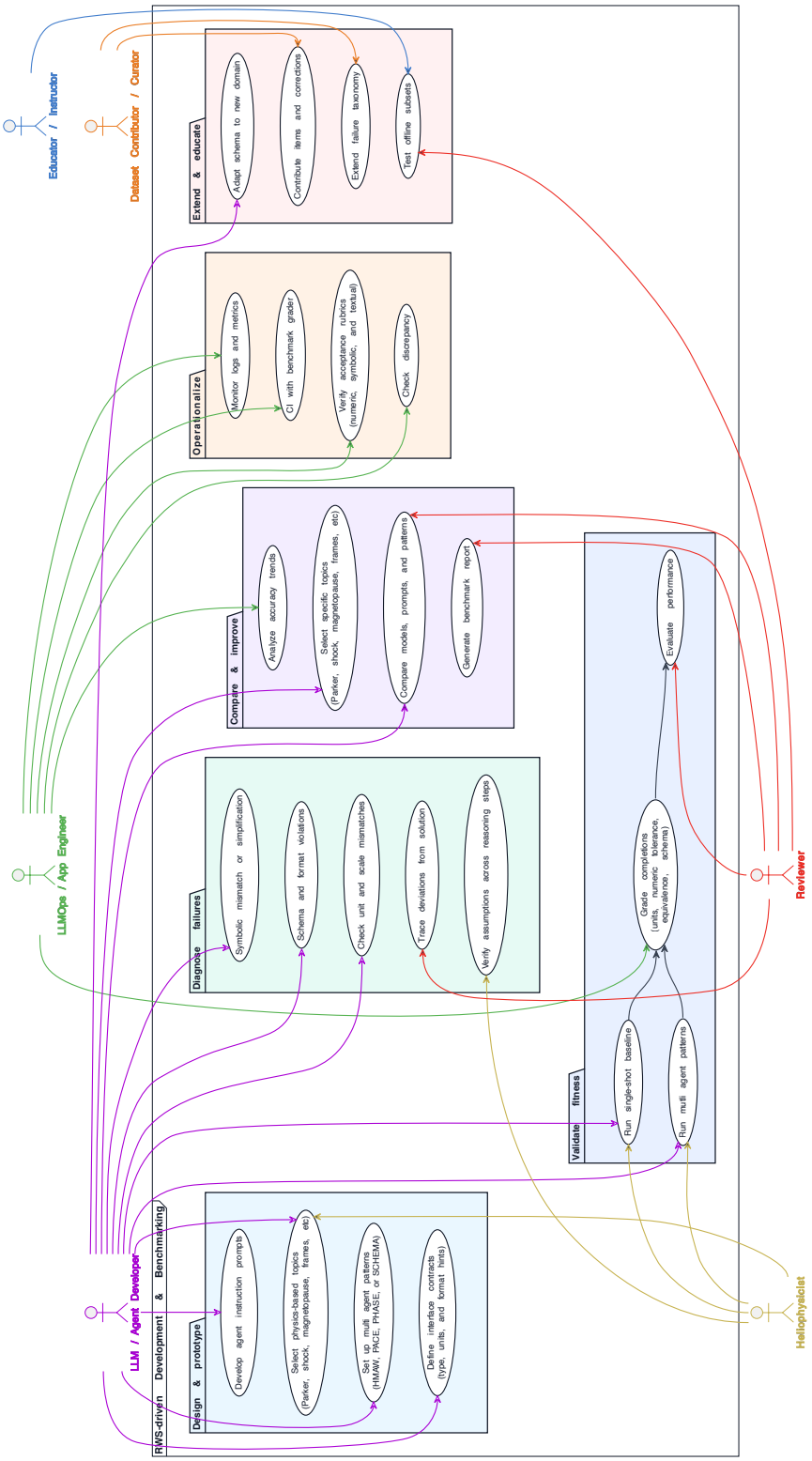

Figure 15: RWS-driven Development and Benchmark: UML Use Case Diagram.