OCCAM'S PROGRESSIVE HARNESS FOR AGENTIC MIGRATION OF LEGACY SCIENTIFIC PROGRAM

Anonymous authors
Paper under double-blind review

ABSTRACT

Migrating legacy Fortran solvers to modern stacks like JAX manually is expensive, since one solver can reach tens of thousands of lines. Coding agents with a regular harness can pursue a goal autonomously. Such a harness carries the agent's notes, or a history kept by a manager, from one session to the next, and ends the work by decision of the agent, a judge or the manager. A successful migration, however, should preserve the full scientific functionality of the legacy solver. A harness that ends the work without identifying the missing requirements and the redundant code can deliver a program of low quality. We propose Occam's Progressive Harness (OP-Harness), which takes a minimal sufficient program as its target, a program that meets every requirement and contains nothing else, and manages one verified measurement, the distance from the candidate program to that target. A builder plans and builds toward the target from the last measurement. A verifier measures the distance again as the gaps and the redundancies of the program. The two alternate on the same candidate program until both counts are zero. Across six whole legacy Fortran solvers migrated to JAX under the same model and task information, OP-Harness consistently delivers more complete programs, with stronger performance on held-out tests and post-migration probes. These results suggest that explicitly managing the remaining distance to the target can improve the reliability of autonomous whole-solver migration, bringing autonomous modernization of legacy scientific software closer to practical use.

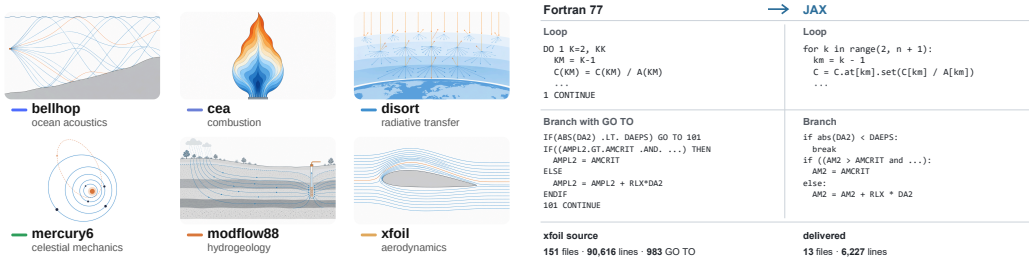

Figure 1: Background. *Left*: our released benchmark with six scientific whole legacy Fortran solvers, spanning six disciplines: ocean acoustics, combustion, radiative transfer, celestial mechanics, hydrogeology and aerodynamics. *Right*: illustration of the Fortran to JAX migration task.

1 INTRODUCTION

Many scientific solvers validated over decades are still written in Fortran. These legacy solvers remain the de facto standard of their fields, yet many are no longer maintained and have no modern replacement. Recent LLM-based coding agents have demonstrated the ability to migrate kernels and small programs automatically (Gupta et al., 2025; Kaplan et al., 2026; Moraru et al., 2026). This raises a natural question: can the same capability extend to complete legacy solvers? Migrating a complete solver is more challenging than migrating an individual kernel. A solver couples many computational components through shared state, may span tens of thousands of lines, and often has no specification beyond the source code itself (Table 1). Agentic migrations of whole

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

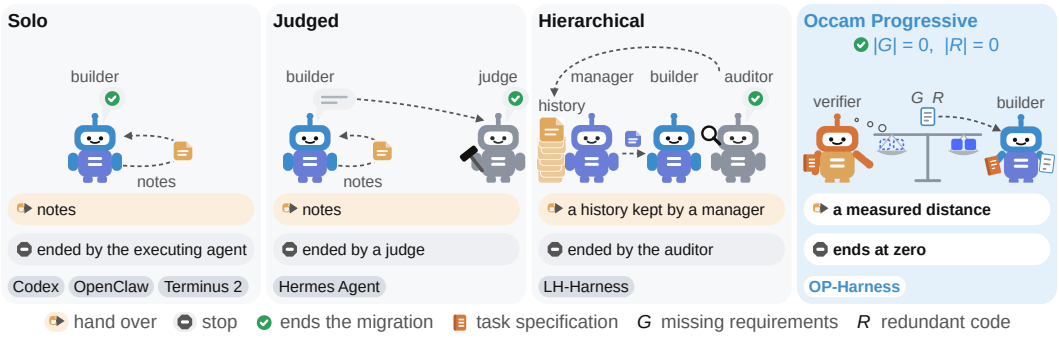

Figure 2: Harnesses for long-running tasks, classified by state transfer and termination.

solvers have begun to appear, but they remain human-in-the-loop (Shen et al., 2026; Hoshino et al., 2026; Marinov & Sfiligoi, 2026; Galletti et al., 2026). In this work, we study the fully autonomous migration of complete legacy Fortran solvers to JAX. Without human intervention, the agent must sustain progress across many sessions and determine whether the migration is complete, rather than stopping when only part of the solver has been validated.

Recent coding agents can pursue a single objective across many sessions, and owe this ability to the harnesses that orchestrate their long-running work (Young, 2025). A harness manages the state shared across sessions and the termination of the long-running task (Figure 2). A common design is for the agent to leave notes for the next session, with termination determined either by the agent itself or by an external judge. A more recent design externalizes state management to a separate manager (Ma et al., 2026). The manager maintains a history of prior sessions and uses it to plan subsequent work, while an auditor determines when the task is complete. For a solver spanning tens of thousands of lines, tests derived from the reference program typically cover only part of its functionality, so passing them does not establish that the migration is complete. At the same time, notes or maintained histories may omit information the task depends on, causing subsequent plans to diverge from the actual state of the program. These limitations can lead to premature termination or misaligned subsequent work, leaving required solver functionality incomplete. More related work is discussed in Appendix A.

We therefore define the target of a migration as a minimal sufficient program, a program that meets every requirement of the task and contains nothing else. The remaining work can then be described in two parts: gaps, the requirements still lacking, and redundancies, the components that the task does not need. A migration thus approaches the target from two directions, filling in what is missing and removing what is unnecessary, in line with Occam's principle (Blumer et al., 1987) and the maxim: as simple as possible, but not simpler.

Motivated by this view, we propose Occam's Progressive Harness (OP-Harness). OP-Harness alternates between a builder and a verifier, with each session starting from a fresh context. The builder uses the current candidate program and the verifier's latest report to plan and implement the next changes. The verifier then checks the program against the task and reports the remaining gaps and redundancies. The harness passes this report to the next builder session and ends the migration when no gaps or redundancies remain.

This paper makes three contributions.

- We formulate a migration as constructing a minimal sufficient program and propose OP-Harness with two roles: a builder that follows Occam's Progressive Strategy, satisfying one requirement at a time, and a verifier that performs Occam's Sufficiency Verification, listing the gaps and the redundancies of the current program.
- To evaluate the agentic migration of whole legacy scientific solvers to JAX, we build, to our knowledge, the first benchmark of its kind: six Fortran solvers from different disciplines, along with visible tests for development and held-out tests and probes for evaluation.
- Across six whole legacy Fortran solvers migrated to JAX under the same model and task information, OP-Harness passes 95% of both held-out tests and post-migration probes and achieves

the highest or tied-highest probe pass rate on every solver. Relative to LH-Harness, OP-Harness reduces unresolved gaps by 84% and redundancies by 52%.

2 PRELIMINARIES

2.1 TASK DEFINITION

This paper considers the *migration* task that using agents to rewrite a legacy scientific solver from Fortran stack to JAX stack and preserves every scientific function. Formally, the task is a tuple

$$T = (p_{\mathrm{src}}, C, \mathcal{V}, \mathcal{H}, W). \tag{1}$$

Here p_{src} is the source program, a legacy Fortran solver that builds and runs. C is the task contract. It names the target stack, the input scope, the output format with tolerances, and the prohibited practices, such as calling the source program. $\mathcal{V}$ is the set of visible tests, given to the agent with the contract. $\mathcal{H}$ is the set of held-out tests, withheld from the agent until the final evaluation. The final evaluation also runs probes, further tests generated after the migrations. The probes spread over the input scope of C, and p_{src} supplies the reference answers. W is the time limit of a migration. The agent receives p_{src}, C, and $\mathcal{V}$ with the tools of both stacks and outputs the delivered program.

A legacy Fortran solver contains thousands to tens of thousands of lines (Table 1). Migrating such a solver to JAX is therefore a long-running coding task that spans multiple agent sessions (Young, 2025). At the start of session t, the agent has a fresh context with context budget B and resumes the program p_{t-1} produced in the previous session. In particular, $p_0 = p_{\mathrm{src}}$. Within session t, the agent plans an ordered sequence of program edits $a_t = (s_{t,1}, \ldots, s_{t,K_t})$, where $s_{t,k}$ denotes the k-th edit and K_t is the number of edits in this session. It then applies these edits one by one. Let $p_t^{(k)}$ denote the program after the first k edits in session t. The program evolves as follows:

$$p_{t-1} = p_t^{(0)} \xrightarrow{s_{t,1}} p_t^{(1)} \xrightarrow{s_{t,2}} \cdots \xrightarrow{s_{t,K_t}} p_t^{(K_t)} = p_t. \tag{2}$$

The *harness* links the sessions of Eq. 2 into one migration. Each session starts from a state, the material that the harness passes from the previous session. At the end of session t, it decides whether to terminate and, if the task continues, which state to pass to session $t+1$. Upon termination, the current candidate program is delivered. Figure 2 classifies harnesses by these rules for state transfer and termination.

2.2 ANALYSIS OF PREVIOUS MIGRATION HARNESSES

For the convenience of analyzing a migration, let $Q = \{q_1, \ldots, q_m\}$ denote the checkable requirements that a completed migration must satisfy. Here, the challenge is that passing the visible tests cannot establish that every requirement in Q is met. These tests cover only part of the source program, and finite tests in general cannot establish correctness (Dahl et al., 1972). The regular harnesses in Figure 2 make the state transfer and termination decisions in three types:

1. **Solo.** The agent writes notes at the end of each session for the next session to read. The task stops when the agent declares completion (Codex in goal mode, OpenClaw, and Terminus 2; Merrill et al., 2026).
2. **Judged.** The agent's notes are passed to the next session, but a separate judge decides whether to stop by reading the task goal and the agent's latest responses (Hermes Agent).
3. **Hierarchical.** A manager retains the note history and uses it to plan the next session. An auditor inspects the program and tests without modifying the program. The task stops when the auditor reports every item in the manager's plan complete (LH-Harness; Ma et al., 2026).

Despite different termination rules, all three types pass the current program p_t and a summary h_t to the next session to guide subsequent edits, where h_t contains agent notes in Solo and Judged and the manager's note history in Hierarchical harness. We denote the next session's edit plan as

$$a_{t+1} = \mathrm{Plan}(T, p_t, h_t; B). \tag{3}$$

As a summary of earlier session, h_t cannot establish which requirements in Q the current program satisfies. As a result, the above three types of harnesses suffers from the three limitations.

First, *the summary does not help manage the progress of a large long-running coding task.* The summary mostly lists the actions and observations of the agent. The next session, however, needs the requirements in Q that the current program does not yet satisfy. Moreover, the budget B keeps the notes brief. A requirement that no visible test covers and the summary omits is then likely overlooked in subsequent sessions as well. Even a requirement recorded as satisfied may fail after a later edit. The manager of a Hierarchical harness even plans from the history alone and does not inspect the current program. Therefore, the longer the migration runs, the less the summary reveals about the progress of the task.

Second, *redundant code accumulates*. In every session, the agent starts from a fresh context, reads the summary, and must investigate the program again within the budget B. Meanwhile, superseded implementations, code that never runs, and debugging scripts keep increasing over a long-running migration that can last 24 hours or more. The redundant code causes no test failure, and the summary never reports such code. However, the harnesses mentioned above take no measures to remove the code. The code thus fills the workspace, occupies the valuable context of the agent, and may lead the agent to edit a superseded implementation. As the sessions pass, the program becomes ever harder to investigate and modify.

Third, *neither the plan nor the state is well-structured*. This structure should take the requirements in Q and the parts of the program into consideration. However, the notes and the history are narratives of sessions. The next session therefore cannot read the progress of each requirement from the narrative. Likewise, the agent improvises the plan of a session, and the edits do not correspond to the requirements one to one. Even in Hierarchical harnesses, the manager plans a list of steps, and the auditor checks only the completion of these steps. As a result, both execution and termination become highly arbitrary. During execution, the next step depends on the impression that the agent assembles from the narrative. At termination, the agent's declaration, the judge's decision, and the auditor's check rest on the same plan and summary without structure. Hence, none of the three can establish that every requirement in Q is satisfied.

A migration can therefore end once the visible tests pass, with requirements in Q incomplete and much redundant code left. Such migrations may deliver programs that lack functions of the solver and vary in quality.

3 Occam's Progressive Harness

3.1 Occam's Principle of Migration

To address the three limitations above, we formulate a migration as constructing a minimal sufficient program. This formulation borrows Occam's principle of sufficiency and simplicity (Blumer et al., 1987). Sufficiency means no missing requirement, and simplicity means no redundant part. For a program p, let $G(p) \subseteq Q$ be the set of requirements that p does not satisfy, the gaps of p. Let $R(p)$ be the set of parts of p that the task does not need, the redundancies of p. The distance of p is the number of gaps and redundancies of p,

$$d(p) = |G(p)| + |R(p)|. \tag{4}$$

A program with $G(p) = R(p) = \emptyset$ satisfies every requirement in Q and contains no redundancy. Such a program is a minimal sufficient program and a target of the migration. Since the current program p_t holds the results of all earlier iterations, $G(p_t)$ and $R(p_t)$ alone describe the remaining work. In contrast to the regular harnesses, we therefore need no summary of the session history but only the distance. A harness should direct the agents to reduce this distance from two directions, by satisfying the missing requirements and by removing the redundancies.

3.2 State and Termination of OP-Harness

To manage this distance, OP-Harness structures both the plan and the state. Since the task description does not list Q, the harness cannot compute $d(p_t)$ directly. OP-Harness therefore assigns an agent, the verifier, to examine p_t. The verifier writes a report r_t with two lists, G_t and R_t, the readings of $G(p_t)$ and $R(p_t)$. A second agent, the builder, plans and implements the next edits from the

Algorithm 1 OP-Harness loop on the candidate program

Input: task T, context budget B of one builder session, cap N
Let $Q = \{q_1, \ldots, q_m\}$ denote the latent set of checkable requirements induced by T; Q need not be explicitly enumerated by the harness.
Initialize: $p_0 \leftarrow$ the initial program; $r_0 \leftarrow$ the empty report ▷ one program, one report
for $t = 1 \ldots N$ **do** ▷ one iteration: two sessions, each with a fresh context
  $a_t = (q_{i_{t,1}}, \ldots, q_{i_{t,K_t}}) \leftarrow \mathrm{Plan}(T, p_{t-1}, r_{t-1}; B)$ ▷ decisive first, to limit redundancy
  $p_t \leftarrow p_{t-1}$ with a_t implemented ▷ Occam's Progressive Strategy
  $r_t = (G_t, R_t) \leftarrow \mathrm{Verify}(T, p_t)$ ▷ Occam's Sufficiency Verification
  if $G_t = \emptyset$ **and** $R_t = \emptyset$ **then break** ▷ the distance $|G_t| + |R_t|$ is zero
return p_t ▷ the delivered program

current program and the report,

$$\begin{aligned} r_t &= \mathrm{Verify}(T, p_t), \\ (G_t, R_t) &= r_t, \\ a_{t+1} &= \mathrm{Plan}(T, p_t, r_t; B). \end{aligned} \tag{5}$$

The two agents alternate for at most N iterations.

Equation 5 differs from Equation 3 only in the third argument of the plan. Agents with a regular harness plan from a summary of earlier work, and OP-Harness enables the agents to plan from a measurement of the current program. OP-Harness keeps the state that passes between iterations as two objects, the candidate program p_t and the last report r_t. The harness keeps no history of the sessions. An iteration of OP-Harness consists of a builder session and a verifier session, and each session starts from a fresh context. In Equation 5 and Algorithm 1, the index t counts iterations. Only the builder session of iteration t edits the program, so iteration t corresponds to session t in Equation 2. At the end of an iteration, the harness passes this state to the builder of the next iteration. The system prompts of the two agents are in Appendix D. The harness ends the migration once G_t and R_t are both empty. The cap N and a cap on the duration of a session also end the migration. Once the migration ends, the most recent candidate program is the delivered program. In the terms of Figure 2, OP-Harness passes a measured distance and ends the migration at zero distance. Algorithm 1 summarizes the loop of OP-Harness.

Occam's Progressive Strategy. The builder follows Occam's Progressive Strategy. The builder receives the strategy through a system prompt. The strategy aims at the full satisfaction of the requirements with as little redundancy as possible. The purpose is to keep redundancy from occupying the context of later sessions. At the start of a session, the builder reads the current program and the last report. The builder checks each gap and each redundancy of the report against the program. The builder then adds the accepted entries from the report to the plan and records a reason for each entry that the builder declines. The builder plans K_t edits at the start of the session. Each edit either satisfies one requirement completely or removes one redundancy. For the order of the edits, the builder ranks the requirements that remain from the current state to the completion of the task. The rank follows the importance of each requirement to the correctness of the program. A requirement that other requirements rely on counts as more important. This order aims to reduce redundancy, since later edits then seldom replace earlier code. Satisfying one requirement completely at a time in the order of the plan is the meaning of progressive in the name of the strategy. The K_t edits produce the next candidate program p_t toward the completion of the task. The strategy also asks the builder to add no structure that the task does not need. The verifier finally checks whether the builder follows this rule.

Occam's Sufficiency Verification. The verifier performs Occam's Sufficiency Verification. Starting from a fresh context, the verifier reads the task and the candidate program. The verifier outlines a program that would meet every requirement of the task. The candidate program is then compared with this outline in two directions. The verifier runs the visible tests, adds further checks if the visible tests are insufficient, and never edits the program. The report has two lists, one for gaps and one for redundancies. Each requirement that the verifier judges to be missing enters the list of gaps. A part of the program enters the list of redundancies if the verifier judges that the task remains

Table 1: Our collected six migration solvers and their reference Fortran solvers.

Solver	Discipline	Files	Lines
bellhop (Porter & Bucker, 1987)	Ocean acoustics	43	12,367
cea (Gordon & McBride, 1994)	Combustion	4	7,015
disort (Stamnes et al., 1988)	Radiative transfer	11	23,064
mercury6 (Chambers, 1999)	Celestial mechanics	5	12,230
modflow88 (McDonald et al., 1988)	Hydrogeology	21	11,261
xfoil (Drela, 1989)	Aerodynamics	151	90,616

complete without the part. The list of gaps is G_t, and the list of redundancies is R_t. The harness passes the report to the builder of the next iteration.

4 Experiments

4.1 Setup

Solvers. Our evaluation covers six scientific domains through six contract-scoped Fortran-to-JAX migrations, as shown in Table 1: (1) **bellhop** (Porter & Bucker, 1987), a beam-tracing program for underwater acoustic propagation. (2) **cea** (Gordon & McBride, 1994), a code for chemical equilibrium and rocket performance from the NASA Glenn Research Center. (3) **disort** (Stamnes et al., 1988), a solver of the radiative transfer equation by the discrete ordinates method. (4) **mercury6** (Chambers, 1999) , an integrator of planetary and small-body systems in celestial mechanics. (5) **modflow88** (McDonald et al., 1988), the modular finite-difference model of subsurface water flow from the U.S. Geological Survey. (6) **xfoil** (Drela, 1989), an airfoil analysis program that couples a panel method to an integral boundary-layer model. Appendix B describes the workflow and exclusions of each solver. These publicly available solvers are long-standing, representative, and classic codes in their respective domains, and together they span a wide range of numerical methods, interfaces, and problem scales, allowing us to test the generalization of a single harness. Their appropriate difficulty level also makes them discriminative among current models and harnesses. Each solver has ten held-out tests and at least ten visible tests, and no held-out test is visible during a migration. For bellhop, both sets come from the example cases of the source program. For disort, both sets mix such cases with variants that change the parameters. For the other four solvers, the examples of the source programs are too few or fail our checks. A script therefore draws both sets reproducibly from a grid of input parameters. We release the code that builds these test suites with the task packages in an anonymous repository.

Compared methods. We compare OP-Harness with Codex, OpenClaw, LH-Harness (Ma et al., 2026), Hermes Agent, and Terminus 2 (Merrill et al., 2026). They span agent-led completion, a separate judge, and manager–auditor control: contrasting ways of organizing and stopping a long migration. All coding agents use DeepSeek V4.1 Flash at high reasoning effort. OP-Harness and LH-Harness also share Codex as their coding agent. We exclude Claude Code from our comparison as it is not open-source, and Anthropic's LLM-gateway policy restricts non-Claude models. Evaluating with Claude would confound our comparisons. For each solver, every harness receives the same source and solver-specific migration requirements and runs with two physical cores, 4 GB of memory, and a 24-hour limit. OP-Harness additionally has a 16-iteration cap. We run three trials. In each trial, every harness performs one migration on each of the six solvers. Appendix C gives the full protocol.

Evaluation metrics. To evaluate whether a program fulfills the source functionality specified by the task contract we select distinct test cases across our six solvers. Each case supplies an input set to the program and verifies that it produces valid outputs with all required results matching the Fortran reference within empirically measured tolerances. Cases exposed to the agent during migration are termed visible cases. We reserve held-out cases to assess the program's behavior on unseen inputs after migration completes. Each solver admits a large space of input configurations. For example, bellhop supports regular and irregular receiver layouts so limited held-out cases cannot exhaust all input options. We therefore design dedicated probe cases targeting these input modalities. Each probe consists of concrete inputs and is scored independently under the same criteria as held-out

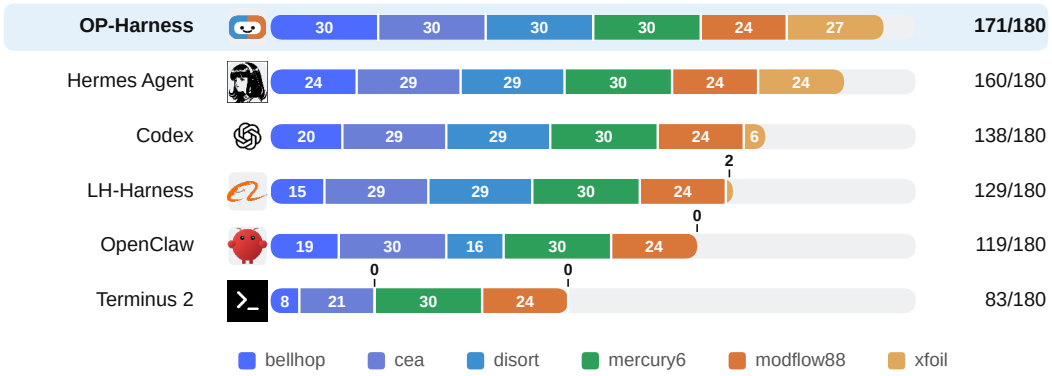

Figure 3: Held-out tests passed by the delivered programs of each harness, summed over the three independent trials of migrations for each of the six solvers, with ten tests per trial.

Table 2: Results on the six Fortran to JAX migration solvers: the pass rate on the probes, as mean±std over three trials. The best value of a column is in bold.

Harness	bellhop	cea	disort	mercury6	modflow88	xfoil	Mean
Codex	.77±.20	.83±.09	.97±.03	1.00±.00	.80±.03	.12±.21	.75±.01
OpenClaw	.73±.10	.77±.05	.64±.28	1.00±.00	.79±.02	.00±.00	.66±.05
LH-Harness	.67±.20	.61±.05	.96±.08	1.00±.00	.77±.03	.07±.12	.68±.01
Terminus 2	.23±.35	.23±.05	.00±.00	1.00±.00	.70±.06	.00±.00	.36±.05
Hermes Agent	.83±.15	.77±.11	.91±.15	1.00±.00	**.82**±.02	.96±.05	.88±.02
OP-Harness (ours)	**1.00**±.00	**.92**±.08	**1.00**±.00	1.00±.00	**.82**±.04	**.98**±.02	**.95**±.02

cases. A case passes only if all four conditions hold simultaneously. First it generates complete well-formatted outputs. Second all required results agree with the Fortran reference within empirically measured tolerances. Third it passes applicable physical-relation certificate checks. Fourth execution finishes within the time limit. Multiple probes may correspond to the same input modality. We analyze delivered programs along two aspects result correctness and robustness across diverse valid inputs. We separately report pass rates for held-out cases and probe cases. Additionally we use Input Output and Property dimensions to evaluate input options output fields physical constraints and implementation properties combining test outcomes with code inspection. The scoring functions and the probes compute every score in Tables 2 and 3. LLM graders assign only the entries that no scoring function or probe examines. Tables 6 and 7 and Figure 5 report these grades.

4.2 Results and Analysis

Comparison of program quality with other harnesses. Figure 3 compares held-out tests passed by the delivered programs of each harness. OP-Harness achieves the strongest overall behavioral performance, passing 95% of the held-out tests, better than 89% for Hermes Agent, the best baseline. Codex, LH-Harness, and OpenClaw achieve lower overall pass rates, while Terminus 2 performs worst at 46%. These differences are not distributed uniformly across tasks. All six harnesses pass every held-out test on mercury6, and the stronger harnesses also perform similarly on cea and disort. The separation becomes much larger on the more difficult bellhop and xfoil migrations. On xfoil in particular, OP-Harness and Hermes Agent still pass most held-out tests, whereas Codex, LH-Harness, and OpenClaw degrade substantially, with OpenClaw passing none.

The post-migration probes reinforce this overall picture while providing a finer-grained test over a broader range of inputs (Table 2). OP-Harness reaches an average probe pass rate of 0.95, still outperforms 0.88 of Hermes Agent, much better than all other harnesses that are no more than 0.75. More importantly, OP-Harness achieves the highest or tied-highest probe pass rate on all six solvers. On xfoil, OP-Harness still reaches 0.98 and Hermes Agent also remains strong, whereas Codex and LH-Harness fall to very low pass rates and OpenClaw and Terminus 2 fail entirely.

Table 3: Multi-dimension scores, execution coverage, and token cost. Each entry is averaged over three trials. The best value of a column is bold, and for Cost the lowest value is best.

Harness	Output	Property	Input	Execution coverage	Cost (M tokens, six solvers)
Codex	.80±.02	.93±.04	.76±.02	.73±.08	**7.6**±2.7
OpenClaw	.73±.01	.90±.09	.67±.03	.63±.03	9.0±4.9
LH-Harness	.74±.03	.90±.03	.69±.02	.73±.09	24.9±6.0
Terminus 2	.59±.12	.73±.04	.52±.12	.61±.02	8.9±2.9
Hermes Agent	.93±.03	.95±.03	.89±.02	.76±.04	15.4±3.5
OP-Harness (ours)	**.98**±.01	**1.00**±.00	**.96**±.02	**.82**±.02	25.8±3.2



Figure 4: (a) The gaps G and (b) the redundancies R that the same verifier reports on the last recorded state of each migration of LH-Harness and OP-Harness, in the three trials of each solver.

The multi-dimension metrics provide a complementary view of the delivered programs (Table 3). OP-Harness covers 96% to 100% of three measured dimensions output, property and input. OP-Harness shows no pronounced deficit on any one dimension, while all the other harnesses exhibit inferior performance in at least one aspect of the required functionality. OP-Harness also achieves the highest execution coverage 82%.

Progress management in long-running migration To examine whether progress is represented more effectively during long-running migration, we assess the final program states of OP-Harness and LH-Harness post hoc using the same verifier. Figure 4 shows that OP-Harness reduces the total number of unresolved gaps from 58 to 9, an 84% reduction, while increasing the fraction of gap-free runs from 17% to 67%, compared to LH-Harness. The remaining runs contain only one or two gaps each. Because each report lists the remaining gaps for the next builder, OP-Harness achieves this reduction.

OP-Harness also improves progress management by controlling redundant structure. It reduces the number of redundancies reported by the verifier from 27 to 13, while increasing the fraction of redundancy-free runs from 22% to 61%. The independently evaluated delivered programs show the same pattern (Figure 5). For superseded code, OP-Harness has a share of 25%, while the five baselines range from 53% to 64%. For unreachable code, OP-Harness is at 56%, compared with 78% to 83% for the baselines. The reduction is consistent across the independently measured structural properties, indicating a broader simplification of the delivered programs. The agreement between verifier reports and the independent program evaluation suggests that OP-Harness's redundancy tracking corresponds to tangible reductions in unnecessary code structure, not just changes in its internal progress state.

Inference cost Over the six solvers, OP-Harness uses 25.8 ± 3.2 million tokens, about 68% more than Hermes Agent, and has the highest monetary cost in Figure 6. At peak pricing, it costs \$6.56 per migration, about $2.2\times$ the \$3.00 cost of Hermes Agent. No cheaper harness reaches comparable quality. OP-Harness therefore extends the cost–quality frontier toward higher migration quality. The comparison with LH-Harness further shows that spending more inference alone does not explain the gain. LH-Harness consumes almost the same number of tokens as OP-Harness, 24.9M versus

25.8M, yet its probe pass rate is only 0.68 and 39% of its runs pass all held-out tests, compared with 72% for OP-Harness.

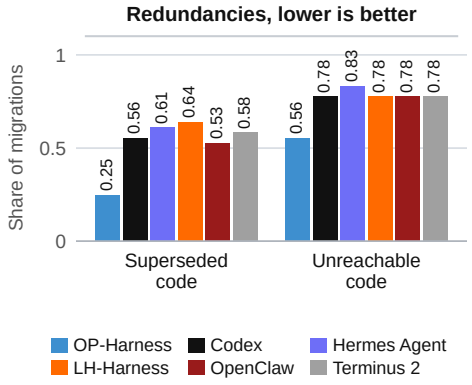

Figure 5: Share of migrations containing superseded or unreachable code across six solvers (three trials/harness). Evaluated by LLM graders (Appendix C), the share is calculated as $1 -$ mean grade (Pass=0, Partial=0.5, Fail=1). Lower is better; numbers above bars indicate exact shares.

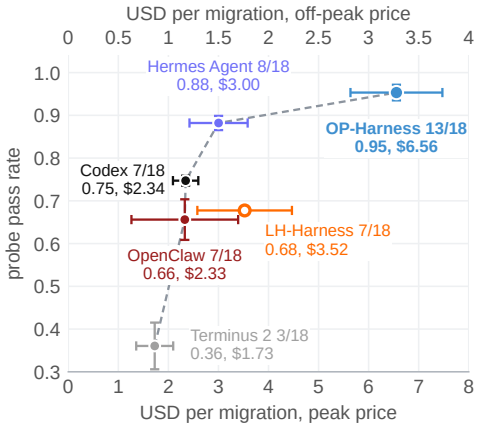

Figure 6: Probe pass rate vs. cost (DeepSeek V4.1 Flash: bottom axis=peak, top=off-peak) as mean$\pm$std over 3 trials. The dashed line denotes the Pareto frontier; hollow points are suboptimal. Each $k/18$ indicates complete migrations passing all 10 held-out tests. Values below names represent mean pass rate and peak cost.

5 Limitations

The learning guarantee behind Occam's principle (Blumer et al., 1987) depends on the relation between the number of samples and the size of the hypothesis. The migrated programs have thousands of lines, and a task provides only ten to sixteen visible tests. This premise therefore does not hold in our setting. We borrow only the two conditions of sufficiency and simplicity, and our experiments measure the performance on the held-out tests and the probes.

6 Conclusion

We study the agentic migration of whole legacy scientific software and introduce Occam's Progressive Harness for migrating legacy Fortran solvers to JAX. OP-Harness treats migration as reaching a minimal sufficient program and carries a verified measurement of the remaining gaps and redundancies across fresh-context builder and verifier sessions. Across six whole scientific solvers, under the same model and task information, OP-Harness leaves less unresolved gaps and redundancies than LH-Harness and outperforms five regular harnesses on held-out tests and post-migration probes, while also producing cleaner program structure. These results support explicitly managing the remaining work as a useful organizing principle for long-running migration, rather than relying only on session history and a final completion judgment. The gains come at higher inference cost, but extend the cost-quality frontier beyond the compared harnesses. Future work should make progress management more efficient by reducing repeated verification without losing track of remaining work, enabling autonomous migration to scale to large legacy scientific software.

AI USE STATEMENT

Generative AI tools assisted with manuscript polishing and literature discovery. DeepSeek V4.1 Flash graded the rubric entries that require a grader (Appendix C). Generative AI tools were not used for scientific ideation or methodology development. The authors take full responsibility for the final manuscript.

ETHICS STATEMENT

This work involves no human subjects and no personal data. The six reference solvers are publicly available scientific software. The source code of CEA and DISORT is subject to licenses that restrict redistribution. Accordingly, our release includes the task packages, tests, and migrated programs, but not the original CEA or DISORT source code. Users must obtain these sources from their original distributions. The migrated programs are research artifacts and have not been validated for production use.

REPRODUCIBILITY STATEMENT

Appendix B describes the six solvers and the tasks, Appendix C the evaluation protocol with the visible and held-out tests, the probes and the execution coverage, and Appendix D the complete instructions given to the builder and the verifier. The harness, the task packages and the evaluation scripts are released at an anonymous repository, `https://anonymous.4open.science/r/OP-Harness-665E`.

REFERENCES

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, Balasubramanyan Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE): 675–698, 2024.

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5): 61–72, 1988.

John E Chambers. A hybrid symplectic integrator that permits close encounters between massive bodies. *Monthly Notices of the Royal Astronomical Society*, 304(4):793–799, 1999.

Le Chen, Pei-Hung Lin, Tristan Vanderbruggen, Chunhua Liao, Murali Emani, and Bronis De Supinski. Lm4hpc: Towards effective language model application in high-performance computing. In *International Workshop on OpenMP*, pp. 18–33. Springer, 2023.

Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972.

Joshua H Davis, Daniel Nichols, Ishan Khillan, and Abhinav Bhatele. Pareval-repo: A benchmark suite for evaluating llms with repository-level hpc translation tasks. In *Proceedings of the 54th International Conference on Parallel Processing*, pp. 94–103, 2025.

Akash Dhruv and Anshu Dubey. Leveraging large language models for code translation and software development in scientific computing. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pp. 1–9, 2025.

Pedro Domingos. The role of occam's razor in knowledge discovery. *Data mining and knowledge discovery*, 3(4):409–425, 1999.

Mark Drela. Xfoil: An analysis and design system for low reynolds number airfoils. In *Low Reynolds number aerodynamics: proceedings of the conference notre dame, Indiana, USA, 5–7 June 1989*, pp. 1–12. Springer, 1989.

Víctor Gallego. Metal-sci: A scientific compute benchmark for evolutionary llm kernel search on apple silicon. *arXiv preprint arXiv:2605.09708*, 2026.

Gianluca Galletti, Eric Volkmann, and Johannes Brandstetter. gyaradax: Local gyrokinetics jax code. *arXiv preprint arXiv:2604.06085*, 2026.

William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. Evaluation of openai codex for hpc parallel programming models kernel generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, pp. 136–144, 2023.

Sanford Gordon and Bonnie J McBride. Computer program for calculation of complex chemical equilibrium. *NASA reference publication*, 1311(4-5):19–20, 1994.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Sparsh Gupta, Kamalavasan Kamalakkannan, Maxim Moraru, Galen Shipman, and Patrick Diehl. From legacy fortran to portable kokkos: An autonomous agentic ai workflow. *arXiv preprint arXiv:2509.12443*, 2025.

Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

Deyao Hong, Yizhe Chi, Wenyi Li, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. Swe refactor bench: Can coding agents complete a long-horizon, whole-repository stack migration? *arXiv preprint arXiv:2608.23564*, 2026.

Tetsuya Hoshino, Masaya Kato, Kazuhisa Tsuboki, Daichi Mukunoki, Takahiro Katagiri, and Toshihiro Hanawa. Validation-centric ai-assisted gpu porting of a 250,000+ line legacy weather simulation code. *arXiv preprint arXiv:2608.13122*, 2026.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *Proceedings of the ACM on Software Engineering*, 2(FSE):2454–2476, 2025.

Erel Kaplan, Tomer Bitan, Lian Ghrayeb, Le Chen, Tom Yotam, Niranjan Hasabnis, and Gal Oren. Paracodex: A profiling-guided autonomous coding agent for reliable parallel code generation and translation. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16113–16136, 2026.

Nikolay V Koldunov, Suvarchal K Cheedela, Sergey Danilov, Dmitry Sidorenko, Sebastian Beyer, and Thomas Jung. An ocean model ported by a large language model: Experience and lessons from fesom2 (fortran to c to c++/kokkos). *arXiv preprint arXiv:2606.11356*, 2026a.

Nikolay V Koldunov, Sergey Danilov, Suvarchal Cheedela, Dmitry Sidorenko, Sebastian Beyer, Patrick Scholz, Ivan Kuznetsov, Jan Streffing, Aleksei Koldunov, Dmitrii Pantiukhin, et al. Fesom2-jax v1. 0: a differentiable shadow of the ocean-sea-ice model fesom2, cast onto gpus. *arXiv preprint arXiv:2608.01546*, 2026b.

Aya Lahlou, Linnia Hawkins, and Pierre Gentine. Systematic llm translation of legacy scientific code to differentiable frameworks: Application to a land surface model. *arXiv preprint arXiv:2606.07681*, 2026.

Ziyu Ma, Hailang Huang, Shun Zou, Yong Wang, Shidong Yang, Yiming Hu, Fei Wei, and XiangXiang Chu. Longhorizon-harness: Advancing long-horizon agents for real-world tasks. *arXiv preprint arXiv:2608.01964*, 2026.

Anthony Marinov and Igor Sfiligoi. Structuring agentic ai for hpc code modernization. *arXiv preprint arXiv:2606.08710*, 2026.

Alex Mathai, Shobini Iyer, Aleksandr Nogikh, Petros Maniatis, Franjo Ivancic, Junfeng Yang, and Baishakhi Ray. Trim: Reducing ai-generated codeslop via agent trajectory minimization. *arXiv preprint arXiv:2607.18161*, 2026.

Loïc Maurin. fortranspire: Llm + mcp pipeline porting legacy fortran hpc kernels to openacc gpu and differentiable jax, June 2026. URL https://doi.org/10.5281/zenodo.20828814.

Michael G McDonald, Arlen W Harbaugh, et al. *A modular three-dimensional finite-difference ground-water flow model*, volume 6. US Geological Survey Reston, VA, USA, 1988.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Maxim Moraru, Kamalavasan Kamalakkannan, Jered Dominguez-Trujillo, Patrick Diehl, Atanu Barai, Julien Loiseau, Zachary Kent Baker, Howard Pritchard, and Galen M Shipman. Laafd: Llm-based agents for accelerated fpga design. *arXiv preprint arXiv:2602.06085*, 2026.

Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pp. 281–294, 2024.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Rocco Pietrini, Marina Paolanti, and Emanuele Frontoni. Bridging eras: Transforming fortran legacies into python with the power of large language models. In *2024 IEEE 3rd International Conference on Computing and Machine Intelligence (ICMI)*, pp. 1–5. IEEE, 2024.

Michael B Porter and Homer P Bucker. Gaussian beam tracing for computing ocean acoustic fields. *The Journal of the Acoustical Society of America*, 82(4):1349–1359, 1987.

Mohsen Sadr, Emmanuel Lanti, Alexey Mishchenko, Xin Wang, and Laurent Villard. Orb5x v1. 0: a performance-portable global electromagnetic gyrokinetic pic code in c++/kokkos built using agentic ai. *arXiv preprint arXiv:2607.17363*, 2026.

Yuzhong Shen, Masha Sosonkina, Peng Xu, and Mark S Gordon. An agentic workflow for legacy hpc modernization: Converting the two-electron-integral core of gamess. *arXiv preprint arXiv:2608.12249*, 2026.

Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A Seshia, and Koushik Sen. Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis. *arXiv preprint arXiv:2412.14234*, 2024.

Shivamshan Sivanesan and Kazem Ardaneh. A fortran general-purpose transpiler: Proof of concept. *arXiv preprint arXiv:2608.00130*, 2026.

Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pp. 532–543, 2015.

Knut Stamnes, S-Chee Tsay, Warren Wiscombe, and Kolf Jayaweera. Numerically stable algorithm for discrete-ordinate-method radiative transfer in multiple scattering and emitting layered media. *Applied optics*, 27(12):2502–2509, 1988.

Geoffrey I Webb. Further experimental evidence against the utility of occam's razor. *Journal of Artificial Intelligence Research*, 4:397–417, 1996.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv preprint arXiv:2507.17773*, 2025.

Justin Young. Effective harnesses for long-running agents. *Anthropic Engineering Blog, Nov*, 2025.

Wenting Zhao, Nan Jiang, Celine Lee, Justin Chiu, Claire Cardie, Matthias Gallé, and Alexander Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations*, volume 2025, pp. 12061–12076, 2025.

Anthony Zhou, Linnia Hawkins, and Pierre Gentine. Proof-of-concept: using chatgpt to translate and modernize an earth system model from fortran to python/jax. *arXiv preprint arXiv:2405.00018*, 2024.

A Related Work

Migration of scientific software. Language models translate single kernels and small programs from Fortran into Python (Pietrini et al., 2024) and JAX (Zhou et al., 2024; Maurin, 2026). Benchmarks and case studies evaluate language models on parallel code and single kernels (Godoy et al., 2023; Chen et al., 2023; Nichols et al., 2024; Wen et al., 2025). At the level of repositories, ParEval-Repo finds that build systems and dependencies across files make mini-applications hard to translate between GPU programming models (Davis et al., 2025). The migrations of whole scientific applications reported so far keep a human in the loop at some stage (Dhruv & Dubey, 2025; Koldunov et al., 2026a; Sadr et al., 2026; Lahlou et al., 2026; Koldunov et al., 2026b; Shen et al., 2026; Hoshino et al., 2026; Marinov & Sfiligoi, 2026; Galletti et al., 2026), and the authors of one Fortran transpiler argue that language models fail at this scale (Sivanesan & Ardaneh, 2026). The present paper evaluates migrations that run autonomously. Among benchmarks, Commit0 scores whole libraries reproduced from scratch with unit tests (Zhao et al., 2025), SWE Refactor Bench scores migrations of whole repositories of general software and first checks that each migration took place (Hong et al., 2026), Metal-Sci scores the optimization of single scientific kernels against a roofline target (Gallego, 2026), and KernelBench scores single GPU kernels (Ouyang et al., 2025). To our knowledge, no existing benchmark evaluates the migration of whole legacy solvers by numerical equivalence with the reference program.

Finite tests and Occam's principle. Tests sample the behavior of a program and cannot establish its correctness (Dahl et al., 1972). Many programs pass the same tests, but not all of them are correct (Gulwani, 2011; Smith et al., 2015). A migration therefore needs a target beyond the tests, and OP-Harness takes it from the task and the reference program. In learning theory, an Occam algorithm returns a hypothesis that is consistent with all observations and close to minimal in size, and such a hypothesis generalizes to unseen samples (Blumer et al., 1987). OP-Harness takes the two conditions together as the target of a migration. Simplicity alone does not make a hypothesis more accurate (Domingos, 1999; Webb, 1996), so the two roles hold minimality as an aim, and sufficiency decides correctness. The aim has an empirical motive. Code written by agents carries speculative edits and abandoned hypotheses into the final patch (Mathai et al., 2026), and is more verbose than open-source code (Orlanski et al., 2026). The order in which the builder works has precedents in the fail-first principle of constraint satisfaction (Haralick & Elliott, 1980), the risk-driven spiral model (Boehm, 1988), and one feature per session (Young, 2025). Systems that translate whole repositories order the work by the dependencies of the source program (Shetty et al., 2024; Ibrahimzada et al., 2025; Bairi et al., 2024). The builder orders it by the task, and the requirement most important to correctness comes first.

B The Reference Solvers

This appendix describes the reference software of the six rows of Table 1. For each solver it names the discipline, the computation, the numerical method, and the part of the reference program that the task migrates.

bellhop. BELLHOP is a beam-tracing program for underwater acoustic propagation. From one environmental deck that gives the sound-speed profile, the boundaries, the source, the launch angles, and the receiver, it computes the pressure field, the ray trajectories, or the arrival times and amplitudes, as the mode declared in the deck selects. It traces rays through the medium and sums Gaussian or hat beams at the receiver. The task covers coherent, incoherent, and semi-coherent pressure, ray tracing, and ASCII or binary arrivals; three-dimensional decks and eigenrays lie outside its scope.

cea. CEA2 is a code for chemical equilibrium and rocket performance from the NASA Glenn Research Center. From an input deck it reads a problem type and a set of reactants and computes the equilibrium composition of the product mixture together with the thermodynamic properties. Equilibrium comes from minimizing the free energy over a thermochemical library, and the assigned-state, detonation, and rocket calculations are evaluated from that solution. The task migrates the five template families tp, hp, tv, detonation, and rocket; transport, mass-fraction output, frozen composition, shock problems, and ion charge lie outside its scope.

disort. DISORT is a solver of the radiative transfer equation by the discrete ordinates method. From a namelist deck that supplies the optical properties, the geometry, the sources, and the surface reflection, it computes the diffuse intensities and the fluxes at every allocated level and angle. The task covers plane and pseudo-spherical geometry; beam, isotropic, and thermal sources; Lambert and four bidirectional surface models; the delta-M and delta-M+ scalings; both IBCND intensity modes; ONLYFL; automatic angles and depths; the TMS and IMS corrections; and Fourier accumulation. Moment generation, printing, and the self-test mode of the reference program lie outside its scope.

mercury6. Mercury6 is an integrator of planetary and small-body systems in celestial mechanics. From a native initial-condition deck it advances a central body, massive bodies, and massless test particles and writes the terminal Cartesian positions and velocities of every body. The integration uses the fixed-step second-order mixed-variable symplectic algorithm with symplectic correctors. The task keeps Newtonian systems without close encounters, collisions, or ejections; relativity, user forces, nonzero zonal harmonics, massive small bodies, and non-Cartesian initial conditions lie outside its scope.

modflow88. MODFLOW-88 is the modular finite-difference model of subsurface water flow from the U.S. Geological Survey. From a name file it reads a grid, the boundary and stress packages, and the time controls, and it writes the native heads, drawdowns, and water budgets for every declared stress period and time step. The flow equation is solved by the strongly implicit procedure or by successive over-relaxation. The task covers BAS with BCF, OC, RCH, WEL, DRN, EVT, GHB, and RIV and the SIP or SOR solver; cases that select another package lie outside its scope.

xfoil. XFOIL is an airfoil analysis program that couples a panel method to an integral boundary-layer model. From a command session and an optional geometry file it computes one viscous operating point, with the Reynolds number, the Mach number, the transition settings, and the angle of attack that the session gives, and it writes polars, surface pressures, and boundary-layer and wake quantities. The task covers one operating point per session with normalization, loaded or built-in geometry, paneling, transition control, and file output; interactive plotting, the design menus, and sessions with more than one operating point lie outside its scope.

C Evaluation Protocol

Environment. The migrations ran on one machine with two AMD EPYC 7H12 processors, and the final evaluation ran on another with two Intel Xeon Platinum 8470 processors; neither uses a GPU. The candidate programs run on the CPU with Python 3.12 and JAX 0.11.0 in 64-bit precision: Python 3.12.3 in the image of the migrations and 3.12.14 in the final evaluation.

Final evaluation. The final evaluation runs after all migrations have ended, on one machine. The candidate is pinned to two physical cores of one socket with their hyperthreads, and the memory limit is 4 GB. A test that exceeds this limit does not count as passed, and a test passes only when its answer is correct within the per-test limit of the task contract (Table 5). Scoring calls the scoring function of the task package: it compares every field of the answer with the output of the reference program under a tolerance band and checks the physical certificates that the task prescribes, for example, the conservation of energy and of angular momentum, and the result of every test falls into one category: passed, numerical disagreement, missing answer, wrong output format, unparsable answer, timeout, or operator fault. We rebuild the reference program on this machine with the procedure of the task package and confirm with the scoring function of the task that its output on the held-out tests agrees with the reference answers. For every test we also record the time from the start of the process to the written answer. No agent receives any result of the held-out tests.

Tolerances of the held-out tests. The task package records the tolerances of every output field in every held-out test. The task packages set every tolerance before the first migration, and the final evaluation used the same packages. A tolerance should admit the numerical differences between correct variants of the reference program. The task package therefore runs several variants of the reference program on the same tests. The variants differ in compiler version, optimization

level, convergence threshold, arithmetic precision, or summation order. Variants compiled with `-ffast-math` serve only as diagnostics and enter no tolerance.

The scoring functions of bellhop, cea, modflow88, and xfoil compare a field of the answer with the reference answer through the normalized distance

$$d = \max_j \frac{|\hat{y}_j - y_j^{\mathrm{ref}}|}{s}. \tag{6}$$

Here $\hat{y}_j$ is the jth value of the field in the answer, and y_j^{ref} is the same value in the reference answer. The scale s is the largest magnitude of the field. The field passes if $d \le \tau$. In modflow88, d and s are computed separately for each layer at each time step. The tolerance is

$$\tau = \left\lceil \max\{2E,\ 10g,\ 50F,\ 100\epsilon\} \right\rceil_2. \tag{7}$$

Here $\lceil \cdot \rceil_2$ rounds upward to two significant digits, and $\epsilon = 2^{-52}$ is the machine epsilon of double precision. The other three terms are distances d between variants. One variant supplies the reference answers, and a second variant changes only the convergence threshold or the arithmetic precision. g is the distance between these two variants. E is the largest distance between any variant and the more accurate variant of the pair. F is the largest distance between variants that differ only in compiler version or optimization level. In cea, the maximum includes a fifth term

$$u = \min\{100\epsilon S/s,\ 10^{-4}\}. \tag{8}$$

Here S is the largest magnitude among the fields of the same unit in the test. A field enters the score only if $\tau \le 10^{-4}$, and the other fields serve only as diagnostics. In modflow88, the storage terms of the water budget and the fields of zeros also serve only as diagnostics.

For disort and mercury6, the tolerances come from the spread of many variants. The disort package runs 13 variants in double precision. The spread of a value is the difference between the largest and smallest outputs of the variants. A value passes if

$$|\hat{y} - y^{\mathrm{ref}}| \le a + r\,|y^{\mathrm{ref}}|, \tag{9}$$

and each quantity of each test has one pair (a, r). Let s be the scale of the quantity, and let ρ be the largest relative spread among values with magnitude at least $10^{-6}s$. Then

$$r = \left\lceil 50 \max\{\rho,\ \epsilon\} \right\rceil_2, \qquad a = \left\lceil 50 \max\{\alpha,\ \epsilon s\} \right\rceil_2, \tag{10}$$

with α the largest spread among the other values. The mercury6 package builds the reference program three times at each of four optimization levels. For each component of position and velocity, the normalized spread e is the spread divided by the scale s of the body. The tolerance of a component is

$$\tau = \left\lceil 50 \max e \right\rceil_2, \tag{11}$$

with the maximum over all bodies in the visible and held-out tests. All tests therefore share the same six tolerances. A value passes if

$$|\hat{y} - y^{\mathrm{ref}}| \le \max\{\tau\,|y^{\mathrm{ref}}|,\ As\}. \tag{12}$$

Here $A = \lceil 50 \max e \rceil_2$, with the maximum over the bodies of the test.

Besides the tolerances, the scoring function checks the structure of the answer and rejects values that are not finite. A held-out test passes only if all scored fields and all required certificates pass. The visible tests follow the same rules. Table 4 lists the comparison and the range of the tolerances for each task.

Time limits and warm-up. The contract of each task sets a time limit for every test and may name a warm-up entry. For the three tasks with a warm-up entry, the harness calls this entry once before the evaluation, and the first compilation counts against the warm-up allowance, not against the per-test limit. For the three tasks without a warm-up entry, the new process of every test must complete its compilation within the per-test limit. Table 5 lists the values, and the per-test limits of the six tasks are therefore not directly comparable. The limit of disort was recalibrated from the cold-start times of one real candidate program, and the other five tasks keep the limit of 60 seconds. These time limits also act as a performance requirement on the delivered program.

Table 4: Comparisons and ranges of the tolerances over the scored fields of the ten held-out tests of each task.

Task	Comparison	Scored fields		Minimum	Maximum
bellhop	$d \le \tau$	24 of 24	τ	2.3×10^{-14}	9.3×10^{-5}
cea	$d \le \tau$	317 of 317	τ	2.3×10^{-14}	6.1×10^{-5}
disort	Equation 9	all	a	1.1×10^{-15}	1.2×10^{-8}
			r	1.2×10^{-14}	6.4×10^{-8}
mercury6	Equation 12	all	τ	4.5×10^{-14}	1.2×10^{-12}
			A	5.8×10^{-15}	1.1×10^{-12}
modflow88	$d \le \tau$ for each layer and time step	111 of 258	τ	1.3×10^{-7}	9.9×10^{-5}
xfoil	$d \le \tau$	221 of 230	τ	2.3×10^{-14}	7.9×10^{-5}

Table 5: Tests and per-test time limits of the six tasks.

Task	Visible tests	Held-out tests	Probes counted	Time limit per test (s)	Warm-up allowance (s)	Warm-up entry
bellhop	10	10	30	60	0	no
cea	10	10	29	60	600	yes
disort	16	10	30	974	974	yes
mercury6	10	10	30	60	60	yes
modflow88	10	10	30	60	0	no
xfoil	10	10	30	60	0	no

Probes. We use 179 probes in total: each task has 30 probes, except for cea, for which we used 29. The probes are stratified by the entries of the input dimension of the rubric, and every entry examined by the probes receives at least three probes. The input of every probe lies within the scope stated by the contract(Appendix D), and for each probe we record the input axis that it varies, the value, and the corresponding sentence of the contract. The probes pass the admission checks of the task package, the procedure of the task package derives their tolerance bands, and the reference program supplies their reference answers. As a self-check, we submit the output of the reference program to the scorer as if it were the answer of a candidate, and all probes must pass.

Dimensions and gate. Every entry of the rubric receives a grade: 1 when satisfied, 0.5 when partly satisfied, and 0 when not satisfied, and an entry that does not apply or has no evidence is excluded. An entry of the output dimension corresponds to a group of fields of the scorer; it receives 1 when all score lines of the group pass, 0 when all of them fail, and 0.5 otherwise. The physical certificates on the property dimension and the entries examined by probes on the input dimension are graded in the same manner from their score lines and their probes. LLM graders assign the remaining entries after reading the delivered program. Each grader reads a copy of the program and never runs the program. Two DeepSeek V4.1 Flash graders grade every entry independently, and a third grader settles each disagreement. The two graders agree on 815 of 882 entries (0.924). Five of these entries on the property dimension concern the program itself: the target stack performs the computation, the program depends on no external computation, the program stores no answers, the program carries no superseded implementation, and the program carries no unreachable code. The reading of a dimension is the mean grade of its applicable entries, and Tables 3, 6, and 7 average the readings over the solvers with equal weights. The entries on external computation and on stored answers form a gate. When either entry is not satisfied, we set the pass rates of the migration to 0. When an evaluation is not completed for a reason on our side, or when any single test of a migration suffers an operator fault, we leave the pass rates and the three dimensions of the migration blank and repeat the evaluation instead of assigning 0.

Table 6: Grades of the LLM graders on the property dimension, one column for each of the five program properties that every task shares and Mean for the reading of the dimension, as mean±std over three trials. Two DeepSeek V4.1 Flash graders grade every entry independently, and a third grader settles each disagreement. The best value of a column is in bold.

Harness	Target stack	No external computation	No stored answers	No superseded code	No unreachable code	Mean
Codex	.92±.08	**1.00**±.00	**1.00**±.00	.44±.13	.22±.10	.72±.03
OpenClaw	.92±.08	**1.00**±.00	**1.00**±.00	.47±.05	.22±.10	.72±.04
LH-Harness	**.97**±.05	**1.00**±.00	**1.00**±.00	.36±.13	.22±.10	.71±.03
Terminus 2	.36±.05	.58±.00	**1.00**±.00	.42±.14	.22±.19	.52±.07
Hermes Agent	.86±.10	**1.00**±.00	**1.00**±.00	.39±.10	.17±.00	.68±.00
OP-Harness (ours)	.92±.08	**1.00**±.00	**1.00**±.00	**.75**±.08	**.44**±.10	**.82**±.04

Table 7: Grades of the LLM graders on the input dimension, one column for each task with the mean grade of the input entries that no probe examines and Mean for the reading of the dimension over the six tasks, as mean±std over three trials. Two DeepSeek V4.1 Flash graders grade every entry independently, and a third grader settles each disagreement. The best value of a column is in bold.

Harness	bellhop	cea	disort	mercury6	modflow88	xfoil	Mean
Codex	.83±.00	.83±.07	.61±.10	.96±.07	.94±.10	.75±.00	.82±.03
OpenClaw	.83±.00	.88±.00	.67±.00	.96±.07	.94±.10	.33±.58	.77±.09
LH-Harness	.78±.10	.88±.00	.72±.10	.96±.07	.94±.10	**.92**±.14	.87±.02
Terminus 2	.61±.10	.54±.14	.50±.00	.83±.07	.89±.19	.58±.14	.66±.08
Hermes Agent	.89±.19	.83±.07	.78±.10	**1.00**±.00	**1.00**±.00	.67±.14	.86±.03
OP-Harness (ours)	**.94**±.10	**.92**±.07	**.83**±.00	**1.00**±.00	**1.00**±.00	**.92**±.14	**.94**±.01

Execution coverage. The object is the set of all Python source files of the delivered program. We exclude by name the files under a directory called test or tests and the files whose names match test_*.py, *_test.py, or conftest.py. Executable lines follow the definition of Python itself: we compile a file into code objects and take the lines that occur in their line tables, which leaves out comments, blank lines, and docstrings. The inputs are all held-out tests and probes of the task, and when the interface of the task prescribes a warm-up entry, the warm-up also runs once. We record the lines that each input executes and keep only the inputs that the program passes, the held-out tests and probes whose answers are correct within the time limit of the task contract, as in Table 2 and Figure 3. For a delivered program with the set of executable lines E, the set of passed inputs I, and the set X_i of lines that input i executes, the execution coverage is $C = |E \cap \bigcup_{i \in I} X_i|/|E|$. A program that passes no input reads 0, and a program without executable lines has no reading; an entry of Table 3 is the mean over the tasks with a reading, reported as mean±std over the three trials. The body of a jitted function executes once, line by line, during tracing, and all branches of a conditional execute during tracing; these lines count as executed. The pass that records lines measures no time.

Readings of Tables 2 and 3. Tables 2 and 3 report four readings of the final evaluation. The pass rate on the probes of a task is k/K. Here K is the number of probes counted for the task. The delivered program passes k of these probes within the time limit that the task contract sets for each test. The cost of a harness is the total number of tokens that the six migrations of one trial consume, excluding cached reads, in millions. The dollar cost in Figure 6 also counts the cached reads at the DeepSeek prices. Per million tokens, the peak prices are $0.006 for cached reads, $0.30 for other input, and $1.20 for output. Outside the peak hours, every price is half as high. OP-Harness reads 3.9 times as many cached tokens as LH-Harness and writes 1.3 times as much output. The cost ratio of the two harnesses is therefore 1.04 in tokens and 1.86 in dollars. The paragraph *Dimensions and gate* defines the readings of the dimensions, and the paragraph *Execution coverage* defines the execution coverage.

The compared harnesses. The five regular harnesses differ in the state they keep between sessions and in the judgment of completion.

Codex. Codex ran in its goal mode. The executing agent declares completion itself. Our migrations set the goal to the task contract. We used Codex CLI 0.154.0.

OpenClaw. OpenClaw keeps the goal as session state and adds to every turn one line that names the objective and asks the agent to advance it; long-term memory is written to Markdown files in the workspace, and compaction changes only the part of the history that the model sees. The model marks the goal complete itself, and no separate judgment exists. We used OpenClaw 2026.9.4.

LH-Harness. LH-Harness assigns three roles: a manager that decomposes the task and routes one step per round without acting, an executor that performs the step with a fresh context, and a read-only auditor that checks the environment, the files, and the tests, and rebuilds the list of constraints of the task, including the constraint not to do more (Ma et al., 2026). The manager may declare completion only after an auditor report that finds every requirement complete, and the rounds are kept in an append-only ledger. Our migrations used Codex as the executor, the same binary as in the Codex row. We used LH-Harness 0.1.7.

Hermes Agent. Hermes Agent accumulates skills of its own during a migration: the agent writes them with a dedicated tool, and the harness stores them and loads them on demand in later sessions. A comparison on equal terms would require the same skill facility in every harness, so we mark its row in Table 2. Every Hermes Agent migration in our evaluation starts from the same prepared home directory, and no state carries over from one migration to another. Between sessions, the goal and the transcript are kept in a database, and memory files enter the system prompt at the start of a session. Hermes Agent ran in its goal mode, in which a judge call reads the goal and the last response of the agent after every turn and returns one of done, blocked, continue, and wait; the judge ran on the same model as the agent. We used Hermes Agent 0.21.3.

Terminus 2. Terminus 2 requires every response of the agent to carry an analysis, a plan, and the commands to run; when the context nears its limit, three auxiliary calls summarize the history, pose questions about it, and answer them, and the summary replaces the history. The agent declares completion itself and must confirm the declaration in the next turn, and no separate judgment exists. We used Terminus 2 as shipped with Harbor 0.23.0.

Sessions of OP-Harness. Every session of OP-Harness, the builder's and the verifier's alike, is one run of Codex CLI 0.154.0 with a fresh context. The driver of the harness reads the two counts from the verifier's report; in the implementation, the verifier marks a report with nothing under Gaps and nothing under Redundant by the commit subject PASS, and the driver ends the migration on that subject, which encodes the two counts and carries no separate judgment. In all OP-Harness migrations of Table 2, none of the sessions compacted its context, and the largest single request used 72 percent of the context window of 996 thousand tokens; every session therefore stayed within one context window.

Reference software. Table 8 lists the reference software of the six rows of Table 1: the pinned version or commit and the license.

Rubric template. Every solver is graded with the same rubric: one shared sheet with a slot for each part that the task fills in. The card below is that sheet. It fixes the form of an entry on each of the three dimensions, the meaning of the three grades and of the fourth value, and marks every solver-dependent slot with angle brackets. The table after the card counts the entries that the six solvers fill in. The card renames surface to dimension, and the header of the CSV file keeps the column name `surface`.

Rubric template (one sheet per solver)

Rubric: `<solver>`

`<summary>`

Table 8: Reference software of the six tasks: the pinned version or commit and the license.

Solver	Version or commit	License
bellhop	`ae1a4772`	GPL-3.0-or-later
cea	May 21, 2004 build	NASA LEW-17687-1
disort	4.0.99	No explicit license text
mercury6	`aee9e0f6`	GPL-3.0
modflow88	2.6	Public domain (US)
xfoil	6.99	GPL-2.0-or-later; plotlib LGPL-2.0-or-later

Rubric template (one sheet per solver) (continued)

The candidate program is one program that migrates this task's legacy Fortran solver to `<target stack>`. The task package fixes the interface, the cases and the required outputs; this sheet fixes how you grade.

Your role and its limits

You judge one candidate program. You are given that program, the task package, and the results of the scorer and of any probes. Read them all, and run what you need. Modify neither the candidate program nor the task package: to run the candidate program, first copy it to a temporary directory outside both. You are not told which harness, model or agent produced the candidate program, and you must not guess; no criterion below depends on it.

The three dimensions

- **Input dimension.** The input forms and options the task interface requires, and how many the candidate program supports.
- **Output dimension.** The families of output fields the task requires, and how much of each family the candidate program computes correctly.
- **Property dimension.** The physical properties the task declares must hold, and the properties the candidate program itself must have.

Properties every candidate program must have

These five are the same for every solver; each is one atom, reported in the CSV under `property`.

- `prop.target_stack`: the numerical core on the default compute path is `<target stack>`; parsing and file I/O may be Python.
- `prop.no_external_compute`: the default compute path does not ship, call or otherwise depend on a reference implementation; it starts no subprocess and loads no dynamic library other than the ones the target stack itself loads, and every result comes from the implementation the candidate program delivers.
- `prop.no_stored_values`: no value is stored, tabulated or keyed by case identifier; every reported result is computed from the case input.
- `prop.no_superseded_code`: no implementation that a later one replaced is still in the candidate program.
- `prop.no_unreachable_code`: no code is unreachable from the entry point; the executed-code fraction from the mechanical check is evidence for this atom, not a grade by itself. Judge this atom and the previous one at the level of definitions: `0` when a function, class or module is superseded or unreachable, `1` when none is, `0.5` only when static reading cannot settle it. A smaller remnant, such as an unused constant, does not lower the grade; list it in the note.

Atoms of this solver

Each row below is one atom; `check` is `mechanical` when the scorer decides it, `probe` when a probe decides it, and `rubric` when you decide it. The `binding` column ties the scorer's own report to the atom: it holds the field patterns of an output atom and the certificate patterns of a property atom, both matched against the scorer's row labels, and the input axis a probe atom varies.

Rubric template (one sheet per solver) (continued)

Input atoms

`<input atoms>`

Output atoms

`<output atoms>`

Property atoms

`<property atoms>`

Grades

Give every atom you decide one of `1`, `0.5`, `0` or `NA`. You decide the atoms whose `check` is `rubric` and the five properties above; the `mechanical` and `probe` atoms are computed by another script from the scorer's and the probes' results, so neither grade them nor transcribe them.

- `1`: the criterion holds on all evidence you examined.
- `0.5`: it holds in part; write which part holds and which does not.
- `0`: it does not hold, or the functionality is missing.
- `NA`: only where the atom's `applies when` column excludes the case at hand; give the reason.

Evidence and output

Every grade needs evidence, and evidence is only one of three kinds: the result of the scorer or a probe (file and line), a code location (path and line number), or a command you ran with its output. A grade without evidence is void and does not count as covered. Write one CSV per candidate program with header `surface,atom_id,applicable,grade,evidence`, one row per atom you graded, that is the `rubric` atoms and the five properties, with `surface` one of `input`, `output` or `property` and `applicable` one of `yes` or `no`. Add a note of at most 40 lines. Do not compute coverage yourself; another script derives it from the grades you file.

Rubric entries of the six solvers. The tables below give the entries of the input, output and property dimensions of each solver, in the wording of the solver's rubric sheet. A solver keeps the shared skeleton of the card above and fills in its own atoms. Each entry states its criterion, and the last column names what decides it: the scorer, a probe or the grader. The five program properties of the template apply to every solver and are not repeated here, so the property rows below are the solver-specific ones. The entries are grouped by solver, so that the three dimensions of one solver can be read together.

Table 10: Rubric entries of the bellhop task, by dimension.

Entry	Criterion	Decided by
Input dimension		
entry point, case directory and answer location	From the root of the candidate program, `python main.py --case CASE_DIR` computes the case: the one BELLHOP environmental deck the case directory holds, named `<stem>.env` and never a file whose basename is exactly `.env`, is read together with that deck's native auxiliary files, the native answer and the named answer are written into CASE_DIR, and the case's input files are left unmodified.	grader

Entry	Criterion (continued)	Decided by
coherent, incoherent and semi-coherent pressure runs	A deck whose run mode selects a pressure calculation is answered by that calculation: the coherent, incoherent or semi-coherent transmission-loss computation the mode names runs at runtime from the deck and its auxiliary data, and the native answer `<stem>.shd` plus the named answer are written into the case directory.	probe
ray tracing runs	A deck whose run mode selects a ray trace is answered by ray tracing: the rays are traced at runtime from the deck, the native ray answer `<stem>.ray` is written with its title, frequency, source and beam counts, surface and bottom depths, `rz` and, for each ray, launch angle, point, top-bounce and bottom-bounce counts and range/depth rows in ray-ordinal then point-ordinal order, and the named answer is written beside it.	probe
arrival runs in the native ASCII and binary formats	A deck whose run mode selects an arrival calculation is answered by that calculation: the arrival structure is computed at runtime and written as the native arrival answer `<stem>.arr` in the format the mode selects, native ASCII or little-endian Fortran sequential binary with four-byte matching record markers, each receiver carrying its arrival count and its ordered records, and the named answer is written beside it.	probe
Cartesian Gaussian and hat beam types	The beam type the deck declares is honored: a run that propagates beams uses the Cartesian Gaussian beam or the Cartesian hat beam the deck selects, rather than one fixed beam model for every deck.	probe
rectilinear and irregular receiver layouts	The receiver layout the deck declares is honored: both the rectilinear layout and the original irregular layout, determined independently from the input mode, are accepted and bind the answer's physical points, so a case that declares the irregular layout is not answered on a rectilinear grid.	probe
native auxiliary case files	The native auxiliary files a case directory carries beside the deck, its bathymetry, attenuation and tabulated sound-speed files, are read and bind the computation, so that a case whose environment comes from such a file is answered from that file rather than from a built-in default.	probe
explicit rejection of out-of-scope decks	A deck outside the declared scope, a three-dimensional deck or a deck whose run mode is outside the pressure, ray and arrival families, eigenrays included, is rejected explicitly with a nonzero exit and no answer files, rather than answered in part or by an unsupported approximation.	grader
one fresh process per case, no separate warmup	Each case is one fresh process with no warmup pass: startup, compilation, computation and serialization all count toward the case's lease of 60 seconds, the value every scored case carries in `visible/SETTINGS.json`, so the candidate program depends on no earlier case and on no warmup invocation, and its cost including compilation fits the lease.	grader
Output dimension		

Entry	Criterion (continued)	Decided by
complex pressure field family	Every `pressure.real` and `pressure.imag` field label a scored pressure case produces is answered: the complex pressure is written at every receiver point of the deck's grid both to the native SHD answer and to the named answer in native physical point order, and the scorer's row for each of the two labels, the largest absolute deviation over that label's points divided by its frozen reference scale, stays inside the label's fixed band.	scorer
ray point coordinate field family	Every `ray.range` and `ray.depth` field label a scored ray case produces is answered: the range and depth of every point of every traced ray are written, in ray-ordinal then point-ordinal order, both to the native RAY answer and to the named answer, and the scorer's row for each of the two labels stays inside the label's fixed band.	scorer
arrival record field family	Every `arrival.amplitude`, `arrival.phase`, `arrival.source_angle`, `arrival.receiver_angle`, `arrival.delay.real` and `arrival.delay.imag` field label a scored arrival case produces is answered: each of those fields of every arrival record is written, in the record order the receiver declares, both to the native ARR answer and to the named answer, and the scorer's row for each of the six labels stays inside the label's fixed band.	scorer
Property dimension		
exact-zero imaginary pressure on incoherent and semi-coherent runs	On a case whose run mode is an incoherent or a semi-coherent pressure calculation, the imaginary component of the complex pressure the candidate program writes is exactly zero at every receiver point, so the certificate row the scorer prints passes: the threshold is the contract's exact-zero test, "the imaginary component must be exactly zero", which the scorer applies as an equality before any band.	scorer

Table 11: Rubric entries of the cea task, by dimension.

Entry	Criterion	Decided by
Input dimension		
Entry point and case-directory input form	From the root of the candidate program, `python run_agent.py --case CASE_DIR` runs on a case directory that holds one deck `cea2.inp`, reads the deck's input paths relative to that directory, and writes a finite UTF-8 `answer.json` of at most 4 MiB into it.	grader
Workspace self-containment	The runtime libraries and the static thermochemical tables the default path reads are in the candidate program, so a case is answered with nothing supplied from outside it; the practice check's own PATH and shared HOME do not establish this.	grader
The five template families	Each of the five template families tp, hp, tv, det and rocket is accepted and answered when the deck's problem section selects it; a family counts as missing when its problem type is refused or answered from another family's path.	probe

Entry	Criterion (continued)	Decided by
Full input grammar of the five families	Within those families the deck's full input grammar is read: reactant records with their amount and assigned-state keys, the mixture-ratio parameters, the requested state-point schedules, and the output section's unit and plot requests, each answered as the deck states it rather than from a default of the program's own.	probe
Species-set selection and the trace threshold	The species the deck selects are the species answered: the deck's `only`, `omit` and `insert` datasets restrict or extend the model, and the input's trace-based species selection sets the threshold at which a species is kept, dropped or restored.	probe
Explicit rejection outside scope	A case the contract puts out of scope (transport, mass-fraction output, frozen composition, shock, ion charge) is rejected explicitly, with a nonzero exit and no answer files, rather than answered in part or by approximation.	grader
Warmup, cache reuse and the case lease	The optional warmup entry `python run_agent.py --warmup` runs on public dev inputs, the measured cases reuse the same compiled implementation and cache, warmup answers are discarded, no hidden input is consumed, and fresh shape-specific work after warmup still finishes inside the 60-second case lease.	grader
Output dimension		
Mole-fraction rows	Every `mole/...` field label a scored case produces is answered, and each field's values stay inside the operator's band for that field: the species columns of each output table, under the table's own printed label, condensed and isomer species included.	scorer
Scalar rows	Every `scalar/...` field label a scored case produces is answered, and each field's values stay inside the operator's band for that field: the state and derived quantities CEA prints for the requested problem, its detonation quantities, and its rocket performance quantities.	scorer
Plot observations	Every `plt/...` field label the deck's requested plot list produces is answered, and each field's values stay inside the operator's band for that field: the plot columns as named observations in plot order, kept separate from the table rows.	scorer
Property dimension		
Fractions finite and inside their domain	Every fraction the answer reports for an output table is finite and lies in [0, 1]; the scorer records `fraction-domain` as failed, and replaces that table's remaining certificates with it, when a value is nonfinite, negative or above 1.	scorer
Table composition sums to one	In every output table the reported fractions, with the trace bound allowed for the species the table's model omits, contain 1 within the operator's interval bounds (the native output quanta plus the scorer's arithmetic padding p), and the scorer prints the test as `sum` with its residual and bound.	scorer
Element balance of the reported composition	In every output table the composition the answer reports carries each element the reactant blend sets in the ratio that blend sets, within the operator's interval bounds; the scorer prints the test per table and per element as `element:H`, `element:C`, `element:N`, `element:O` or `element:AR`, its padding being 100 eps64 of the interval magnitudes, and it prints a channel only for an element the blend carries.	scorer

Table 9: Rubric entries per dimension and solver. The Property column counts the entries that the rubric sheet of that solver adds; the five program properties of the template are shared by every solver and appear in the Shared column.

Solver	Input	Output	Property	Shared program properties	Total
bellhop	9	3	1	5	18
cea	7	3	3	5	18
disort	10	8	5	5	28
mercury6	8	6	4	5	23
modflow88	9	5	2	5	21
xfoil	9	3	9	5	26
Total	52	28	24	30	134

Table 12: Rubric entries of the disort task, by dimension.

Entry	Criterion	Decided by
Input dimension		
single-case entry point and case-directory input form	From the root of the candidate program, `python -B run.py --input-dir CASE_DIR --output-dir OUTPUT_DIR` answers one case: it reads the single complete `input.nml` of that case directory, writes one complete `answer.txt` into the output directory, and returns zero, so a case is answered by this single-case invocation and not by another entry or by a batch driver.	grader
warmup invocation and the persistent scratch cache	The candidate program accepts the warmup invocation `python -B run.py --warmup` on the designated visible warmup shape, and the `DISORT_SCRATCH_DIR` it is given is writable, persists across the warmup and the case processes, and is where the JAX persistent compilation cache is directed, so a case process reuses what the warmup compiled instead of starting from an empty cache.	grader
explicit rejection outside the declared scope	A request the declared scope excludes is refused rather than approximated or partly answered: a malformed, incomplete or reference-rejected deck, GETMOM generation, PRNT printing, the upstream SLFTST self-test, or any other case outside scope ends in a nonzero exit, a concise reason on stderr, and no answer file.	grader
plane and pseudo-spherical geometry	A deck is answered in both geometry modes the scope declares: the plane-parallel default, and the pseudo-spherical mode the deck selects with its `DO_PSEUDO_SPHERE` flag, under the same stream count, quadrature and boundary treatment as the reference.	probe
beam, isotropic and thermal source families	Each of the three source families the scope declares is answered from the deck's own values and not from a built-in default: a beam source at the deck's incidence angle and azimuth (`FBEAM`, `UMU0`, `PHI0`), an isotropic source (`FISOT`), and a thermal source (`PLANK` with `BTEMP`, `TTEMP`, `TEMIS`).	probe
Lambert and the four BRDF surface families	Every surface family the scope declares is answered: a Lambert lower boundary (`LAMBER`, `ALBEDO`), and the Hapke, Cox-Munk, RPV and Ross-Li bidirectional reflectances precomputed from the deck's `BRDF_TYPE`, `BRDF_ARG` and `NMUG`, each used as the lower-boundary condition it declares.	probe

Entry	Criterion (continued)	Decided by
delta-M and the delta-M+ moment-based downgrade	A delta-M+ request is answered with the effective mode the deck's moments select: the request is honored only when `NMOM >= NSTR+1` and every layer satisfies both `PMOM(NSTR,l) >= 1e-4` and `PMOM(NSTR+1,l) >= 0.7*PMOM(NSTR,l)`, otherwise the case runs as ordinary delta-M with the effective DELTAMPLUS reported as F, and a request with insufficient moment capacity is rejected.	probe
the two IBCND modes and the ONLYFL flag	Both problem types the scope declares are answered, the beam-source mode IBCND=0 and the reciprocal-medium mode IBCND=1, whose comparison covers ALBMED and TRNMED for the positive angle half instead of the flux and intensity labels; and the deck's ONLYFL flag is honored, so a flux-only case omits the UU comparison while still writing every allocated slot.	probe
supplied or automatic angles and depths	Both grid forms the scope declares are answered: the user-supplied angles and depths the deck's `USRANG` and `USRTAU` flags select, and the automatic grids the reference derives when a flag is F, with the comparison over the meaningful slots those flags leave.	probe
Fourier accumulation and the ACCUR stopping criterion	Fourier accumulation and the ACCUR stopping criterion are both honored: the azimuthal expansion the deck's `NPHI`, `PHI` and `ACCUR` select is accumulated to the accuracy the deck asks for, with the deck's own ACCUR value rather than a built-in stopping bound.	probe
Output dimension		
direct-beam flux at each level	Every `RFLDIR` label a scored case prints is answered: the direct-beam flux is written at every allocated level, and the scorer's row for that label, the largest scaled deviation over those levels, passes the case's absolute-plus-relative band.	scorer
diffuse downward flux at each level	Every `RFLDN` label a scored case prints is answered: the diffuse downward flux is written at every allocated level, and the scorer's row for that label, the largest scaled deviation over those levels, passes the case's absolute-plus-relative band.	scorer
diffuse upward flux at each level	Every `FLUP` label a scored case prints is answered: the diffuse upward flux is written at every allocated level, and the scorer's row for that label, the largest scaled deviation over those levels, passes the case's absolute-plus-relative band.	scorer
flux divergence at each level	Every `DFDT` label a scored case prints is answered: the flux divergence, the derivative of the net flux with respect to optical depth, is written at every allocated level, and the scorer's row for that label, the largest scaled deviation over those levels, passes the case's absolute-plus-relative band.	scorer
mean intensity at each level	Every `UAVG` label a scored case prints is answered: the mean intensity is written at every allocated level, and the scorer's row for that label, the largest scaled deviation over those levels, passes the case's absolute-plus-relative band.	scorer
intensity at each angle, level and azimuth	Every `UU` label a scored case prints is answered: the intensity is written at every allocated angle, level and azimuth, and the scorer's row for that label, the largest scaled deviation over those components, passes the case's absolute-plus-relative band; a flux-only case compares no UU label, and the slots it leaves ungraded must still be present and finite.	scorer

Entry	Criterion (continued)	Decided by
albedo of the medium per incident angle	Every `ALBMED` label a scored case prints is answered: the albedo of the medium is written for each compared incident beam angle, and the scorer's row for that label, the largest scaled deviation over those angles, passes the case's absolute-plus-relative band.	scorer
transmissivity of the medium per incident angle	Every `TRNMED` label a scored case prints is answered: the transmissivity of the medium is written for each compared incident beam angle, and the scorer's row for that label, the largest scaled deviation over those angles, passes the case's absolute-plus-relative band.	scorer
Property dimension		
net-flux constancy certificate	On every case whose contract carries the net-flux rule, the scorer's `net_constant:flux` certificate passes: the spread of `RFLDIR+RFLDN−FLUP` over the case's levels stays within that case's own threshold, the threshold being the package's rule, "ten times the larger of the reference residual and 100 binary64 epsilons times physical scale".	scorer
reflected-medium albedo and transmissivity conservation certificate	On the case whose contract carries the albedo-transmission rule, the scorer's `albedo_transmission:ALBMED+TRNMED` certificate passes: the largest `abs(ALBMED+(1−ALBEDO)*TRNMED−1)` over the compared incident angles stays within that case's threshold, which the package records as 5.3e-9 on the one case that carries it, and the coupled field bands must pass as well.	scorer
nonnegativity certificate of the reported quantities	On every case, each nonnegativity channel the case contract carries passes: the largest negative value among that quantity's reported values stays within the case's threshold, the larger of its nonnegativity bound and its field absolute tolerance; the channels are the five quantity labels RFLDIR, RFLDN, FLUP, UAVG and UU, and a quantity a contract omits carries no channel.	scorer
flux-divergence relation certificate	On every case whose contract carries the DFDT relation rule, the scorer's `DFDT_zero:DFDT` certificate passes: the largest absolute DFDT value stays within the case's threshold; the rule is report-only, so it is reported and does not gate the case.	scorer
reciprocity operator diagnostic	On the case whose contract carries the reciprocity rule, the scorer's `reciprocity:UU` row is reported; the rule is a diagnostic of the reference program and not a requirement on the candidate program, because a candidate receipt always reports a missing partner and carries no candidate reciprocity value, so the row names no candidate result.	scorer

Table 13: Rubric entries of the mercury6 task, by dimension.

Entry	Criterion	Decided by
Input dimension		
case-directory entry point	The candidate program computes a case when its root is invoked as `python −B run.py −−case CASE_DIR`, taking the deck from that case directory and writing the answer files into it.	grader

Entry	Criterion (continued)	Decided by
warmup invocation and shared cache directory	The candidate program accepts the warmup invocation `python -B run.py --case CASE_DIR --warmup` on a copy of a supplied case, and takes the location of its writable compilation cache from MERCURY6_SCRATCH_DIR, so that the warmup and the later case processes share one cache.	grader
MVS algorithm selected by the deck	A deck whose algorithm is MVS is integrated by the fixed-step second-order mixed-variable symplectic algorithm with the reference symplectic correctors, not by another integrator and not with another splitting.	grader
explicit rejection of out-of-scope decks	A deck outside the declared scope is rejected explicitly with a nonzero exit and no answer files, rather than answered by an unsupported or approximate path.	grader
body population	A case with one positive-mass central body, 1 to 8 positive-mass large bodies and 0 to 200 massless small bodies is accepted, and every body of the case appears in the answer files.	probe
central mass and body masses	The central mass and the mass of every body are the ones the deck declares, with the central body and the large bodies of positive mass and the small bodies massless.	probe
Cartesian states and epoch	The initial state of every body is the deck's Cartesian position and velocity at the declared epoch, read in the deck's units of AU, AU per day and solar masses, with the body epochs synchronized to the start time.	probe
fixed-step schedule and terminal epoch	The start time, stop time and timestep declared in param.in drive a fixed-step integration that stops at exactly the declared terminal epoch, which the big.dmp header carries.	probe
Output dimension		
terminal x position	The terminal x position of every body is written in the native state records and passes the scorer's x field row, the largest scaled deviation over the bodies of the case.	scorer
terminal y position	The terminal y position of every body is written in the native state records and passes the scorer's y field row, the largest scaled deviation over the bodies of the case.	scorer
terminal z position	The terminal z position of every body is written in the native state records and passes the scorer's z field row, the largest scaled deviation over the bodies of the case.	scorer
terminal x velocity	The terminal x velocity of every body is written in the native state records and passes the scorer's vx field row, the largest scaled deviation over the bodies of the case.	scorer
terminal y velocity	The terminal y velocity of every body is written in the native state records and passes the scorer's vy field row, the largest scaled deviation over the bodies of the case.	scorer
terminal z velocity	The terminal z velocity of every body is written in the native state records and passes the scorer's vz field row, the largest scaled deviation over the bodies of the case.	scorer
Property dimension		
energy residual certificate	The energy certificate the scorer computes from the input masses and the candidate states passes: "the energy certificate requires abs(E_t-E_0)/abs(E_0) <= 4.7e-13", a zero numerator and denominator gives zero, and a nonzero numerator over zero fails.	scorer

Entry	Criterion (continued)	Decided by
angular-momentum vector certificate	The angular-vector certificate the scorer computes from the input masses and the candidate states passes: "the angular-vector certificate requires norm(L_t-L_0)/norm(L_0) <= 6.9e-13", under the same zero-denominator rule.	scorer
semimajor-axis certificate	On a designated single-planet case the semimajor-axis certificate passes: "Designated single-planet cases also require abs(a_t/a_0-1) <= 2.3e-13".	scorer
orbital phase and analytic period certificate	On a designated single-planet case the phase-period certificate passes: "abs((2*pi*T/delta_M)/P_analytic-1) <= 2.3e-13", and a zero or nonfinite phase advance fails.	scorer

Table 14: Rubric entries of the modflow88 task, by dimension.

Entry	Criterion	Decided by
Input dimension		
Entry point, case directory and output directory	From the root of the candidate program, `python main.py --input-dir CASE_DIR --output-dir OUTPUT_DIR` computes a case: CASE_DIR holds one MODFLOW-88 name file and its fixed-width package files, DOS path tokens in the NAM are resolved by their uppercase basename, the inputs are read from CASE_DIR (which is left unmodified), and the NAM-declared listing and native answer files are written into the fresh OUTPUT_DIR.	grader
Package families the NAM file selects	The packages a case's NAM file selects among the in-scope families BCF, OC, RCH, WEL, DRN, EVT, GHB and RIV are read and honored: each selected package's control records are parsed and its terms enter the solution, so that a package the development cases happen not to combine is still supported and no fixed package set is assumed.	probe
Solver package selection (SIP or SOR)	A case whose NAM file selects SIP or SOR is answered: the selected solver package and its control records are read, and no particular iteration count or algorithm is required, so a candidate program may use an iteration of its own but must accept either selection.	probe
BAS grid, IBOUND, starting heads and the stress-period schedule	The discretization and schedule the case's BAS package declares drive the computation: the layer, row and column counts, the IBOUND statuses (active, inactive, constant head), the starting heads, and every stress period and time step with its length, its step count and its steady or transient flag are read from the case, so that a fixed grid or a single steady period is not assumed.	probe
BCF hydraulic properties	The hydraulic properties the case's BCF package declares are read and drive the flow solution: the layer types and the transmissivity, leakance and storage arrays, with the dry-cell and wetting controls, are taken from the case rather than fixed or ignored.	probe
Boundary and source package data	The RCH, WEL, DRN, EVT, GHB and RIV package data a case declares are read and enter the flow solution as the case declares them: recharge and evapotranspiration rates, well rates, drain and river conductances and stages, and general-head boundary conductances and heads, per stress period and layer as the package files give them.	probe

Entry	Criterion (continued)	Decided by
NAM-declared listing and native answer files	The listing and native answer files are the ones the case's NAM declares, not names of the candidate program's own choosing: the LIST token and the DATA unit tokens the NAM carries fix the file names written into OUTPUT_DIR, and the supplied cases declare `model.lst`, `head.hds`, `drawdown.drw` and `budget.cbc`.	probe
Explicit rejection outside scope	A case whose NAM file selects a package outside the in-scope families, STR for example, is rejected explicitly with a nonzero exit and no answer files, rather than answered in part or by approximation.	grader
Per-case lease with no separate warmup	Each case is one fresh process with no separate warmup pass: startup, compilation, solution and output all count toward the case lease, so a candidate program must not depend on a warmup invocation or on state left by an earlier case, and its cost including compilation has to fit the lease.	grader
Output dimension		
Saved native head arrays	Every `native:head` field label a scored case produces is answered: the float32 heads saved at each scheduled period, step and layer the case's output control saves are written to the NAM-declared head file in the native record layout, and the field's maximum scaled deviation from the reference stays inside its band.	scorer
Saved native drawdown arrays	Every `native:drawdown` field label a scored case produces is answered: the float32 drawdowns saved with the heads are written to the NAM-declared drawdown file in the native record layout and the field's maximum scaled deviation stays inside its band, the drawdown scale being the parent-head scale at the same stress period, step and layer.	scorer
Saved native cell-by-cell budget terms	Every `native:budget:<package>` field label a scored case produces is answered: the float32 cell-by-cell terms of each registered budget package, the constant-head, storage, internal face and package terms the case's budget file registers, are written to the NAM-declared budget file with the required record headers, and each such field passes its band.	scorer
Listing budget terms, current rate	Every `listing:rate:<package>:<in|out>` field label a scored case produces is answered: each scheduled listing budget carries its separate inflow and outflow package terms at the current step's rate, a missing package term being a format failure, and each such field passes its band.	scorer
Listing budget terms, cumulative volume	Every `listing:volume:<package>:<in|out>` field label a scored case produces is answered: the same package terms are reported as cumulative volume from the simulation start, storage release counting as inflow and accumulation as outflow, and each such field passes its band.	scorer
Property dimension		
Model-wide balance certificate	For each scheduled listing budget the parsed package totals I and O satisfy nonnegative totals and abs(I-O)/max(I,O) <= 9.0e-4 for rate and <= 8.8e-4 for cumulative volume, a double-zero row passing and a single zero total giving one; the scorer prints the test as `model-wide-budget`, one row per channel, `rate` and `volume`.	scorer

Entry	Criterion (continued)	Decided by
Head/drawdown identity certificate	At every saved native cell whose frozen output-time IBOUND is nonzero, constant-head cells included, abs(h+d-S)/L_c <= 9.7e-7, where h and d are the candidate float32 head and drawdown, S the original float32 starting STRT and L_c the fixed positive scale; the scorer prints the test as `head-drawdown-identity`, channel `identity`.	scorer

Table 15: Rubric entries of the xfoil task, by dimension.

Entry	Criterion	Decided by
Input dimension		
entry point and case directory	The candidate program computes a case when its root is invoked as `run_xfoil --case CASE_DIR`, taking the one session from that case directory and writing the six answer files back into it.	grader
session control and a single operating point	The analysis is driven by the session's own control flow through `PLOP`, `G` and `QUIT`, at the single operating point the session requests rather than by a fixed computation run on every case.	probe
geometry source and normalisation	The airfoil geometry is the one the session declares, either a coordinate file read by `LOAD` or a built-in `NACA` name, and it is normalised when the session asks for `NORM`.	probe
panelling settings	Panelling follows the session's `PPAR` settings and `PANE` repanelling, so the answer carries the surface rows of the session's own panel count rather than of a fixed mesh.	probe
OPER settings	The `OPER` settings the session gives drive the analysis: the iteration limit, the Reynolds number and the free-stream Mach number are the session's values, and the answer's Mach and Reynolds headers report them within the comparison bands.	probe
transition control	Transition follows the session's `VPAR` and `XTR` settings, by the amplification criterion at the session's Ncrit where the session leaves it free and forced at the trip position where the session fixes it, and the answer reports both transition positions within the chord.	probe
one angle of attack	The single angle of attack the session sets by `ALFA` is the operating point the answer reports, within the comparison band.	probe
session output control	The session's own output control, `PACC`, `CPWR` and `DUMP`, selects the three native answers and their `.named` companions written into the case directory.	probe
explicit rejection of out-of-scope cases	A case outside the declared scope is rejected explicitly with a nonzero exit and no answer files, rather than answered by an unsupported or approximate path.	grader
Output dimension		
polar operating-point record	Every `polar.` field label a scored case produces is answered: the candidate program writes one native polar record for the session's single operating point and a `.named` companion holding the same operating point at higher precision, and each of the nine labels' largest scaled deviation over the case stays inside that label's band.	scorer

Entry	Criterion (continued)	Decided by
surface pressure rows	Every `cp.` field label a scored case produces is answered: the candidate program writes one pressure coefficient per surface point, in the native order and at the same points as the boundary-layer surface rows, together with a `.named` companion, and both labels' largest scaled deviations over the points stay inside their bands.	scorer
boundary-layer surface and wake rows	Every `bl.` field label a scored case produces is answered: the candidate program writes the native boundary-layer surface rows and the wake rows of the case's own wake count, with the four surface-only quantities present, together with a `.named` companion, and each of the twelve labels' largest scaled deviations over its rows stays inside that label's band.	scorer
Property dimension		
lift coefficient from the surface pressure integral	The CL certificate passes on both printed channels: the lift coefficient a case reports agrees with the normal-force integral of the answer's own surface pressure row over the panel geometry, that is the mean of adjacent pressure coefficients times the panel's chordwise projection after the reported-alpha rotation, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-4` for CL, `1e-5` for Cp and `1e-5` for the coordinates.	scorer
pressure drag coefficient from the surface pressure integral	The CDp certificate passes on both printed channels: the pressure drag coefficient a case reports agrees with minus the normal-projection integral of the answer's own surface pressure row, the mean of adjacent pressure coefficients times the panel's projection normal to the chord after the reported-alpha rotation, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-5` for CD and CDp, `1e-5` for Cp and `1e-5` for the coordinates.	scorer
quarter-chord pitching moment	The CM certificate passes on both printed channels: the pitching moment a case reports agrees with the closed-contour by-parts integral about the quarter chord over the answer's own surface geometry, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-4` for CM, `1e-5` for Cp and `1e-5` for the coordinates.	scorer
total drag from the wake momentum thickness	The CD_SquireYoung certificate passes on both printed channels: the total drag coefficient a case reports agrees with the Squire-Young wake relation `2*Theta*Ue**((5+Dstar/Theta)/2)` evaluated on the answer's own last wake row, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-5` for CD, `1e-6` for Dstar and Theta and `1e-5` for the edge velocity; the wake row lies inside the certificate domain the reader requires before any comparison.	scorer
surface pressure against the edge velocity	The Cp_Ue certificate passes on both printed channels: every surface pressure coefficient agrees with the compressibility transformation of the absolute edge velocity `|Ue/Vinf|` at the same point under the case's Mach number, the Karman-Tsien relation the reference uses, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-5` for Cp and `1e-5` for the edge velocity.	scorer

Entry	Criterion (continued)	Decided by
kinematic shape factor	The Hk_shape certificate passes on both printed channels: on every row the scorer's mask selects, the shape factor a case reports agrees with `(Dstar/Theta - 0.29*Ms)/(1 + 0.113*Ms)` for the local edge Mach number, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-4` for Hk and Hstar, `1e-6` for Dstar and Theta and `1e-5` for the edge velocity.	scorer
P from momentum thickness and edge velocity	The P_ThetaU2 certificate passes on both printed channels: on the surface rows the quantity P a case reports agrees with `Theta*(Ue/Vinf)**2` of the same row, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-5` for P and the edge velocity and `1e-6` for Theta.	scorer
m from displacement thickness and edge velocity	The m_DstarU certificate passes on both printed channels: on the surface rows the quantity m a case reports agrees with `Dstar*(Ue/Vinf)` of the same row, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-5` for m and the edge velocity and `1e-6` for Dstar.	scorer
K from the kinetic shape factor, momentum thickness and edge velocity	The K_HstarThetaU3 certificate passes on both printed channels: on the surface rows the quantity K a case reports agrees with `Hstar*Theta*(Ue/Vinf)**3` of the same row, to within the print-error bound built from the answer's own printed quanta, whose caps are `1e-5` for K and `1e-4` for Hstar.	scorer

Probe strata. Every task is examined by thirty probes, stratified by the entries of its input dimension, except for cea, where one probe was excluded and twenty-nine remain. The table below lists, for each solver, the stratum each probe belongs to and how many probes it holds, with the solver's total in the last column. The table gives no probe input, no reference value and no tolerance.

Table 16: Probes of each task by stratum of the input dimension.

Solver	Stratum of the input dimension	Probes	Total
bellhop	coherent, incoherent and semi-coherent pressure runs	5	30
	ray tracing runs	5	
	arrival runs in the native ASCII and binary formats	5	
	Cartesian Gaussian and hat beam types	5	
	rectilinear and irregular receiver layouts	5	
	native auxiliary case files	5	
cea	The five template families	10	29
	Full input grammar of the five families	9	
	Species-set selection and the trace threshold	10	
disort	plane and pseudo-spherical geometry	4	30
	beam, isotropic and thermal source families	5	
	Lambert and the four BRDF surface families	5	
	delta-M and the delta-M+ moment-based downgrade	4	
	the two IBCND modes and the ONLYFL flag	4	
	supplied or automatic angles and depths	4	
	Fourier accumulation and the ACCUR stopping criterion	4	
mercury6	body population	8	30
	central mass and body masses	8	
	Cartesian states and epoch	7	
	fixed-step schedule and terminal epoch	7	

Solver	Stratum of the input dimension	Probes	Total
modflow88	Package families the NAM file selects	5	30
	Solver package selection (SIP or SOR)	5	
	BAS grid, IBOUND, starting heads and the stress-period schedule	5	
	BCF hydraulic properties	5	
	Boundary and source package data	5	
	NAM-declared listing and native answer files	5	
xfoil	session control and a single operating point	4	30
	geometry source and normalisation	5	
	panelling settings	4	
	OPER settings	5	
	transition control	4	
	one angle of attack	4	
	session output control	4	

Task contract template and task descriptions. The section on completion applies to the regular harnesses, whose agent declares completion itself; the contract inserted into the prompts of OP-Harness carries no such section, since only the verifier ends a migration. Each migration is handed to the builder as a task contract: a fixed frame, reproduced below, in which four sections carry the task's own text: TASK, SCOPE, REQUIREMENTS, and ACCEPTANCE. The card shows these four sections as one inner card, as the role instructions of Appendix D.2 do. The table after the card gives, for each solver, the paragraph that the *TASK* section receives, which states the object of the migration and its scope.

Task contract template (one frame per solver)

You work in `./work` on the task described below. `./work` is a git repository, and your last commit is what is judged: work that is not committed is not part of what you deliver.

How you work is yours: read, run and instrument what the task gives you; write throwaway code; rewrite a part. None of it is required and none of it is graded. Your own checks are diagnostics, never passes.

PATHS

Path	What
`./task/`	the task materials: {{`TASK_MATERIALS`}}
`./task/check_visible.sh`	the practice check for the shipped cases: per case and field, pass or fail and the distance to the band. It covers the shipped cases only.
`./work/`	a git repository: your working tree; your last commit is what is judged
`./scratch/`	a place outside `./work` for the small tools and sketch you make as you work

TASK, SCOPE, REQUIREMENTS, ACCEPTANCE

...

COMPLETION

When you consider the candidate program complete, make one more commit: an empty commit whose subject is exactly

`PASS`

and whose second line states in one line what you consider delivered. Do not change any file in it, so the candidate program you deliver stays the program you built. The command is

```
git commit --allow-empty -m PASS -m "<one line:  what you
consider delivered>"
```

Table 17: Task description of each solver, the text the task contract inserts at *TASK*.

Solver	Task description
bellhop	The purpose of this migration is to move BELLHOP's in-scope acoustic propagation into JAX so the solver can use the CUDA execution stack through JAX. Port BELLHOP from ./task/reference_source into Python + JAX in ./work: a complete, standalone program, faithful to the original solver, that reads the deck's `<stem>.env` and writes the one native answer its run mode determines, plus answer.json and answer.npz.
cea	The purpose of this migration is to move CEA2's in-scope numerical computations into JAX so the solver can use the CUDA execution stack through JAX. Port CEA2 from ./task/reference_source into Python + JAX in ./work: a complete, standalone program, faithful to the original solver, that reads cea2.inp and writes answer.json for every problem type in scope.
disort	The purpose of this migration is to move DISORT-scope plane-parallel discrete-ordinate radiative transfer into JAX so batches of optical profiles and spectral bands can use the CUDA execution stack through JAX, retaining supported pseudo-spherical beam geometry. Port DISORT 4.0.99 from ./task/reference_source into Python + JAX in ./work: a complete, standalone program, faithful to the reference solver, that reads input.nml and writes the full allocated flux and intensity output for one call.
mercury6	The purpose of this migration is to move Mercury6's in-scope N-body integration into JAX so batched test particles and groups of initial conditions can use the CUDA execution stack through JAX. Port Mercury6 from ./task/reference_source into Python + JAX in ./work: a complete, standalone program, faithful to the original solver, that reads the native deck the case declares and writes the native big.dmp and small.dmp final-state files.
modflow88	The purpose of this migration is to move MODFLOW-88's in-scope groundwater flow computation into JAX so the solver can use the CUDA execution stack through JAX. Port MODFLOW-88 from ./task/reference_source into Python + JAX in ./work: a complete, standalone program, faithful to the original solver, that reads the NAM file the case declares and writes the native answers and listing it declares, for every stress period and time step in scope.
xfoil	The purpose of this migration is to move XFOIL's in-scope viscous-inviscid airfoil analysis into JAX so the solver can use the CUDA execution stack through JAX. Port XFOIL from ./task/reference_source into Python + JAX in ./work: a complete, standalone program, faithful to the original solver, that reads input.session and its optional airfoil geometry file and writes polar.txt, cp.txt, bl.txt and their .named companions in that case directory.

D System Prompts of the Builder and the Verifier

OP-Harness gives the builder and the verifier one system prompt each, the same for every task, with the task contract at the inner card. From the second iteration on, the opening of the builder also points to the verification reports. The cards rename actor to builder, critic to verifier, round to iteration, and run to migration. The cards also rename deliverable or tree to candidate program, the task's own checks to visible tests, and this contract to task contract, in the commands as well.

The prompts also set three rules. First, the plan opens with the requirement that most decides correctness, and a prerequisite comes just before the item that needs the prerequisite. Then, each finished item is a commit that records the plan for later sessions. Finally, the verifier reads the same materials as the builder, judges the whole scope, and adds checks where the visible tests fall short.

D.1 Openings of the Builder

Builder: opening of the first iteration

You work in ./work on the task described below. git log is where you record what you do: commit as you go, and your last commit must contain the candidate program, including all implementation the task contract asks for before you exit.

Builder: opening of a later iteration

You work in ./work on the task described below, continuing what earlier commits have built. git log is where you record what you do, and where the work so far is recorded: your own commits carry the checklist as it stood at each step. Your last commit must contain the candidate program, including all implementation the task contract asks for before you exit.

The verification

Where: commits by the author `verifier`, newest first.

What: a reading of the candidate program before it, made from the same materials and the same history you have, by a reader who can run and instrument whatever you can run and instrument.

How to treat it: investigate first and judge for yourself; then weigh each thing it names against the current code, take what holds, set aside what does not, and say which. Read as far back as you see fit.

D.2 Builder Instructions

Builder

What is on disk

`./task/`	the task materials
`./work/`	a git repository: your working tree and the record so far
`./scratch/`	a place outside ./work for the small tools and sketch you make as you work

Occam's Progressive Strategy

1. Begin by deciding everything the task contract asks for that ./work does not yet have, and order it.
2. At each step, build the one thing that most decides whether what you deliver is right.
3. Build it completely: nothing less, because a part left open is not done; nothing more, because what the task contract does not ask for is not progress.
4. Something you must have before that thing can be started comes first, and only then.
5. Scaffolding and temporary paths that stand in for the work itself buy you a way to run, not a step toward the delivered program.
6. Consider the task's whole scope, not only what the visible tests exercise: they can all pass while the work is unfinished.

Write the list down before you build: an empty commit, subject `Plan`, body the checklist, one item per line, each marked `[ ]`, and one line on why the first item is first.

How you work is yours: read, run and instrument what the task gives you; write throwaway code; rewrite a part. None of it is required and none of it is graded. What is graded is your last commit.

The work

Work down the list. Each item you finish is one commit: subject `n/N <item>`, body the whole checklist as it now stands, the item marked `[x]`, with an optional note under it. An item you decide not to build is marked `[-]` with the reason on the line under it; an item you

Builder (continued)

discover is appended with `[ ]`; an item half done may be committed with `[ ]` and a note, so that nothing is lost if you stop. The newest commit therefore always carries the current state of the list.

```
2/4 <item two>
[x] 1. <item one>
       note: <optional>
[x] 2. <item two>
[ ] 3. <item three>
[-] 4. <item four>
       note: dropped, <reason>
Why 1 first: <one line>
```

Run your local checks before each commit; they are diagnostics, never passes. Your last commit before you exit is what is judged.

Task contract

...

The delivered program is judged once on the held-out tests, which no one has seen.

D.3 Verifier Instructions

Verifier

You are the verifier in a loop of iterations with a builder. The harness wakes you with a fresh context when the builder has exited; you judge the candidate program at its last commit, reply with a commit of your own, and exit. You have exactly what the builder has: the task materials and the git history. You do, in a fresh context, what a careful developer does before calling the work done: read the program, run it, check it against the task; then write the three parts below.

What is on disk

`./task/`	the task materials, identical to the builder's
`./work/`	the git repository; HEAD is the builder's last commit of the iteration
`./scratch/`	a place outside ./work for the small tools and sketch you make as you judge

The iteration's commits run from the builder's `Plan` to its last commit, whose message carries the checklist as it stood when it stopped: `[x]` done, `[ ]` open, `[-]` dropped; open items at exit mean the builder stopped before finishing. The earlier iterations are behind them. How far back you read, and how closely, is your call.

Your reply

Your judgment is one commit that changes no file. To end the migration:

```
git commit --allow-empty --author="verifier <verifier@ss>"
    -m PASS -m "<body>"
```

Otherwise the standing line takes the place of PASS:

```
git commit --allow-empty --author="verifier <verifier@ss>"
    -m "<standing line>" -m "<body>"
```

The subject is the word `PASS` and nothing else, or the standing of the commit in one line, in the form given under Standing line. `PASS` followed by anything else on the same line does not end the migration. Body: the labels `Facts:`, `Redundant:`, `Gaps:`, `Ran:`, each on its own line followed by its content; `Ran:` is what you ran. Write nothing into the candidate program; the builder decides how to build.

Verifier (continued)

Occam's Sufficiency Verification

Redundant and Gaps are the two directions of one question: whether what is there is no more than the task asks for, and no less. Facts are what you observed; they settle neither.

1. **Facts.** {{FACTS}} Changes against the previous iteration, including tests that stopped passing, are diagnostics: report them; they decide nothing. Facts are observations of this commit. Keep them apart from the two parts below.
2. **Redundant.** Picture a candidate program that would satisfy the task contract as you now understand it. Against that picture, name what is there and not needed to complete the task: a structure nothing requires, a duplicate path, a special case carved out for one input, a part nothing else depends on. For each: the basis (the task's scope, its materials, the mathematics, or a check you ran), why the task's whole scope is still covered without it, and what is uncertain. If you find nothing, say so.
3. **Gaps.** What the task asks for and this commit does not have, each backed by a requirement or by evidence: a part not built; one built but not reachable from the official output; one built wrongly; or a question you cannot settle yet, with the check that would settle it. A list of wishes is not this section.

Your picture of a candidate program that satisfies the task contract is a sketch you revise in every iteration, not a program and not evidence; the builder may argue with it.

PASS

PASS means the candidate program satisfies the task contract as far as you can tell: the visible tests, where the task has any, all pass; what passes is faithful as the task defines it; and both Redundant and Gaps are empty. Only you write it, and it ends the migration. After the migration, the builder's last commit is judged once on the held-out tests, which neither of you has seen; PASS does not consider them, and your picture of a candidate program that satisfies the task contract covers the task's scope, not only what the visible tests exercise.

Task contract

...