OpenSeesAgentBench: A Benchmark and Evaluation Framework for Agentic Structural Analysis in OpenSeesPy

Takayuki Shinohara [1]

Abstract

Large language model (LLM) agents are being applied to structural analysis, but evaluation is often reduced to executability and can blur benchmark construction with agent capability. We present *OpenSeesAgentBench*, an OpenSeesPy benchmark and executable reference evaluation stack spanning *OpenSeesQuery* (90 questions), *OpenSeesCodeBench* (24 code tasks), and *OpenSeesWorkflowBench* via the *OpenSeesBuildingBench calibration profile* (900 workflow cases). The workflow layer uses contract-first evaluation with strict `CaseSpec` validation, fail-closed checks, and bounded repair. In the current paper we report only the reference stack, so the workflow results should be read as a calibration study rather than a head-to-head systems comparison: observed shard expected-match rates of 89.33%, 70.33%, and 60.67% stay within one case of the profile targets. These results show that the released workflow profile preserves its intended difficulty under executable expansion and provides an auditable benchmark for future comparative evaluation; they do not, by themselves, establish superiority of a particular agent architecture.

1. Introduction

Earthquake engineering and seismic simulation rely on workflows that must translate scientific intent into valid structural models, ground-motion inputs, numerical integration settings, recorder configurations, and postprocess artifacts. In practice, even small errors in model topology, damping, time stepping, or output specification can make a run scientifically useless. Success is therefore not simply obtaining one executable script, but producing an auditable analysis whose assumptions, numerical behavior, and deliverables remain consistent with the intended engineering question.

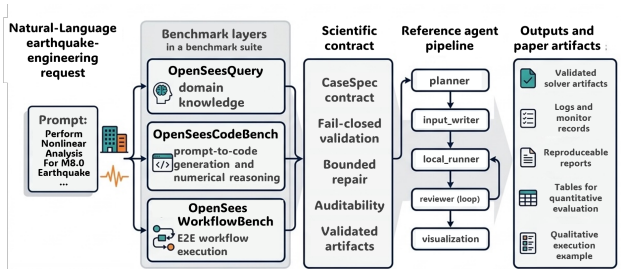


Figure 1. **Current release overview of OpenSeesAgentBench.** The benchmark is organized into three layers: *OpenSeesQuery* for domain knowledge, *OpenSeesCodeBench* for prompt-driven OpenSeesPy generation and numerical checking, and *OpenSeesWorkflowBench* for contract-based end-to-end execution. In the present paper, the workflow layer is instantiated by *OpenSeesBuildingBench calibration profile*, a 900-case profile with easy, medium, and hard shards, and audited using the released reference execution stack for planning, generation, execution, validation, and bounded repair.

Large language models (LLMs) perform strongly on coding and interactive reasoning, and recent work shows that agentic structural analysis is feasible. MASSE models structural engineering as a coordinated multi-agent process, lightweight OpenSeesPy agents improve reliability via role decomposition, FeaGPT extends automation to FEA workflows, and Liu et al. improve beam-analysis reliability by reframing the task as executable OpenSeesPy generation (Liang et al., 2025b; Geng et al., 2025; Qi et al., 2025; Liu et al., 2025). However, current evaluation remains largely system-centric: domain knowledge, code generation, workflow control, generalization, and repair are often collapsed into a single outcome, making it difficult to identify what failed and why. We therefore argue that the missing evaluation unit is the *scientific contract*: the agent must produce the intended model, satisfy numerical and recorder constraints, surface failures under fail-closed checks, repair them within bounded budgets, and leave behind artifacts that another researcher can audit.

We address this gap with *OpenSeesAgentBench* (Figure 1),

[1]National Institute of Advanced Industrial Science and Technology, Tsukuba, Japan. Correspondence to: Takayuki Shinohara <shinohara.takayuki@aist.go.jp>.

Proceedings of the 43^{rd} International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026.

an *implemented* OpenSeesPy-centered benchmark and reference evaluation stack. This paper intentionally focuses on the current executable release rather than a broader multi-solver orchestration vision so that the evaluation contract remains concrete, reproducible, and auditable. The released suite spans three complementary competencies: domain knowledge through *OpenSeesQuery*, prompt-driven code generation and numerical reasoning through *OpenSeesCodeBench*, and end-to-end workflow execution through *OpenSeesWorkflowBench*. In the present study, however, the new quantitative evidence is concentrated in the workflow layer, instantiated by *OpenSeesBuildingBench calibration profile*, a 900-case profile sharded into easy, medium, and hard subsets with target expected-match rates of 89.67%, 70.33%, and 60.67%. To support reproducible use of this evaluation contract, we release a reference multi-agent execution stack in `src/opensees_agent/`: `planner` → `input_writer` → `local_runner` → `reviewer(loop)` → `visualization`. The system uses a strict Pydantic `CaseSpec` contract, fail-closed validation, preflight checks, and bounded three-stage repair.

This paper makes three contributions. First, we distill a design agenda for evaluating AI-scientist-style systems in simulation sciences, centered on contract-first tasks, capability separation, fail-closed validation, bounded repair, difficulty calibration, and auditable outputs (Section 2). Second, we instantiate this agenda in a released OpenSeesPy benchmark that jointly defines domain-knowledge, code-generation, and end-to-end workflow evaluation under explicit case contracts. Third, we report the first workflow calibration study on *OpenSeesBuildingBench calibration profile*: the released reference stack produces shard-level expected-match rates of 89.33%, 70.33%, and 60.67%, staying within one case of the calibration targets overall. We interpret this result as evidence that the executable workflow profile preserves its intended shard-level difficulty and auditing contract. Because the present draft reports only the full reference stack, it should be read as benchmark validation and reference-stack release rather than as a final ranking of orchestration strategies.

2. Related Work

LLMs in Structural and Earthquake Engineering Large language models are being explored for structural design support, code consultation, numerical-analysis assistance, and building-model authoring (Xie et al., 2025). Prior work shows that natural-language interfaces can aid structural optimization, that prompting and in-context learning can improve structural-analysis workflows, and that multi-agent systems can support code-compliant design and engineering decisions (Qin et al., 2024; Liang et al., 2025a; Avila et al., 2025; Liu et al., 2025; Geng et al., 2025; Chen & Bao, 2025; Du et al., 2025; Deng et al., 2025; Youwai et al., 2025). Liu et al. further show that replacing free-form answers with executable OpenSeesPy code improves reliability on a beam-analysis benchmark (Liu et al., 2025), while retrieval-augmented and multimodal systems have also been applied to safety and code consultation (Fan et al., 2024; Adil et al., 2025; Joffe et al., 2025; Aqib et al., 2025). These studies demonstrate utility across subtasks, but they do not provide a reusable benchmark for agentic structural analysis.

Agentic OpenSeesPy and Finite Element Workflows The closest line of work applies agents directly to structural or finite element simulation. MASSE formulates structural engineering as a coordinated multi-agent workflow (Liang et al., 2025b); a lightweight 2D-frame system shows that decomposed OpenSeesPy modeling can outperform monolithic prompting (Geng et al., 2025); Liu et al. combine code generation and execution for more reliable beam analysis, but on a narrow eight-problem benchmark (Liu et al., 2025); and FeaGPT extends the idea to broader FEA automation (Qi et al., 2025). Together these works show feasibility, but they remain largely *system-centric* and use heterogeneous evaluation protocols. They also do not cleanly separate knowledge, code, and workflow capabilities, or enforce fail-closed diagnosis and bounded repair under a shared contract (Han et al., 2026). OpenSeesAgentBench is intended to evaluate exactly these dimensions within one OpenSeesPy-centered benchmark.

Multi-Agent Workflow Systems and Scientific-Agent Infrastructure More broadly, multi-agent systems such as ChatDev, HyperAgent, MetaGPT, Magentic-One, and AutoGen show the value of role specialization for long-horizon tool use (Qian et al., 2023; Phan et al., 2024; Hong et al., 2023; Fourney et al., 2024; Wu et al., 2023). ReAct, Tree-of-Thoughts, and state-driven orchestration motivate controllable reasoning and tool use (Yao et al., 2023b;a; Wu et al., 2024), while recent analyses highlight brittle coordination and long-horizon instability (Cemri et al., 2025; Microsoft Corporation, 2025; Xiong et al., 2025; Sinha et al., 2025). This literature motivates our reference architecture, but not a domain-specific benchmark. OpenSees workflows impose tighter scientific constraints than generic coding tasks because model topology, numerical settings, recorder contracts, physical plausibility, and repair actions must remain mutually consistent.

Benchmarks for Scientific Reasoning, Code, and Agents Scientific and agentic evaluation already has a broad benchmark ecosystem (Rein et al., 2024; Wang et al., 2023; Glazer et al., 2024; Zhang et al., 2025; Lee et al., 2023; Cui et al., 2025; Starace et al., 2025; Bogin et al., 2024; Siegel et al., 2024; Chen et al., 2024; Majumder et al., 2024; Mitchener et al., 2025). Coding benchmarks such as MBPP, Hu-

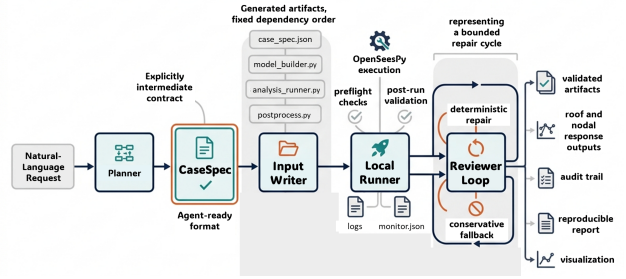


Figure 2. **Reference agent architecture for the current release.** A natural-language request is first mapped to a typed `CaseSpec` by the planner, then expanded by the input writer into dependent OpenSeesPy artifacts, executed by the local runner, and passed through validation and a bounded reviewer loop before visualization and auditable outputs are produced. The implemented path is `planner` → `input_writer` → `local_runner` → `reviewer(loop)` → `visualization`.

manEval, DS-1000, and SWE-Bench test synthesis and debugging, but not whether an agent can produce a physically meaningful, auditable OpenSeesPy workflow (Austin et al., 2021; Chen et al., 2021; Lai et al., 2023; Jimenez et al., 2023). Closer resources include FEABench and structural-engineering datasets such as DrafterBench, SOM-1K, and LLM datasets for building static analysis (Li et al., 2025a;b; Wan et al., 2025; Vega et al., 2025). However, they do not jointly test the three capabilities central here: structural knowledge, prompt-driven numerical code generation, and calibrated end-to-end workflow execution with failure repair.

Position of OpenSeesAgentBench OpenSeesAgentBench fills this gap by shifting the contribution from another agent architecture to a reusable benchmark and reference evaluation stack. By narrowing the domain to OpenSeesPy, it makes the evaluation contract scientifically concrete while combining *OpenSeesQuery*, *OpenSeesCodeBench*, and *OpenSeesWorkflowBench* / *OpenSeesBuildingBench calibration profile*. The benchmark therefore measures not only whether an agent can generate runnable OpenSeesPy code, but also whether it can meet calibrated workflow targets under prompt and wrapper variation, diagnose benchmark-defined failures, and repair them within a bounded budget.

3. OpenSeesAgentBench: current implemented benchmark and reference agent

This section defines the benchmark, the reference agent, and the scoring protocol used throughout the paper. Our goal is to make the evaluation contract explicit before presenting performance claims: we first specify what is measured, then how cases are constructed, and finally how success is scored.

OpenSeesAgentBench is an OpenSeesPy-centered benchmark for earthquake-engineering workflows. It is organized into three layers with increasing operational scope: *OpenSeesQuery* measures prerequisite domain knowledge, *OpenSeesCodeBench* isolates prompt-to-code generation and numerical reasoning, and *OpenSeesWorkflowBench* measures end-to-end execution under validation and bounded repair. For the present difficulty-controlled study, the workflow layer is instantiated by *OpenSeesBuildingBench calibration profile*, a 900-case profile sharded into `easy`, `medium`, and `hard` sets (300 each) with target expected-match rates near 90%, 70%, and 60%. Table 1 summarizes this benchmark design.

3.1. Reference Multi-Agent System

The reference system implemented in `src/opensees_agent/` is a lightweight but executable OpenSeesPy workflow agent built around a typed intermediate contract rather than direct monolithic code generation. The current graph is

`planner` → `input_writer` → `local_runner` → `reviewer(loop)` → `visualization`

and is instantiated as a LangGraph state machine with explicit routing after planning, running, and reviewing. The same implementation also exposes a FastMCP interface with tools for `plan`, `generate_files`, `run`, `review`, `apply_fixes`, `visualize`, and `monitor`. Rather than prompting one model to emit a full solver script in one step, the system maintains explicit intermediate state and records stage transitions in `monitor.jsonl`. Figure 2 summarizes this contract-driven execution flow.

Typed contract: `CaseSpec` The central interface is a strict Pydantic `CaseSpec`. In the current implementation, `CaseSpec` explicitly constrains the model family, geometry, material/section parameters, boundary conditions, mass assignment, damping, excitation, analysis setup, recorders, and units. The supported model families are `sdof`, `2d_frame`, and `3d_frame`, with corresponding `ndm`/`ndf` combinations $(1, 1)$, $(2, 3)$, and $(3, 6)$. The current analysis scope is intentionally narrow: transient analysis only, with Newmark time integration, Newton iteration, Rayleigh damping, and either synthetic sinusoidal or file-based ground-motion input. This contract is therefore not only a serialization format, but a fail-closed interface between natural-language requirements, generated solver artifacts, and benchmark evaluation.

Planner The planner maps a natural-language user request into a validated `CaseSpec`. It combines a domain guard, OpenSees-specific retrieval, and optional LLM synthesis. Importantly, planning is not purely generative: when retrieval confidence is low, the planner skips LLM-based case-spec synthesis and falls back to deterministic template

Table 1. Benchmark structure summary for the current release. Quantitative benchmark information is reported in table form: layer size, task unit, primary metric, and the workflow calibration targets used in the present paper run.

Layer	Scope	Size	Task unit	Primary metric / target
OpenSeesQuery	Domain knowledge	90	Multiple-choice question	Accuracy
OpenSeesCodeBench	Code + numerical reasoning	24	Prompt-driven task	Executability + error + convergence
OpenSeesWorkflowBench / calibration profile	End-to-end workflow	900	Workflow case	EMR targets: 89.67% / 70.33% / 60.67%

construction. It also supports direct `CaseSpec` override for controlled experiments and debugging. This design separates language understanding from configuration validity and makes prompt-only evaluation analyzable.

Input writer Conditioned on a validated `CaseSpec`, the input writer emits five artifacts in a fixed dependency order: `case_spec.json`, `model_builder.py`, `analysis_runner.py`, `postprocess.py`, and `opensees_model.py`. This order is implementation-critical: later files inherit a summarized dependency context from earlier ones so that node labels, section definitions, recorder settings, and output paths remain consistent. The generated `model_builder.py` currently contains separate templates for `sdof`, `2d_frame`, and `3d_frame`; the generated `analysis_runner.py` handles both synthetic and file-based motion input, computes Rayleigh damping from eigenvalues, runs transient analysis, and writes both roof-response and nodal-history artifacts.

Runner and fail-closed execution The execution stage is implemented around local case execution, preflight environment checks, and strict post-run validation. Before generation and execution, the CLI runs preflight checks for Python/OpenSeesPy import failures and other environment issues. During execution, the agent records logs and structured summaries such as `run.out`, `run.err`, `opensees_agent_summary.json`, and `monitor.jsonl`. In addition to single-case execution, the current implementation supports local parallel batch runs over a directory of ground-motion files via `--batch-ground-motions-dir`, automatically normalizing file-based excitation `dt` and duration into the generated `CaseSpec`.

Validation and output artifacts The implementation is deliberately fail-closed: a run is not accepted merely because the Python process terminates. The generated workflow writes response histories to `results/response.csv`, nodal histories to `results/nodal_disp_history.csv`, and model metadata to `results/model_metadata.json`. Visualization then converts these artifacts into static plots and, for 3D cases, deformed-configuration animations when the necessary metadata and nodal histories are available. At the top level, the run summary records termination reason, run status, validation status, metrics, error category, retrieval trace, fix attempts, and the current workflow stage.

Reviewer and bounded repair If execution or validation fails, the reviewer enters a bounded repair loop. The current implementation is explicitly rule-first rather than fully free-form. First, error logs are mapped into a structured error taxonomy including `TAG_DUPLICATE`, `EIGEN_FAIL`, `CONVERGENCE_FAIL`, `TIME_STEP_FAIL`, `NUMERICAL_DIVERGENCE`, `MOTION_INPUT_INVALID`, `PYTHON_RUNTIME_ERROR`, `MATERIAL_PARAM_INVALID`, `GEOM_TRANSF_ERROR`, `MISSING_RECORDER`, `IMPORT_ENV`, `VALIDATION_FAIL`, and `DOMAIN_MISMATCH`. The system then applies deterministic file-level rewrite templates for the selected category. Only in later fix stages does it allow LLM-assisted rewrite planning, and even then, edits are filtered to an allowed set of generated files. This design keeps repair auditable and prevents unconstrained post-hoc rewriting.

Implementation scope and current limits The present agent is best understood as a typed OpenSeesPy execution stack rather than a general multi-solver orchestration system. The implementation already supports three structural families (`sdof`, `2d_frame`, `3d_frame`), local execution, local parallel batch processing over multiple motions, strict validation, and MCP exposure. At the same time, the current path remains intentionally lightweight: it is centered on transient response workflows, local execution is the primary path, and robustness comes from typed contracts, deterministic guards, and bounded repair rather than from unconstrained autonomous exploration.

3.2. Benchmark Layers

OpenSeesQuery *OpenSeesQuery* is the prerequisite-knowledge layer. The current release contains 90 four-choice questions covering structural dynamics, damping, numerical integration, convergence diagnostics, ground-motion handling, OpenSees API usage, recorder design, and validation logic. Each item stores `id`, `split`, `difficulty`, `topic`, `stem`, `choices`, `answer`, `rationale`, and `tags`. The intent is to test whether an agent has the conceptual knowledge required to interpret engineering requests before generating solver artifacts.

OpenSeesCodeBench *OpenSeesCodeBench* is the code-generation and numerical-reasoning layer. The current release contains 24 tasks. Each task provides a natural-language prompt together with a structured problem speci-

fication containing the governing equation, boundary conditions, initial conditions, domain definition, saved values, and numerical method. In official `prompt_only` mode, the agent receives only the prompt plus structured payload injection; in `spec_injection` mode, the benchmark directly supplies the intended `CaseSpec` for scoring and debugging. Each task is evaluated at coarse, medium, and fine resolutions against a reference OpenSeesPy realization so that executability can be separated from numerical correctness.

OpenSeesWorkflowBench and OpenSeesBuildingBench The workflow layer used in the current difficulty-calibration study is instantiated by *OpenSeesBuildingBench calibration profile*. This profile contains 900 workflow cases in total, split into 300 easy, 300 medium, and 300 hard cases. This calibration profile is a difficulty-controlled set rather than a new public/hidden split. Each difficulty shard is generated from an empirically calibrated building seed set and preserves success/failure semantics by cloning both matched and mismatched seeds. The building archetype space is fixed to eight 2D/3D families: `2d_regular_low`, `2d_regular_mid`, `2d_braced_mid`, `2d_soft_story`, `3d_regular_low`, `3d_regular_mid`, `3d_torsional`, and `3d_setback`. In other words, it is designed as a calibrated evaluation profile, not as an arbitrary collection of prompts.

Although cases are stratified by difficulty, they still follow realistic seismic-analysis workflows. Each case starts from a structural archetype and requests linear transient OpenSeesPy analysis under synthetic or file-based ground motion, with optional monitor, repair, and visualization actions and postprocess deliverables such as `modal_summary`, `drift_envelope`, and `story_shear_summary`. Fault-injection episodes remain present through invalid-input diagnosis, time-step reconciliation, recorder restoration, or bounded repair. Each repair case includes an `episode_spec` that records the injected fault, expected diagnosis, allowed repairs, and maximum repair steps.

Prompt robustness and workflow-wrapper diversity The calibration profile embeds prompt diversity directly in the benchmark rather than treating robustness as a separate afterthought. Each source seed is expanded using seven prompt realizations—`canonical`, `surface_reorder`, `bullet_rewrite`, `compressed_synonym`, `freeform_prose_hard`, `noisy_context_hard`, and `mixed_locale_hard`—and five workflow wrappers—`design_brief`, `peer_review`, `qa_note`, `consultant_request`, and `handoff_memo`. This forces the agent to solve the same engineering task under varied linguistic and operational framing while preserving calibrated difficulty semantics.

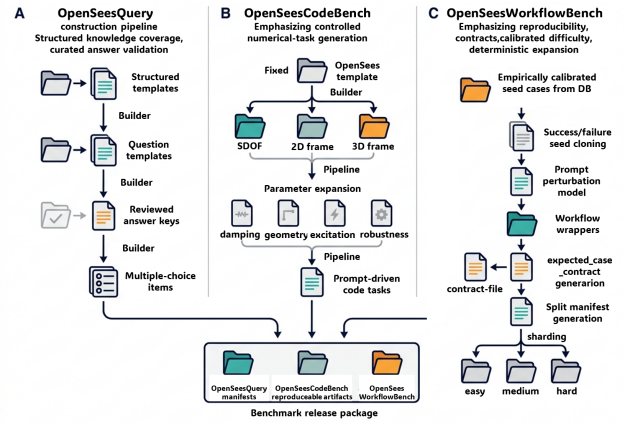


Figure 3. **Deterministic dataset construction and split generation.** *OpenSeesQuery* is assembled from topic templates with reviewed answer keys, *OpenSeesCodeBench* is expanded from fixed SDOF, 2D-frame, and 3D-frame OpenSees templates under controlled numerical settings, and *OpenSeesWorkflowBench* is generated from a calibrated building seed set through prompt perturbations, workflow wrappers, and explicit `expected_case_contracts`. The resulting workflow profile is sharded into 300 easy, 300 medium, and 300 hard cases.

3.3. Dataset Construction and Splits

All benchmark layers are generated through deterministic builders in the repository rather than ad hoc prompt collection. Figure 3 summarizes this reproducible build process. Query items are programmatically assembled from topic templates and reviewed answer keys. CodeBench tasks are constructed from fixed OpenSees templates for SDOF, 2D frame, and 3D frame families, then expanded over damping, geometry, excitation, and robustness settings. For workflow calibration, the profile builder starts from an empirically calibrated building seed set and expands it by perturbation mode and workflow wrapper. Every resulting case stores an `expected_case_contract`, so evaluation can distinguish mere executability from true task grounding. Difficulty labels are tied to explicit target success counts from previously measured seed behavior, which keeps comparisons interpretable across perturbation modes and wrappers.

For actual execution, the 900 workflow cases are sharded by difficulty into three case directories: `easy`, `medium`, and `hard`, each containing 300 cases. This sharding keeps runtime manageable and enables direct comparison against calibrated targets of $269/300$, $211/300$, and $182/300$.

3.4. Evaluation Metrics

OpenSeesQuery OpenSeesQuery is scored by exact multiple-choice accuracy, reported overall and by split/topic. The full formal definition is deferred to Appendix C.

OpenSeesCodeBench OpenSeesCodeBench uses a conjunctive task-success criterion over executability, relative-

error tolerance, and numerical-convergence checks. We report only the aggregate benchmark score in the main text; the binary component checks and formal success definition are deferred to Appendix C.

OpenSeesWorkflowBench Workflow evaluation is stricter than raw process execution. For each case, the benchmark records termination reason, validation status, generated artifacts, and expected metric checks. We define a binary match indicator

$$M_{\text{flow}}^{(i)} = \mathbb{1}\left[\begin{array}{c}\text{termination, validation, and metric checks} \\ \text{match the benchmark specification}\end{array}\right].$$

The primary workflow score is the *expected match rate*,

$$\text{EMR} = \frac{1}{K}\sum_{i=1}^{K} M_{\text{flow}}^{(i)}.$$

This score is reported overall and by split, archetype, motion source, regime, and visibility.

For *OpenSeesBuildingBench calibration profile*, we additionally report the easy, medium, and hard expected match rates, together with deviation from the target success rates 89.67%, 70.33%, and 60.67%. The primary difficulty-calibration score is still expected match rate. The benchmark is designed to preserve target difficulty, not merely to maximize a leaderboard number.

Prompt robustness metrics For prompt-perturbation evaluation, we report overall completed rate and overall expected match rate together with per-perturbation and per-wrapper breakdowns. In the calibration profile, perturbations are embedded directly in benchmark cases, so robustness metrics test whether the same engineering contract remains satisfiable under instruction drift and operational reframing.

Suite-level gates The current implementation uses layer-wise gates to define suite success. A run passes the suite if query accuracy is at least 0.70, CodeBench success is at least 0.70, and workflow expected match rate is at least 0.85. In addition, a separate *spec health* run evaluates whether `spec_injection` achieves at least 0.90 CodeBench success, which serves as a sanity check that the evaluator itself is well-posed.

3.5. Reproducibility and Release Artifacts

The repository produces a paper-style output bundle containing raw reports, aggregate JSON summaries, CSV/Markdown/LaTeX tables, figures, command logs, environment snapshots, dataset cards, datasheets, and Croissant metadata. The final paper experiment used in this work stores its full provenance in `run_manifest.json`, `paper_experiment_report.json`, and the `repro/` directory. This design makes the benchmark auditable at both the case level and the aggregate level. All builders are versioned with deterministic seeds and split manifests, allowing independent reruns to regenerate identical case identifiers and expected-contract files across machines and release dates. At the same time, the public release pipeline intentionally remains fail-closed: strict publication validation only succeeds once external hosting targets and final license approval are explicitly provided.

4. Experiments and Results

This section reports the current workflow calibration study on *OpenSeesBuildingBench calibration profile*, the difficulty-controlled 900-case profile used in the present paper run. The emphasis is benchmark fidelity and executable auditability rather than raw agent saturation: the central question is whether observed expected-match rates track the intended calibration targets across easy, medium, and hard shards once the workflow profile is materialized into runnable cases.

4.1. Experimental Setup

The released benchmark remains three-layered: *OpenSeesQuery* contains 90 multiple-choice questions and *OpenSeesCodeBench* contains 24 prompt-driven OpenSeesPy code-generation tasks. In this revision, however, the new quantitative evidence is concentrated in the workflow layer. The third layer is instantiated here by *OpenSeesBuildingBench calibration profile*, a 900-case profile sharded into `easy`, `medium`, and `hard` subsets with 300 cases each.

The calibration profile is expanded from an empirically calibrated building seed set. Because the expansion procedure is designed to preserve benchmark-defined success/failure semantics across matched and mismatched variants, the primary quantitative role of the present run is internal calibration checking: can the released reference stack execute the expanded corpus while preserving the intended shard-level difficulty profile under prompt and wrapper variation? These numbers should therefore be read as evidence about benchmark construction and executable reproducibility, not as independent proof of broad structural-analysis generalization.

Query and CodeBench are unchanged in this revision and are included as released benchmark components with fixed scoring protocols. The current paper does not yet add new cross-model or prompt-regime comparisons for those layers, so the workflow layer carries the main empirical load.

We evaluate the offline reference execution stack on the three calibration shards under `prompt_only` execution with strict validation enabled. Because this is a calibration

profile rather than a public/hidden split, the key comparison in this draft is between observed shard-level performance and the calibrated shard targets summarized in Table 2.

A complete comparative systems study on this benchmark should evaluate at least five configurations under one shared contract: a single-agent baseline, a MASSE-like role-decomposed baseline, a lightweight 2DFrame-like OpenSeesPy baseline, a FeaGPT-like end-to-end FEA baseline, and the full reference system, as summarized in Appendix Table 3. The current paper reports only the full reference stack, so the numbers below should be interpreted as benchmark-calibration and reference-stack audit results, not as isolating the gains of any particular orchestration strategy.

Workflow performance is measured by expected match rate rather than raw completion. A case counts as matched only if termination reason, validation status, metric bounds, and `expected_case_contract` agree with the benchmark specification. For the calibration profile we report completed rate and expected match rate by difficulty shard, together with deviation from the calibrated targets in both cases and percentage points. This makes it possible to separate operational completion from benchmark-level task satisfaction.

4.2. Results

Table 2 is the primary quantitative summary for the workflow calibration study. The medium and hard shards match the calibrated expected-match targets exactly, the easy shard differs by one case, and the overall result stays within one case of the 900-case target. Our main interpretation is therefore not that the reference stack is uniquely strong, but that the executable calibration profile preserves its intended shard-level difficulty after expansion and contract-based evaluation.

The hard shard deserves an explicit caveat. One split runner became unstable after emitting partial case directories but before writing a top-level report. The reported hard aggregate is therefore an audited reconstruction assembled from completed shard reports, reconstructed crash-interrupted segments, and a direct rerun of the final case. This is sufficient for calibration accounting, but it is weaker than a clean single-pass report and should not be over-interpreted as fine-grained evidence about architectural advantage.

Table 2 also shows that completed rate remains above expected match rate, with the gap widening on the harder shards. Many cases still run to completion but fail contract-level checks. This is a useful property of the benchmark because it distinguishes executability from contract satisfaction and prevents inflated scores from partially correct workflows.

Unlike the earlier workflow study, the calibration profile does not treat prompt perturbation as a separate side benchmark. The 900-case corpus already mixes seven prompt realizations and five workflow wrappers inside the difficulty shards, so the shard-level expected-match rates jointly reflect robustness to instruction drift and robustness to operational framing while holding the physical engineering task fixed. Here again, the point is benchmark design: robustness is baked into the evaluation profile rather than introduced as a post hoc stress test.

To complement the aggregate calibration metrics, Figure 4 presents four qualitative execution examples drawn from the calibration profile. These panels provide case-level illustrations of the outputs produced by the workflow benchmark under prompt and wrapper variation. They support qualitative interpretation of the benchmark behavior, while the primary quantitative conclusions still come from the shard-level calibration results.

4.2.1. Discussion

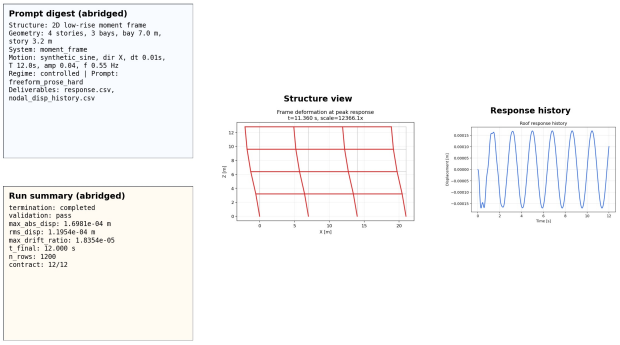
The updated workflow evaluation is more informative than the earlier saturated release because it no longer treats 100% success as the only meaningful outcome. For this paper, the key question is whether the workflow benchmark retains a calibrated difficulty profile after executable expansion and whether the released reference stack materializes that profile under the stated contract. Because the calibration profile is derived from empirically calibrated seeds and the expansion procedure is designed to preserve benchmark-defined success/failure semantics, agreement with the shard targets should be read primarily as an internal fidelity result. It shows that the current benchmark construction does not collapse into a trivial executability test and that the released execution stack is sufficiently stable to reproduce the intended profile in practice, while not yet establishing independent difficulty validation, generalization beyond the calibrated profile, or superiority over alternative agent organizations. Because prompt perturbations and workflow wrappers are embedded into the cases themselves, the shard-level scores should be read as calibrated operational metrics rather than as a standalone paraphrase benchmark.

4.2.2. Limitations

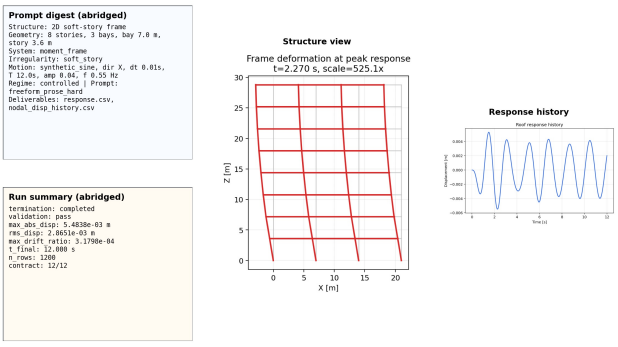
First, Query and CodeBench are part of the released benchmark but are not newly stress-tested in this revision, so the new evidence is concentrated in the workflow layer. Second, because the workflow profile is expanded from calibrated seeds that preserve benchmark-defined success/failure semantics, matching target expected-match rates is evidence of benchmark fidelity rather than non-circular external difficulty validation. Third, this is a local calibration profile rather than a public/hidden release split, and the draft

Table 2. Primary quantitative results on *OpenSeesBuildingBench calibration profile*. The table should be read as a workflow calibration audit of the released reference stack, reporting target counts/rates, completed counts/rates, observed match counts/rates, and deviation from the calibrated target.

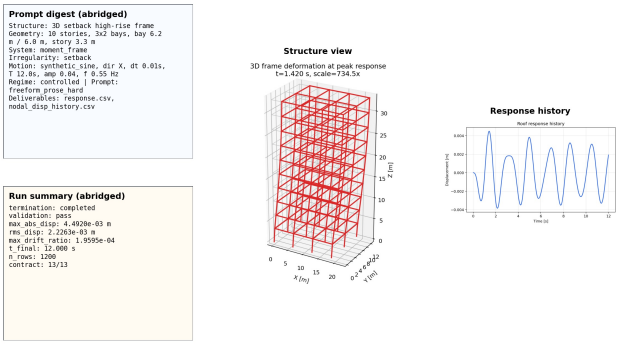
Shard	Target	Target EMR	Completed	Comp. rate	Observed	Observed EMR	Deviation
Easy	269/300	89.67%	299/300	99.67%	268/300	89.33%	−1 case / −0.33 pp
Medium	211/300	70.33%	265/300	88.33%	211/300	70.33%	0 case / 0.00 pp
Hard	182/300	60.67%	196/300	65.33%	182/300	60.67%	0 case / 0.00 pp
Overall	662/900	73.56%	760/900	84.44%	661/900	73.44%	−1 case / −0.11 pp



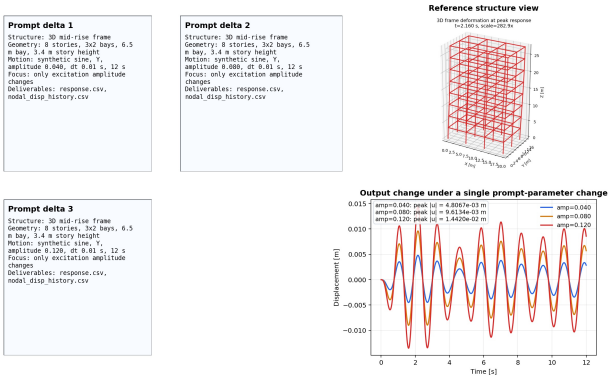
(a) Execution example 01.



(b) Execution example 04.



(c) Execution example 08.



(d) Execution example 11.

Figure 4. Qualitative execution examples from *OpenSeesBuildingBench calibration profile*, shown for cases 01, 04, 08, and 11. These panels illustrate representative case-level outputs and complement the aggregate calibration results, but they are not introduced as separate quantitative metrics.

still lacks the key five-way comparison among single-agent, MASSE-like, 2DFrame-like, FeaGPT-like, and full-system baselines under one evaluator (Appendix Table 3). Fourth, the hard-shard aggregate includes an audited reconstruction after runner instability, and repair success remains an auxiliary rather than primary calibration diagnostic.

5. Conclusion

We introduce *OpenSeesAgentBench*, an OpenSeesPy benchmark and executable reference evaluation stack for structural-analysis agents. The release spans *OpenSeesQuery*, *OpenSeesCodeBench*, and *OpenSeesWorkflowBench*; this paper focuses on the workflow-layer *OpenSeesBuildingBench calibration profile*.

The reference stack couples a strict `CaseSpec` planner with dependency-aware generation, execution checks, and bounded repair. On the calibration profile it reaches expected-match rates of 89.33%, 70.33%, and 60.67%, within one case of the 900-case target overall. We read this as benchmark-calibration evidence rather than an architectural ranking. The next step is controlled comparison with single-agent, MASSE-like, 2DFrame-like, and FeaGPT-like baselines, together with cleaner hard-shard reruns and stronger independent difficulty validation.

References

Adil, M., Lee, G., Gonzalez, V. A., and Mei, Q. Using vision language models for safety hazard identification in construction. *ArXiv*, abs/2504.09083, 2025.

Aqib, M., Hamza, M., Mei, Q., and Chui, Y. H. Fine-tuning large language models and evaluating retrieval methods for improved question answering on building codes. *ArXiv*, abs/2505.04666, 2025.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Avila, C., Ilbay, D., and Rivera, D. Human–ai teaming in structural analysis: A model context protocol approach for explainable and accurate generative ai. *Buildings*, 15 (17):3190, 2025.

Bogin, B., Yang, K., Gupta, S., Richardson, K., Bransom, E., Clark, P., Sabharwal, A., and Khot, T. Super: Evaluating agents on setting up and executing tasks from research repositories. *arXiv preprint arXiv:2409.07440*, 2024.

Cemri, M., Pan, M. Z., Yang, S., Agrawal, L. A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., et al. Why do multi-agent llm systems fail? *ArXiv*, abs/2503.13657, 2025.

Chen, J. and Bao, Y. Multi-agent large language model framework for code-compliant automated design of reinforced concrete structures. *Automation in Construction*, 177:106331, 2025.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Chen, Z., Chen, S., Ning, Y., Zhang, Q., Wang, B., Yu, B., Li, Y., Liao, Z., Wei, C., Lu, Z., et al. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. *arXiv preprint arXiv:2410.05080*, 2024.

Cui, H., Shamsi, Z., Cheon, G., Ma, X., Li, S., Tikhanovskaya, M., Norgaard, P., Mudur, N., Plomecka, M., Raccuglia, P., et al. Curie: Evaluating llms on multitask scientific long context understanding and reasoning. *arXiv preprint arXiv:2503.13517*, 2025.

Deng, Z., Du, C., Nousias, S., and Borrmann, A. Bimgent: Towards autonomous building modeling via computer-use agents, 2025. URL https://arxiv.org/abs/2506.07217.

Du, C., Esser, S., Nousias, S., and Borrmann, A. Text2bim: Generating building models using a large language model-based multi-agent framework, 2025. URL https://arxiv.org/abs/2408.08054.

Fan, C., Mei, Q., Wang, X., and Li, X. Ergochat: A visual query system for the ergonomic risk assessment of construction workers. *ArXiv*, abs/2412.19954, 2024.

Fourney, A., Bansal, G., Mozannar, H., Tan, C., Salinas, E., Niedtner, F., Proebsting, G., Bassman, G., Gerrits, J., Alber, J., et al. Magentic-one: A generalist multi-agent system for solving complex tasks. *ArXiv*, abs/2411.04468, 2024.

Geng, Z., Liu, J., Cao, R., Cheng, L., Wang, H., and Cheng, M. A lightweight large language model-based multi-agent system for 2d frame structural analysis. *arXiv preprint arXiv:2510.05414*, 2025.

Glazer, E., Erdil, E., Besiroglu, T., Chicharro, D., Chen, E., Gunning, A., Olsson, C. F., Denain, J.-S., Ho, A., Santos, E. d. O., et al. Frontiermath: A benchmark for evaluating advanced mathematical reasoning in ai. *arXiv preprint arXiv:2411.04872*, 2024.

Han, Z., Li, L., Yin, Y., Wang, W., Li, M., Jiang, Y., et al. Pragmatic prompting: A comprehensive framework for software development with actionable insights from ai-generated code. *arXiv preprint arXiv:2603.07728*, 2026.

Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S., Lin, Z., and Zhou, L. MetaGPT: Meta programming for multi-agent collaborative framework. *arXiv preprint*, arXiv:2308.00352, 2023.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Joffe, I., Felobes, G., Elgouhari, Y., Kalaleh, M. T., Mei, Q., and Chui, Y. H. The framework and implementation of using large language models to answer questions about building codes and standards. *Journal of Computing in Civil Engineering*, 2025. ISSN 0887-3801. doi: 10.1061/JCCEE5.CPENG-6037.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, W.-t., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Lee, Y., Lee, K., Park, S., Hwang, D., Kim, J., Lee, H.-i., and Lee, M. Qasa: advanced question answering on scientific articles. In *International Conference on Machine Learning*, pp. 19036–19052. PMLR, 2023.

Li, W., Zhang, X., Guo, Z., Mao, S., Luo, W., Peng, G., Huang, Y., Wang, H., and Li, S. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. *arXiv preprint arXiv:2503.06680*, 2025a.

Li, Y., Dong, Z., and Shao, Y. Drafterbench: Benchmarking large language models for tasks automation in civil engineering, 2025b. URL https://arxiv.org/abs/2507.11527.

Liang, H., Kalaleh, M. T., and Mei, Q. Integrating large language models for automated structural analysis. *ArXiv*, abs/2504.09754, 2025a.

Liang, H., Zhou, Y., Talebi-Kalaleh, M., and Mei, Q. Automating structural engineering workflows with large language model agents. *arXiv preprint arXiv:2510.11004*, 2025b.

Liu, J., Geng, Z., Cao, R., Cheng, L., Bocchini, P., and Cheng, M. A large language model-empowered agent for reliable and robust structural analysis, 2025. URL https://arxiv.org/abs/2507.02938.

Majumder, B. P., Surana, H., Agarwal, D., Mishra, B. D., Meena, A., Prakhar, A., Vora, T., Khot, T., Sabharwal, A., and Clark, P. Discoverybench: Towards data-driven discovery with large language models. *arXiv preprint arXiv:2407.01725*, 2024.

Microsoft Corporation. Taxonomy of failure modes in agentic ai systems: Whitepaper. Technical report, Microsoft, 2025. URL https://cdn-dynmedia-1.microsoft.com/is/content/microsoftcorp/microsoft/final/en-us/microsoft-brand/documents/Taxonomy-of-Failure-Mode-in-Agentic-AI-Systems-Whitepaper.pdf.

Mitchener, L., Laurent, J. M., Tenmann, B., Narayanan, S., Wellawatte, G. P., White, A., Sani, L., and Rodriques, S. G. Bixbench: a comprehensive benchmark for llm-based agents in computational biology. *arXiv preprint arXiv:2503.00096*, 2025.

Phan, H. N., Nguyen, T. N., Nguyen, P. X., and Bui, N. D. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *ArXiv*, abs/2409.16299, 2024.

Qi, Y., Xu, R., and Chu, X. Feagpt: An end-to-end agentic-ai for finite element analysis. *arXiv preprint arXiv:2510.21993*, 2025.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., et al. Chatdev: Communicative agents for software development. *ArXiv*, abs/2307.07924, 2023.

Qin, S., Guan, H., Liao, W., Gu, Y., Zheng, Z., Xue, H., and Lu, X. Intelligent design and optimization system for shear wall structures based on large language models and generative artificial intelligence. *Journal of Building Engineering*, 95:109996, 2024.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

Siegel, Z. S., Kapoor, S., Nagdir, N., Stroebl, B., and Narayanan, A. Core-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *arXiv preprint arXiv:2409.11363*, 2024.

Sinha, A., Arun, A., Goel, S., Staab, S., and Geiping, J. The illusion of diminishing returns: Measuring long horizon execution in llms. *ArXiv*, abs/2509.09677, 2025.

Starace, G., Jaffe, O., Sherburn, D., Aung, J., Chan, J. S., Maksin, L., Dias, R., Mays, E., Kinsella, B., Thompson, W., et al. Paperbench: Evaluating ai's ability to replicate ai research. *arXiv preprint arXiv:2504.01848*, 2025.

Vega, C. F. A., Yupa, D. J. I., Tapia, P. V. T., and Tapia, E. D. R. Toward responsible ai in high-stakes domains: A dataset for building static analysis with llms in structural engineering, 2025.

Wan, Q., Wang, Z., Zhou, J., Wang, W., Geng, Z., Liu, J., Cao, R., Cheng, M., and Cheng, L. Som-1k: A thousand-problem benchmark dataset for strength of materials. *ArXiv*, abs/2509.21079, 2025.

Wang, X., Hu, Z., Lu, P., Zhu, Y., Zhang, J., Subramaniam, S., Loomba, A. R., Zhang, S., Sun, Y., and Wang, W. Scibench: Evaluating college-level scientific problem-solving abilities of large language models. *arXiv preprint arXiv:2307.10635*, 2023.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Wu, Y., Yue, T., Zhang, S., Wang, C., and Wu, Q. Stateflow: Enhancing llm task-solving through state-driven

workflows. In *First Conference on Language Modeling*, 2024.

Xie, H., Mei, Q., and Chui, Y. H. Ai applications for structural design automation. *Automation in Construction*, 179:106496, 2025.

Xiong, Z., Lin, Y., Xie, W., He, P., Tang, J., Lakkaraju, H., and Xiang, Z. How memory management impacts llm agents: An empirical study of experience-following behavior. *ArXiv*, abs/2505.16067, 2025.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. R. Tree of thoughts: Deliberate problem solving with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023a.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023b.

Youwai, S., Phim, D., Murcia, V. G., and Onas, R. C. Investigating the potential of large language model-based router multi-agent architectures for foundation design automation: A task classification and expert selection study, 2025. URL https://arxiv.org/abs/2506.13811.

Zhang, X., Dong, Y., Wu, Y., Huang, J., Jia, C., Fernando, B., Shou, M. Z., Zhang, L., and Liu, J. Physreason: A comprehensive benchmark towards physics-based reasoning. *arXiv preprint arXiv:2502.12054*, 2025.

A. Supplementary System, Benchmark, and Release Details

This appendix documents the *currently implemented* OpenSeesPy-centered benchmark and reference agent used in this paper. The repository is centered on *OpenSeesPy* rather than heterogeneous routing across multiple production simulators. Accordingly, the appendix focuses on the implemented execution contract, benchmark layers, evaluator behavior, and reproducibility artifacts exposed by the current codebase.

A.1. Implemented Benchmark Layers

The current benchmark stack contains three layers:

- **OpenSeesQuery**: 90 multiple-choice questions covering structural dynamics, damping, numerical integration, recorder design, OpenSeesPy API usage, and validation logic.
- **OpenSeesCodeBench**: 24 prompt-driven code-generation and numerical-reasoning tasks.
- **OpenSeesWorkflowBench**: the end-to-end workflow layer used for prompt-to-artifact-to-execution evaluation under validation and bounded repair.

The benchmark is OpenSeesPy-centered by design: its purpose is to evaluate whether an agent can transform natural-language structural-analysis requests into validated OpenSeesPy artifacts with auditable outputs. The current release is intentionally narrower than a broader multi-solver orchestration vision and should be interpreted as a reproducible benchmark and reference execution stack for structural dynamics workflows.

A.2. Reference Agent: Current Implemented Workflow

The reference agent is implemented in `src/opensees_agent/` and follows a staged LangGraph workflow:

$$\texttt{planner} \rightarrow \texttt{input\_writer} \rightarrow \texttt{local\_runner} \rightarrow \texttt{reviewer(loop)} \rightarrow \texttt{visualization}.$$

This design is intentional. Rather than prompting a monolithic script directly from a user request, the system makes intermediate state explicit through a strict Pydantic-validated `CaseSpec`. The `CaseSpec` contract records the intended model family, geometry, material/section parameters, mass and damping settings, excitation, analysis setup, recorder configuration, and units. The same contract is used for generation, execution, validation, and audit logging.

Planner The planner maps a natural-language user requirement into a validated `CaseSpec`. It combines a domain guard, OpenSees-specific retrieval, and optional LLM synthesis. If LLM access is unavailable or retrieval confidence is too weak, the planner can fall back to deterministic template-based construction. This design separates language understanding from configuration validity and makes prompt-only evaluation analyzable.

Input writer Conditioned on the validated `CaseSpec`, the input writer emits the following artifacts in fixed dependency order:

- `case_spec.json`
- `model_builder.py`
- `analysis_runner.py`
- `postprocess.py`
- `opensees_model.py`

The dependency order is implementation-critical because later files inherit a summarized dependency context from earlier ones. This preserves consistency of node labels, section definitions, recorder configuration, and output paths.

Runner and fail-closed validation The runner executes the generated OpenSeesPy workflow in a case directory and writes structured outputs such as:

- `run.out`, `run.err`
- `results/response.csv`
- `planning_summary.json`
- `opensees_agent_summary.json`
- `monitor.jsonl`

Before execution, the system performs preflight checks for Python/OpenSeesPy import failures and environment issues. Runtime acceptance is fail-closed: a case is accepted only if both execution and post-run validation succeed. Validation checks include finite outputs, CSV schema consistency, response-order sanity, and task-dependent output requirements.

Reviewer and bounded repair If execution or validation fails, the reviewer enters a bounded repair loop. The current implementation uses a three-stage policy:

1. deterministic rule-based repair;
2. constrained LLM rewrite with allowed-file restrictions;
3. conservative fallback.

The reviewer therefore does not perform unconstrained self-modification; instead, it applies bounded post-execution correction under an explicit failure taxonomy and repair budget.

Visualization If requested or enabled, the visualization stage converts generated response histories into static artifacts such as `response.png`. For 3D cases, the implementation may also emit deformed-configuration visual outputs when the required metadata and nodal histories are present.

A.3. Implementation Scope and Current Limits

The present implementation should be interpreted as a typed OpenSeesPy execution stack rather than a general multi-solver orchestration framework. It already supports:

- model families `sdof`, `2d_frame`, and `3d_frame`;
- local case execution;
- local parallel batch runs over a directory of ground-motion files;
- strict validation;
- MCP exposure of planning, generation, execution, review, fixing, visualization, and monitoring.

At the same time, the current OpenSeesPy path remains intentionally lightweight. The implementation is centered on transient workflows and local execution, and robustness comes primarily from typed contracts, deterministic guards, and bounded repair. The migration notes also still describe the OpenSeesPy path as experimental and list MCP hardening, SLURM backend maturation, nonlinear extension, and regression benchmarking as next implementation priorities.

A.4. Minimum Comparative Configurations

A complete systems comparison on this benchmark should evaluate five configurations under one shared contract-first evaluator. Table 3 summarizes the minimum comparison set referenced in the main text. The table records intended comparison roles only; it does not report quantitative results because the present paper executes only the full reference system.

Table 3. Minimum comparative configurations for a controlled systems study on *OpenSeesBuildingBench calibration profile*. All configurations should be evaluated under the same `CaseSpec` contract, validator, and expected-match protocol.

Configuration	Comparison role	Representative structure	Status
Single-agent baseline	Minimal no-decomposition baseline for prompt-to-artifact generation and repair	One agent or one prompt loop handles planning, generation, execution feedback, and repair	Not reported
MASSE-like baseline	Role-decomposed structural-engineering baseline	Multiple specialized agents coordinate planning, analysis design, and review	Not reported
2DFrame-like baseline	Lightweight OpenSeesPy-specific decomposition baseline	Small multi-agent pipeline focused on OpenSeesPy artifact generation and validation	Not reported
FeaGPT-like baseline	End-to-end FEA automation baseline	Broader task-to-workflow automation with tool orchestration across analysis stages	Not reported
Reference system	Current typed contract-first system	`planner` → `input_writer` → `local_runner` → `reviewer(loop)` → `visualization`	Reported

A.5. Execution Interfaces

The repository exposes two execution surfaces.

CLI surface The main entry point is `opensees_main.py`. The implemented CLI supports:

- `--prompt_path`
- `--output_dir`
- `--always_visualize`
- `--case-spec-path`
- `--batch-ground-motions-dir`
- `--backend local|slurm`
- `--rag-mode faiss_hybrid|lexical_only`
- `--generation-mode sequential_dependency|parallel_no_context`
- `--strict-validation/--no-strict-validation`
- `--max-fix-per-category`

MCP surface The repository also exposes a FastMCP server for staged external use. The current tool surface includes:

- `plan`
- `generate_files`
- `run`
- `review`
- `apply_fixes`
- `visualize`
- `monitor`

This MCP interface is important because it exposes the same typed workflow stages to external agent systems rather than hiding them behind one monolithic CLI.

B. Supplementary Agent Contract

B.1. Typed `CaseSpec`

The central wire format is `CaseSpec`. In the current implementation it explicitly constrains:

- model family;
- geometry and dimensionality;
- materials and sections;
- mass assignment;
- damping;
- excitation source and direction;
- analysis setup;
- recorder and output requirements;
- units.

The supported model families are:

- `sdof`
- `2d_frame`
- `3d_frame`

The current implementation is centered on transient analysis with Rayleigh damping and either synthetic or file-based ground-motion input. This is important because the benchmark contract should reflect what the code actually supports, rather than a broader future system vision.

B.2. Generated File Contract

The current OpenSeesPy path uses a stable artifact contract. The most important generated or emitted files are:

- `case_spec.json`
- `model_builder.py`
- `analysis_runner.py`
- `postprocess.py`
- `opensees_model.py`
- `run.out`, `run.err`
- `results/response.csv`
- `results/response.png`
- `planning_summary.json`
- `opensees_agent_summary.json`
- `monitor.jsonl`

This contract matters because it gives the benchmark auditable intermediate state rather than only final success/failure labels.

C. Supplementary Evaluator Specification

C.1. Layer-wise Metrics

OpenSeesQuery Query is scored by exact multiple-choice accuracy:

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}[\hat{y}_i = y_i].$$

OpenSeesCodeBench Each CodeBench task is successful only if all three binary checks pass:

- executability;
- relative error;
- convergence.

The task success indicator is:

$$M_{\text{code}}^{(i)} = \mathbb{1}\left[M_{\text{exec}}^{(i)} = 1 \wedge M_{\text{err}}^{(i)} = 1 \wedge M_{\text{conv}}^{(i)} = 1\right].$$

The benchmark score is the mean of $M_{\text{code}}^{(i)}$ over all tasks.

OpenSeesWorkflowBench Workflow tasks are scored using expected-match evaluation. A case is counted as matched only if:

- the observed termination reason matches the benchmark expectation;
- validation status matches the benchmark expectation;
- required metric checks match the benchmark contract; and
- the structured expected case contract is satisfied.

The workflow score is the expected match rate:

$$\text{EMR} = \frac{1}{K}\sum_{i=1}^{K} M_{\text{flow}}^{(i)}.$$

C.2. Prompt Robustness Metrics

For prompt-perturbation evaluation, we report overall completed rate and overall expected match rate together with per-perturbation and per-wrapper breakdowns. These metrics are intended to test whether the same engineering contract remains satisfiable under instruction drift and operational reframing.

C.3. Suite Gates

The suite uses layer-wise gates:

- Query accuracy ≥ 0.70
- CodeBench success rate ≥ 0.70
- Workflow expected match rate ≥ 0.85

All three must pass for suite success.

In addition, we run a separate *spec health* check in which CodeBench is executed in `spec_injection` mode. This check is passed if CodeBench success is at least 0.90. The purpose of spec health is to verify evaluator soundness independently of prompt-only generation quality.

D. Supplementary Failure Taxonomy

The reviewer uses a structured failure taxonomy. The most important categories exposed by the current implementation are:

- MOTION_INPUT_INVALID
- TIME_STEP_FAIL
- CONVERGENCE_FAIL
- NUMERICAL_DIVERGENCE
- MATERIAL_PARAM_INVALID
- GEOM_TRANSF_ERROR
- PYTHON_RUNTIME_ERROR
- MISSING_RECORDER
- VALIDATION_FAIL
- IMPORT_ENV
- DOMAIN_MISMATCH
- UNKNOWN

These categories serve two purposes: they guide bounded repair in the reference agent, and they provide structured failure traces for benchmark reporting.

E. Representative Example Tasks

E.1. Canonical Controlled Workflow Example

Example Controlled Prompt

```
Construct a 2D regular low-rise moment frame in OpenSeesPy.  Use a three-story,
two-bay configuration, assign lumped floor masses, apply Rayleigh damping, run linear
time-history analysis under a synthetic ground motion, and save roof displacement plus
nodal response histories.
```

E.2. File-Motion Batch Example

Example Batch-Ground-Motion Prompt

```
Evaluate the same 3D frame archetype against all motions in the provided directory,
normalize motion dt from each file, run strict validation, and summarize failed jobs
by error category.
```

E.3. Repair Episode Example

Example Repair Episode

```
Initial run fails with:
RuntimeError:  motion input invalid.  Ground motion file not found:
./motions/missing_fault_motion.csv
The agent must diagnose MOTION_INPUT_INVALID, restrict changes to the allowed file set,
and recover within the bounded repair budget.
```

F. Reproducibility Notes

The implemented OpenSeesPy path writes structured provenance artifacts for both single-case and batch execution. Depending on the run mode, these include:

- per-case logs and generated files;
- `planning_summary.json`;
- `opensees_agent_summary.json`;
- `opensees_agent_batch_summary.json`;
- `monitor.jsonl`;
- response CSV and plot artifacts;
- aggregate benchmark reports and tables produced by the paper experiment scripts.

This design makes the benchmark auditable at both the case level and the aggregate level. The same artifact structure is also what enables regression evaluation, bounded repair analysis, and future benchmark extension without changing the top-level evaluation contract.