Foam-Agent: A Large Language Model-Based Multi-Agent Framework for Automating Computational Fluid Dynamics Workflows

Ling Yue[a], Nithin Somasekharan[b], Tingwen Zhang[a], Yadi Cao[c], Zhangze Chen[d], Shimin Di[e] and Shaowu Pan[b,*]

[a]*Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 12180, USA*
[b]*Department of Mechanical, Aerospace, and Nuclear Engineering, Rensselaer Polytechnic Institute, Troy, NY, 12180, USA*
[c]*Department of Computer Science and Engineering, University of California San Diego, La Jolla, CA, 92093, USA*
[d]*College of Education, Zhejiang Normal University, Jinhua, Zhejiang, 321004, China*
[e]*School of Computer Science and Engineering, Southeast University, Nanjing, Jiangsu, 210096, China*

ARTICLE INFO

Keywords:
Computational fluid dynamics
Large language models
Multi-agent systems
Retrieval-augmented generation
OpenFOAM
Scientific computing

ABSTRACT

Computational fluid dynamics (CFD) has been the main workhorse of computational physics, yet its steep learning curve and fragmented, multi-stage workflow create significant barriers to entry. We present Foam-Agent, a multi-agent framework that leverages large language models (LLMs) to automate the end-to-end CFD workflow in OpenFOAM from a single natural-language prompt. Foam-Agent rests on three methodological contributions. First, a *multi-index retrieval* scheme organizes domain knowledge along four complementary structural dimensions and selects indices by workflow stage, sharpening retrieval precision over conventional single-index retrieval-augmented generation. Second, *dependency-aware file generation* is formulated as a topological traversal of the OpenFOAM case dependency graph, so that each configuration file is synthesized in the context of its already-generated predecessors, enforcing cross-file consistency. Third, a *trajectory-conditioned reviewer loop* iteratively repairs failed runs by conditioning each correction on the accumulated error-and-diagnosis trajectory of its own previous attempts, applying at each step a minimal configuration edit that targets a reduced solver-error set. Around these contributions, six specialist agents span planning, meshing, file writing, execution, review, and visualization; Foam-Agent additionally exposes its capabilities through the Model Context Protocol as a deployment surface for external orchestrators. On FoamBench, Foam-Agent achieves an 88.2% execution success rate on the 110 Basic-tier tasks and 62.5% on the out-of-distribution Advanced tier, 1.6× and 5× those of the prior MetaOpenFOAM framework, both with Claude 3.5 Sonnet, all without expert intervention, and we report a complementary field-level fidelity metric to distinguish crash-free execution from solution accuracy. These results demonstrate how the strategic harnessing of specialized multi-agent systems can reduce expertise barriers while preserving the rigor of solver-based simulation workflows.

1. Introduction

Computational fluid dynamics (CFD) (Kumar, Moharana, Naik, Prof, Singh and Patel, 2020) underpins much of modern science and engineering. It is used to simulate fluid phenomena in applications that include aircraft (Wang, 2014; Bravo-Mosquera, Cerón-Muñoz, Díaz-Vázquez and Catalano, 2018), wind-turbine design (Aliyar, Bredmose, Roenby, Tomaselli and Sarlak, 2025; Hartwanger and Horvat, 2008; Lanzafame, Mauro and Messina, 2013), and blood flow in arteries (Kong, Kheyfets, Finol and Cai, 2018; Steinman, 2002; Zhang, Zhong, Su, Wan, Yap, Tham, Chua, Ghista and Tan, 2014). It also lowers the cost of physical prototyping (Tu, Yeoh, Liu and Tao, 2023). However, mastering numerical simulations in OpenFOAM (Jasak, Jemcov, Tukovic et al., 2007), along with meshing tools such as Gmsh (Geuzaine and Remacle, 2009) and post-processing utilities, has a steep learning curve in practice. Such workflow requires several steps ranging from pre-processing, such as geometry creation and meshing, context-dependent solver configuration, to post-processing for visualization (Slotnick, Khodadoust, Alonso, Darmofal, Gropp, Lurie and Mavriplis, 2014). As a result, this multi-stage process demands substantial domain expertise, and the resulting

*Corresponding author.
pans2@rpi.edu (S. Pan)
ORCID(s): 0000-0002-2462-362X (S. Pan)

learning curve limits access for many researchers and engineers (Höpken and Mooney, 2012). Recent advances in generative AI, particularly Large Language Models (LLMs), have drawn our attention to the autonomous "AI agents" that can carry out multi-step scientific workflows (Xi, Chen, Guo, He, Ding, Hong, Zhang, Wang, Jin, Zhou et al., 2025). These agents can interpret high-level natural language commands, break down tasks, call software tools, and iteratively refine solutions (Yao, Zhao, Yu, Du, Shafran, Narasimhan and Cao, 2023). Several domains have already adopted them for automation of scientific workflows. In chemistry, ChemCrow (M. Bran, Cox, Schilter, Baldassari, White and Schwaller, 2024) automates synthesis planning by calling external chemical tools. In materials science, multi-agent systems have identified new high-performance alloys by reasoning over material databases (Ghafarollahi and Buehler, 2025). Across these cases, agents autonomously take over tedious repetitive setup work that previously required a domain specialist. Within engineering, this trend has taken root most deeply in computational mechanics. MechGPT (Buehler, 2024) introduced a domain-specialized LLM for mechanics knowledge, and MeLM (Buehler, 2023) provided an early language-model formulation that solves forward and inverse mechanics problems. Multi-agent collaborations followed: MechAgents (Ni and Buehler, 2024) showed that teams of LLM agents can set up and solve mechanics problems and integrate knowledge across them, and ProtAgents (Ghafarollahi and Buehler, 2024) coordinated LLM agents for knowledge retrieval, protein-structure analysis, and physics-based simulation to autonomously design and analyze de novo proteins with targeted mechanical properties. The same agentic approach has reached finite-element analysis (FEA), with frameworks like AutoFEA (Hou, Johnson, Makhija, Chen and Ye, 2025) and MooseAgent (Zhang, Liu, Xin and Jiao, 2025) automating solver-setup workflows, and ALL-FEM (Deotale, Srinivasan, Golestanian, Tian, Zhang, Vlachos and Gomez, 2026) utilizing a fine-tuned LLM for finite-element code generation. More broadly agentic systems have also appeared for further downstream tasks, such as model order reduction (Guo, Park, Qian, Hughes and Liu, 2026).

Within CFD specifically, LLM-based agents have begun to translate natural-language problem descriptions into executable OpenFOAM configuration files. Early systems such as MetaOpenFOAM (Chen, Zhu, Zhou and Ren, 2025) and OpenFOAMGPT (Pandey, Xu, Wang and Chu, 2025) show early promise on this task, typically by way of Retrieval-Augmented Generation (RAG) (Lewis, Perez, Piktus, Petroni, Karpukhin, Goyal, Küttler, Lewis, Yih, Rocktäschel et al., 2020): the model retrieves relevant examples from a knowledge base and produces plausible solver settings from existing case templates. However, several limitations remain. First, their workflow coverage is narrow, focusing almost exclusively on isolated solver configuration while omitting critical pre- and post-processing stages. Second, they generate configuration files independently, failing to account for the rigid cross-file dependencies and structural hierarchy within an OpenFOAM case. Third, their execution paradigm is strictly open-loop, meaning they produce simulation files statically and cannot dynamically diagnose or correct runtime errors when a simulation fails.

In this work, we present Foam-Agent, a multi-agent framework for end-to-end CFD automation in OpenFOAM. First, where agentic designs range from decentralized systems (Wang, Marom, Pal, Luu, Lu, Berkovich and Buehler, 2026) that rediscover the workflow at run time to general-purpose coding agents that write solver inputs directly, Foam-Agent instead binds each of six specialist agents to a canonical OpenFOAM stage: because the workflow is well understood and stable, encoding it explicitly lets each agent specialize rather than spend reasoning budget rediscovering the pipeline. Second, Foam-Agent is built as domain-level *harness engineering*, the design of the agent loop, tool interface, and context engineering around the model, so that its contributions are properties of the agent design rather than of any single transport layer and survive a change of retrieval backend or orchestration protocol. On this foundation we introduce three methodological advances for automated CFD workflow setup with LLM agents:

- **Multi-index retrieval** organizes OpenFOAM domain knowledge along four complementary structural dimensions (case metadata, directory structure, file contents, and execution scripts) and selects the index by current workflow stage. This improves retrieval precision over a flat single-index design.

- **Dependency-aware file generation as a topological traversal** of the OpenFOAM case dependency graph: each file is generated in the verbatim context of its already-emitted predecessors, which enforces cross-file consistency and suppresses the hallucination of undefined symbols.

- **Trajectory-conditioned reviewer loop**: an execution-grounded self-correction mechanism that diagnoses solver errors against the accumulated trajectory of its own previous attempts and applies a minimal configuration edit targeting a reduced error set on the next run.

These three advances are situated at the workflow layer: they concern how retrieval, file generation, and error correction are organized around the solver, rather than the numerical methods, discretization, or mesh technology inside it, which

Foam-Agent uses as provided by OpenFOAM. The remaining features of the framework, namely the Model Context Protocol (MCP) (Anthropic, 2024) interface, HPC execution support, external-mesh ingestion, and post-processing visualization, are engineering capabilities that broaden its practical use rather than methodological contributions. In this sense Foam-Agent is complementary to recent agentic work in computational mechanics that operates at other layers, such as the LLM-empowered CAE agent for model order reduction (Guo et al., 2026) and the fine-tuned finite-element system ALL-FEM (Deotale et al., 2026), which advance the solver and model layers that the present workflow layer takes as given.

The remainder of this paper is organized as follows. Section 2 describes the Foam-Agent workflow, including the agent components, dependency-aware file generation, multi-index retrieval, and MCP interface. Section 3 presents the experimental evaluation, covering overall performance, ablation studies, supported workflow capabilities, and case studies. Section 4 concludes the paper.

2. Methods

2.1. Agent Components

Algorithm 1 Foam-Agent orchestration protocol; the Reviewer call is expanded in Algorithm 4.

Require: Natural language requirement R, maximum iterations M

Symbols: $\Pi = \{T_1, \dots, T_n\}$ is the file-generation plan produced by the Architect, where each task $T_i = (f_i, \rho_i, \sigma_i)$ specifies a target file path, a generation priority, and retrieved schematic constraints; $\mathcal{H} = [(\mathcal{E}_1, d_1), \dots, (\mathcal{E}_{t-1}, d_{t-1})]$ is the append-only history of error sets and Reviewer diagnoses over correction iterations.

Ensure: Validated simulation configuration $\mathcal{S}^*$, Visualization V

$(\Pi, \mathcal{K}) \leftarrow \text{Architect}(R)$ ▷ Plan files; retrieve reference $\mathcal{K}$
$\mathcal{M} \leftarrow \text{MeshingAgent}(R)$ ▷ Generate mesh (Native/Gmsh/External)
$\mathcal{S}_0 \leftarrow \text{InputWriter}(\Pi, \mathcal{M}, \mathcal{K})$ ▷ Generate initial case files
$\mathcal{H} \leftarrow [\,]$ ▷ Initialize iteration history
for $t \leftarrow 1$ to M **do**
  $\mathcal{L}_t, \text{status} \leftarrow \text{Runner}(\mathcal{S}_{t-1})$ ▷ Execute simulation and capture logs
  if status = SUCCESS **then**
    $V \leftarrow \text{VisualizationAgent}(\mathcal{S}_{t-1}, R)$
    return $\mathcal{S}_{t-1}, V$
  end if
  $\mathcal{E}_t \leftarrow \Phi(\mathcal{L}_t)$ ▷ Extract solver errors (Algorithm 3)
  $(\Delta\mathcal{S}_t, d_t) \leftarrow \text{Reviewer}(\mathcal{E}_t, \mathcal{S}_{t-1}, \mathcal{H}, \mathcal{K})$ ▷ Trajectory-conditioned correction (Algorithm 4)
  $\mathcal{S}_t \leftarrow \mathcal{S}_{t-1} \oplus \Delta\mathcal{S}_t$ ▷ Apply configuration edit
  $\mathcal{H} \leftarrow \mathcal{H} \,\|\, [(\mathcal{E}_t, d_t)]$ ▷ Append-only history
end for
return FAILURE

2.1.1. *Architect Agent*

The Architect Agent translates a natural-language description into a structured, dependency-ordered file-generation plan through an explicit five-step pipeline, employing a RAG paradigm together with Pydantic (Colvin, Jolibois, Ramezani, Garcia Badaracco, Dorsey, Montague, Matveenko, Trylesinski, Runkle, Hewitt, Hall and Plot, 2025) schemas for typed, validated intermediate representations. The pipeline runs in five steps: (i) *classification* of the prompt into a four-field Pydantic schema; (ii) *template selection* by FAISS retrieval over the multi-index (Section 2.2), a two-step query that returns the top-k structurally matching tutorials and then re-queries the tutorial-details index for the full file contents of the top-ranked match; (iii) *file-manifest* decomposition of the retrieved tutorial into an explicit list of `(file_name, folder_name)` subtask pairs, augmented with case-specific files when the classification indicates them; (iv) *dependency layering* along the acyclic OpenFOAM `system` ≻ `constant` ≻ `0` ≻ `Allrun` directory order, which every tutorial obeys; and (v) *emission*, in which the Input Writer (Section 2.1.3) generates the files in this layer order, each in the context of its already-emitted predecessors.

The classification schema has four categorical fields: `case_name`, a free-form identifier; `case_domain`, one of the 14 OpenFOAM tutorial-tree categories (`basic`, `combustion`, `compressible`, `discreteMethods`, `DNS`, `electromagnetics`, `financial`, `heatTransfer`, `incompressible`, `lagrangian`, `mesh`, `multiphase`, `resources`, `stressAnalysis`); `case_category`, a solver-family identifier; and `case_solver`, the specific solver name. The latter two fields are restricted at run time to values harvested from the tutorial corpus, and schema-level validation rejects malformed model output and re-prompts with a stricter directive. Turbulence class and time dependence (steady or transient) are not separate top-level fields; they are fixed implicitly by the tutorial retrieved in step (ii).

Formally, the Architect maps the user-requirement space $\mathcal{R}$ to a structured task space $\Pi = \{T_1, T_2, \dots, T_n\}$, where n is the number of files to be generated and each task T_i is a tuple (f_i, ρ_i, σ_i) with target file path f_i, generation priority ρ_i, and retrieved schematic constraints σ_i. This decomposition guarantees that subsequent generation respects the topological dependencies of the OpenFOAM case structure.

As a worked example, consider the request *"simulate a laminar incompressible lid-driven cavity at $Re = 100$"*. Step (i) classifies it with `case_name = lidDrivenCavity`, `case_domain = incompressible`, and `case_solver = icoFoam`. Step (ii) retrieves the `cavity/icoFoam` tutorial as the top-1 match. Step (iii) produces the manifest {`controlDict`, `fvSchemes`, `fvSolution`, `transportProperties`, `blockMeshDict`, `0/U`, `0/p`, `Allrun`}. Step (iv) assigns these to four layers (layer 0 = {`controlDict`, `fvSchemes`, `fvSolution`, `blockMeshDict`}, layer 1 = {`transportProperties`}, layer 2 = {`0/U`, `0/p`}, and layer 3 = {`Allrun`}) and step (v) emits them in that order.

2.1.2. Meshing Agent

The Meshing Agent supports four mesh-generation pathways and autonomously selects one according to the user requirement.

- `blockMesh`. Structured, block-structured hexahedral meshes generated from a `blockMeshDict` that the agent writes; the agent then runs `blockMesh` to produce the `polyMesh` folder. This pathway is fully autonomous.
- `snappyHexMesh`. Unstructured, body-fitted, hex-dominant meshes around STL geometries, generated from an agent-written `snappyHexMeshDict`; this pathway requires a watertight STL surface.
- Gmsh (Geuzaine and Remacle, 2009). Structured or unstructured meshes for complex geometries absent from standard tutorials. From a natural-language description of the physical domain and boundary names, the agent writes a Python script that builds the geometry and mesh through the Gmsh library, produces a `.msh` file, and converts it to OpenFOAM `polyMesh` format with `gmshToFoam`.
- External `.msh`. A user-supplied, externally generated mesh is converted in place with `gmshToFoam`, after which the rest of the pipeline proceeds unchanged. The current external pathway is scoped to the Gmsh `.msh` format; additional external formats would require conversion-specific boundary-patch validation before being treated as fully supported.

Meshing for complex geometries is an open problem in CFD, and the Meshing Agent integrates existing tools rather than introducing a new meshing algorithm; it therefore inherits each tool's limits. The external-`.msh` pathway in particular is not a technical novelty but a deliberate human–AI design boundary: when the in-the-loop meshing tools fall short for a given geometry, the user can supply an expert mesh directly and the remaining agents continue without modification.

2.1.3. Input Writer Agent

The Input Writer Agent realizes the dependency-aware file generation that is the second methodological contribution of this work. It emits the planned files as a topological traversal of the OpenFOAM case dependency graph $G = (V, E)$, where the vertices V are configuration files and the edges E are parameter dependencies. For example, files in the `0` directory depend on physical-property files in `constant`, which in turn depend on solver configuration in `system`. Because the OpenFOAM directory hierarchy fixes this layering (`system` $\succ$ `constant` $\succ$ `0` $\succ$ `Allrun`), the traversal is realized by a stable four-tier priority sort over folder names rather than by re-deriving the graph at run time (Algorithm 2).

For the i-th file the Architect supplies a task specification $T_i = (f_i, \rho_i, \sigma_i)$, and the file content C_i is generated as

$$C_i = \mathrm{LLM}\big(T_i,\ \mathcal{K},\ \{C_j : j \prec i\}\big), \tag{1}$$

Algorithm 2 Input Writer: fixed-layer topological emission

Require: Planned task list Π; retrieved tutorial reference $\mathcal{K}$

function PRIORITY(T_i)
    if FOLDER(T_i) = `system` **then**
        return 0
    else if FOLDER(T_i) = `constant` **then**
        return 1
    else if FOLDER(T_i) = `0` **then**
        return 2
    else
        return 3
    end if
end function
$\sigma \leftarrow$ STABLESORT(Π), keyed by PRIORITY ▷ topological order
$\mathcal{C} \leftarrow [\,]$ ▷ accumulator of emitted files
for $T_i \in \sigma$ **do**
    $C_i \leftarrow \mathrm{LLM}_{\mathrm{Writer}}(T_i,\ \mathcal{K},\ \mathcal{C})$ ▷ $\mathcal{C}$: emitted predecessors
    $\mathcal{C} \leftarrow \mathcal{C} \,\|\, [C_i]$
end for
$A \leftarrow \mathrm{LLM}_{\mathrm{Allrun}}(\mathcal{C},\ \mathcal{K},$ COMMANDHELP$)$ ▷ second LLM call: Allrun script
return $\mathcal{C} \,\|\, [A]$

where $j \prec i$ denotes the files emitted before i in topological order. More generally, the prompt assembled for each LLM call is the concatenation of four distinct components (the complete agent prompts are listed in Section A),

$$\mathrm{prompt}_i = T_i \oplus \mathcal{K} \oplus \{C_j : j \prec i\} \oplus \mathcal{H}, \tag{2}$$

in which T_i is the file task specification; $\mathcal{K}$ is the *retrieved knowledge*, namely the top-ranked tutorial reference returned by the multi-index retrieval module of Section 2.2 (shared across all Input Writer calls for a case); $\{C_j : j \prec i\}$ is the verbatim content of the already-emitted predecessor files; and $\mathcal{H}$ is the iteration history, which is empty during the initial Input Writer pass and grows only over Reviewer iterations (Section 2.1.5). Injecting the predecessor files *verbatim*, rather than as paraphrased summaries, anchors patch names, the solver name, and transport-property entries to the exact upstream tokens, which enforces parameter continuity and suppresses the hallucination of undefined symbols. Once all case files are emitted, a second LLM call generates the `Allrun` execution script from the accumulated files and the command documentation. Because the agents operate on configuration files rather than mesh data, the per-call input context remains small in practice, averaging 9–19k tokens on the benchmark (Section 3.7), so context-window overflow does not arise and no context compaction is required.

2.1.4. Runner Agent

The Runner Agent interfaces with the OpenFOAM execution environment: it prepares the simulation (cleaning artifacts, setting up output capture) and executes the `Allrun` script either on the user's *local machine* or on an *HPC cluster* (Figure 6), as specified in the request. For HPC runs the agent generates a Slurm script, submits the job with the provided account, and monitors it to completion (OpenFOAM High Performance Computing Technical Committee, 2025); partition, node count, and processes-per-node are taken from the prompt or inferred from the mesh decomposition.

At the end of a run the Runner extracts solver errors for the Reviewer. This error detection is the pattern-matching function $\Phi : L \rightarrow \mathcal{E}$ that maps the execution logs L to a set of structured error records $\mathcal{E}$. Φ is *not* backed by a preloaded error catalog: it is a single-regex extractor that captures OpenFOAM's canonical fatal-error prefix from each solver log file, leaving the *classification* of the captured fragment to the LLM Reviewer (Algorithm 3).

Coverage. OpenFOAM standard solvers abort through the `FatalErrorIn` and `FatalIOErrorIn` macros, which emit a line containing the canonical `ERROR:` substring into the solver log. This single substring captures the diagnosable evidence for the dominant failure classes: dictionary syntax errors, missing-file errors, boundary-condition mismatches,

Algorithm 3 Runner: canonical-prefix error extraction

Require: Case directory D; canonical regex `r = re.compile(r"ERROR:(.*)", re.DOTALL)`

$\mathcal{E} \leftarrow \emptyset$
for each log file ℓ in D (file name with prefix `log`) **do**
  $c \leftarrow \textsc{Read}(\ell)$
  if `r` matches c **then**
    $e \leftarrow$ matched fragment after `ERROR:`
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\ell, e)\}$
  end if
end for
return $\mathcal{E}$

and thermophysical-bound aborts. Two classes systematically bypass the prefix: runner-script-level failures that abort before solver invocation (e.g. `mpirun` option errors), and bare floating-point exceptions whose stack trace contains no `ERROR:` line.

Update mechanism. Because there is no catalog, there is nothing to maintain: any new OpenFOAM error type is supported automatically as long as the solver emits the canonical `ERROR:` prefix, after which the LLM Reviewer classifies and repairs the fragment with no agent-side code change. Keeping the runner-side surface deliberately narrow lets the LLM absorb the open vocabulary of solver-specific error wordings.

Failure modes. Errors that bypass the `FatalErrorIn` macros (`mpirun` option errors, `decomposePar` configuration failures, and bare floating-point-exception traces) fall outside this extractor and are not diagnosable from the solver log alone. When a log contains several `ERROR:` fragments, the `re.DOTALL` flag returns one contiguous match per log (the tail starting at the first occurrence), which the Reviewer then decomposes internally during diagnosis.

2.1.5. Reviewer Agent

The Reviewer Agent drives the execution-grounded correction loop that is the third methodological contribution of this work. Within each orchestration iteration, indexed by the iteration counter t, when a run fails, the Reviewer node is invoked as three sequential LLM calls (Algorithm 4): an *analysis* call produces a diagnosis d_t from the extracted error set $\mathcal{E}_t$; a *plan* call turns the diagnosis into a minimal rewrite plan P_t, the smallest set of file changes that targets a reduced error set on the next run; and a *rewrite* call realizes the plan into the configuration edit $\Delta\mathcal{S}_t$ (a set of file changes, not an OpenFOAM boundary patch). The orchestration loop (Algorithm 1) then re-executes the case and repeats until the error set is empty or the iteration cap M is reached.

Algorithm 4 Reviewer node: trajectory-conditioned correction step

Require: Error set $\mathcal{E}_t$; case state $\mathcal{S}_{t-1}$; history $\mathcal{H}$; retrieved reference $\mathcal{K}$
Ensure: Configuration edit $\Delta\mathcal{S}_t$; diagnosis d_t

$d_t \leftarrow \mathrm{LLM}_{\mathrm{Analyze}}(\mathcal{E}_t,\ \mathcal{S}_{t-1},\ \mathcal{K},\ \mathcal{H})$ ▷ diagnose the error set
$P_t \leftarrow \mathrm{LLM}_{\mathrm{Plan}}(\mathcal{E}_t,\ \mathcal{S}_{t-1},\ d_t)$ ▷ minimal rewrite plan: smallest file-change set
$\Delta\mathcal{S}_t \leftarrow \mathrm{LLM}_{\mathrm{Rewrite}}(\mathcal{S}_{t-1},\ P_t)$ ▷ realize the plan into corrected files
return $(\Delta\mathcal{S}_t,\ d_t)$

We describe this loop as a trajectory-conditioned reviewer loop: across iterations it conditions each correction on the accumulated trajectory of its own previous attempts, the error logs and diagnoses recorded in the history $\mathcal{H}$ (the *Iteration history* paragraph below), so that it learns from and avoids its earlier failed strategies, an execution-grounded form of self-reflection (Shinn, Cassano, Labash, Gopinath, Narasimhan and Yao, 2023). Within a single iteration the correction itself is kept minimal: the plan call is instructed to propose the smallest rewrite set, and the rewrite call to return only files that require modification or addition, so the per-step configuration edit $\Delta\mathcal{S}_t$ (a set of file changes, not an OpenFOAM boundary patch) targets a reduced error set with the fewest changes. The per-step edit is chosen greedily with respect to the current error set $\mathcal{E}_t$, while the diagnosis that produces it is conditioned on the full trajectory $\mathcal{H}$; the error reduction it targets is an *empirical* target verified by re-execution rather than a guaranteed monotone descent, because an OpenFOAM error can be masked by an earlier failure and surface only after an edit is applied. The term “optimization” here refers to this minimal-edit objective and re-execution target, not to a numerical solver with an analytic convergence guarantee.

Corrective-action repertoire. The Reviewer is not restricted to time-step reduction; it can issue five categories of corrective action: *(i)* boundary-condition correction, changing a patch type or value (for example, replacing a plain wall with a constraint-type wall when a buoyant solver requires it); *(ii)* discretization-scheme swaps in fvSchemes (for example, Gauss linear → Gauss limitedLinear for shock-dominated cases); *(iii)* time-step and Courant-number adjustment via controlDict; *(iv)* mesh regeneration, re-invoking the Meshing Agent (including refinement-level changes for blockMesh and snappyHexMesh) when checkMesh flags excessive skewness, non-orthogonality, or aspect ratio; and *(v)* solver or turbulence-model replacement (for example, pisoFoam → simpleFoam for a steady case). One scope clarification follows: the current loop can trigger mesh regeneration and mesh-parameter adjustment, but it does not modify the CAD or geometry definition itself in response to a downstream solver failure; closed-loop geometry modification is left as a complex-geometry extension.

Iteration history. To diagnose against earlier attempts, the Reviewer maintains an append-only history $\mathcal{H}$. At each iteration t, an <Error_Logs> block holding $\mathcal{E}_t$ and a <Review_Analysis> block holding the diagnosis d_t are appended, both wrapped in an <Attempt> element. On each Reviewer call the accumulated list is concatenated and wrapped in a <history> block alongside the current error logs, the current case files, and the retrieved tutorial reference. The history is kept *flat and append-only*, rather than digested into a summary, because Reviewer trajectories frequently close non-monotonically: a single fix can expose a previously masked downstream error, so the outstanding error set can grow at intermediate steps even when the run eventually succeeds. In such trajectories the Reviewer needs the full record of which diagnoses were already tried, so it does not re-apply a failed strategy, and the full error-log trace, so it can recognize a re-emerging error pattern, exactly the evidence that a digest would compress away.

2.1.6. Visualization Agent

After the Runner produces a successful run, Foam-Agent checks whether the prompt requested a visualization. If so, the Visualization Agent identifies the target field variables and generates a Python script (using either the PyVista (Sullivan and Kaszynski, 2019) library or ParaView Python routines (Ayachit, 2015), at the user’s choice) to render them (Figure 1). It then executes the script and, like the Reviewer, repairs any errors until the visualizations are saved as PNG files in the run directory, up to a user-configurable retry limit.

2.2. Multi-Index Retrieval

Algorithm 5 Multi-index retrieval with deterministic reranking

Require: Query q with classification tuple; workflow stage s; recall depth k (default 3)
Ensure: Retrieved tutorial reference $\mathcal{K}$

$\mathcal{I}_s \leftarrow \textsc{SelectIndex}(s)$ ▷ stage-specific FAISS index
$\mathcal{D} \leftarrow \textsc{TopK}\big(\mathcal{I}_s,\ \textsc{Embed}(q),\ k\big)$ ▷ top-k recall; no similarity cutoff
sort $\mathcal{D}$ ascending by the lexicographic key
    $\big(-[\textsc{domain match}],\ -[\textsc{solver match}],\ -\textsc{cosine}\big)$
$\mathcal{K} \leftarrow$ file contents of the top-1 candidate of sorted $\mathcal{D}$
return $\mathcal{K}$

The first methodological contribution of Foam-Agent is a multi-index retrieval system that segments OpenFOAM domain knowledge into specialized indices, each optimized for a specific phase of the simulation workflow. Compared with a conventional flat single-index RAG system, this structural decomposition sharpens retrieval precision; the gain is quantified directly by the ablation of Section 3.3.

Retrieval algorithm. An earlier prototype of this module filtered candidates with a cosine-similarity cutoff τ. Ahead of the released implementation we replaced it with the calibration-free procedure of Algorithm 5, because a fixed τ required per-solver tuning that scaled poorly across the heterogeneous OpenFOAM case-family distribution: it over-filtered on rare solver families (returning no candidate) and under-filtered on common ones (returning structurally distant candidates). The current algorithm exposes only two semantically meaningful knobs, both pinned in code:

a top-k recall depth ($k = 3$ by default) and a deterministic lexicographic rerank. Given the Architect's four-field classification, FAISS returns the top-k candidates from the stage-selected index; these are reranked by the key $\big(-[$`case_domain` match$]$, $-[$`case_solver` match$]$, $-$cosine$\big)$, where $[\cdot]$ is the Iverson bracket (one if the enclosed predicate holds, zero otherwise). The cosine score therefore acts only as a tiebreaker after the exact-match flags on `case_domain` and `case_solver`, and is never compared against a numeric cutoff. The top-1 reranked candidate is selected unconditionally and its file contents become the retrieved reference $\mathcal{K}$. The operative sensitivity question is thus the structural decomposition itself (multi-index versus flat single index), studied in Section 3.3; a sweep of the discrete recall depth $k \in \{1, 3, 5\}$ is left as a follow-up experiment.

2.2.1. Knowledge Base Construction

The knowledge base is built from the OpenFOAM tutorial corpus through a five-stage pipeline: (i) *source* selection of the tutorial corpus; (ii) *structural parsing* of each case into typed dictionary triples; (iii) *per-dimension extraction* across four complementary structural dimensions; (iv) *category tagging* of each tutorial; and (v) *indexing* into four FAISS indices. The source is all OpenFOAM Foundation v10 tutorials ($\approx$ 200 cases) under `$WM_PROJECT_DIR/tutorials`, spanning the major solver families (incompressible, compressible, multiphase, reactive, heat transfer, magneto-hydrodynamic), turbulence regimes (laminar, RAS, LES), and mesh-generation pathways. A lightweight parser walks each case directory and extracts `(key, value, type)` triples from the OpenFOAM dictionary files via top-level brace matching; it is targeted rather than a full grammar over the OpenFOAM dictionary language, capturing exactly the patterns the four retrieval dimensions require. The four dimensions are *case metadata* (the solver name from `controlDict`, physics tags from the `Allrun` header, and the turbulence model from `constant/momentumTransport`), *directory structure* (a bag-of-paths over the case file tree), *file contents* (full file text together with the parsed triples), and *execution scripts* (the `Allrun` command sequence). The four dimensions are correlated rather than independent, since a case's directory structure, file contents, and execution script all reflect its solver family. They are complementary in the sense that each serves a different retrieval stage: the design goal is this separation of concerns, so that each stage queries the index whose granularity matches its query, rather than a decomposition into statistically independent factors. Each tutorial is then annotated with the four categorical fields `case_name`, `case_domain`, `case_category`, and `case_solver`; the union of observed values is aggregated into a statistics file that constrains the Architect's run-time classification to values that actually occur in the tutorial tree, in an LLM-assisted pass with expert review. Finally, the four dimensions populate four FAISS (Douze, Guzhva, Deng, Johnson, Szilvasy, Mazaré, Lomeli, Hosseini and Jégou, 2025) indices, one per dimension, queried at recall depth $k = 3$; embeddings are computed locally with the Qwen3-Embedding-0.6B model, which removes the external embedding-API dependency without altering the indexing architecture. The four indices serve distinct roles: the Tutorial Structure Index identifies structural templates; the Tutorial Details Index supplies boundary conditions, numerical schemes, and physical models; the Execution Scripts Index supplies command sequences; and the Command Documentation Index supplies utility usage and parameter selection.

2.3. The Model Context Protocol Deployment Surface

Beyond the three methodological contributions above, Foam-Agent exposes all of its agent functions through the Model Context Protocol (Anthropic, 2024) so that external orchestrators can consume it as a composable component (Ouyang, Yue, Di, Zheng, Yue, Pan, Yin and Zhang, 2026). We are explicit that MCP is a *deployment surface* (an ecosystem-participation choice) rather than a research contribution: nothing inside Foam-Agent depends on MCP, and the three contributions of Section 2 hold independently of it.

What MCP provides is interface unification. Standard function calling could expose the same agent functions, but reaching each orchestrator would then require a bespoke adapter; MCP collapses this to a single server, published once and consumed by any MCP-capable client. For a single, fixed orchestrator, MCP offers little beyond ordinary function calling; its value emerges across orchestrators, where one self-describing server replaces the separate adapter that each additional client would otherwise require. In our deployment the same unmodified server is driven by both Cursor and Claude Code. This is the interoperability rationale that standardized formats already serve elsewhere in computational science, where CGNS (cgn, 2005) standardizes the exchange of mesh and solution data and preCICE (Bungartz, Lindner, Gatzhammer, Mehl, Scheufele, Shukaev and Uekermann, 2016) standardizes black-box solver coupling, so that independently developed tools interoperate without pairwise adapters.

The function surface (atomic, stateful, workflow-decoupled functions identified by `case_id` and `job_id`) is detailed in Section D. When the orchestrator itself drives the call sequence, we implement it as a stateful LangGraph (LangChain Inc.) graph whose nodes are MCP calls and whose edges are conditional transitions, with Pydantic (Colvin et al., 2025) schemas enforcing typed, validated I/O at every boundary.

Foam-Agent in the agentic-AI ecosystem. Foam-Agent is designed to be a plug-and-play CFD component for higher-level scientific-discovery ecosystems such as ScienceClaw (Wang et al., 2026). In our deployment a *Skill* is layered on top of the MCP server: the Skill is a prompt-level wrapper that supplies an orchestrator with task context and usage conventions, while the underlying tool surface remains the MCP-exposed agent functions. The two layers serve different purposes (MCP is the tool-exposure protocol, the Skill is a higher-level prompt-plus-tool composition), and either can be consumed by orchestrators such as Cursor, Claude Code, or ScienceClaw. One consequence is important: when Foam-Agent is consumed as a component, the orchestrator inherits the underlying capabilities (planning, meshing, file writing, execution, review, visualization) but *not* the workflow design itself: planning and the iterative correction loop migrate from Foam-Agent into the external orchestrator.

In that setting the iteration policy is owned by the orchestrator rather than by Foam-Agent: a strong orchestrator repeatedly invokes the repair function until the case completes, whereas a weaker one may exit prematurely, and standalone Foam-Agent retains a deterministic loop up to its configured iteration cap. A measured MCP-versus-no-MCP performance ablation would be uninformative by construction: the MCP wrapper invokes the same in-process agent functions with the same arguments, so the OpenFOAM trajectory and the metrics of Section 3 are identical with or without it; the substantive difference is integration cost, namely the number of orchestrator-specific adapters, not solution quality. Protocol-level details, namely per-call latency and the failure modes at the protocol boundary, are deferred to Section D.

3. Results

We evaluate Foam-Agent on FoamBench, the end-to-end OpenFOAM simulation benchmark of the CFDLLM-Bench suite (Somasekharan, Yue, Cao, Li, Emami, Bhargav, Acharya, Xie and Pan, 2026c). FoamBench is organized into a *Basic* tier of 110 cases across 11 physics scenarios and a harder *Advanced* tier of 16 hand-crafted cases. Each case is specified by a natural-language prompt giving the problem description, physical scenario, geometry, solver, boundary conditions, and simulation parameters.

Evaluation metrics. Because crash-free execution and physical correctness are distinct properties, we report two precise metrics throughout, following the CFDLLMBench convention (Somasekharan et al., 2026c).

- $M_{\text{exec}} \in \{0, 1\}$ is the *execution-success* metric: it is 1 if and only if the OpenFOAM solver returns exit code 0, writes a final time step, and produces no NaN or Inf in the residuals (the "End" marker in the solver log).
- $M_{\text{NMSE}} \in \{0, 0.5, 1\}$ is a *bucketed field-level fidelity* metric: the per-case normalized mean-squared error (NMSE) against the ground-truth solution is scored 1 if NMSE ≤ 0.1, 0.5 if $0.1 <$ NMSE ≤ 0.3, and 0 otherwise. NMSE is the squared L_2 norm of the field deviation normalized by the squared L_2 norm of the ground-truth field, computed field-wise ($\boldsymbol{U}$, p, ρ, T where present) and averaged.

We define *complete success* as $M_{\text{exec}} \wedge (M_{\text{NMSE}}{=}1)$. Byte-level identity to a human-expert reference is explicitly *not* the success criterion: two OpenFOAM configurations can be syntactically distinct yet numerically equivalent in the regime of interest (for example, `fixedValue` versus `codedFixedValue` for a constant boundary condition, or two distinct stable time steps), and the NMSE-based metric correctly credits these as equivalent. Every headline M_{exec} figure below is co-reported with its M_{NMSE}.

3.1. Baselines and Evaluation Setup

We compare Foam-Agent against MetaOpenFOAM (Chen et al., 2025), a representative multi-agent framework for automating OpenFOAM workflows. To separate the contribution of agentic orchestration from raw LLM capability, we additionally include a *zero-shot LLM* baseline (a single LLM call with no agents, retrieval, or Reviewer loop) as a framework-free lower bound. Both frameworks are evaluated with Claude 3.5 Sonnet (Anthropic, 2024) and GPT-4o (Achiam, Adler, Agarwal, Ahmad, Akkaya, Aleman, Almeida, Altenschmidt, Altman, Anadkat et al., 2023); unless stated otherwise, reported numbers use a matched Claude 3.5 Sonnet backbone, so that differences reflect framework design rather than the underlying model.

Table 1
Execution success (M_{exec}) and bucketed field-level fidelity (M_{NMSE}) on FoamBench under the matched-backbone setting. Higher is better for both metrics.

System	Tier	M_{exec}	M_{NMSE}
Foam-Agent	Basic (n=110)	**0.882**	**0.477**
Foam-Agent	Advanced (n=16)	**0.625**	**0.406**
MetaOpenFOAM	Basic (n=110)	0.555	0.173
MetaOpenFOAM	Advanced (n=16)	0.125	0.125
Zero-shot LLM	Basic (n=110)	0.064	0.050
Zero-shot LLM	Advanced (n=16)	0.017	0.009

Knowledge-base and benchmark distributional overlap. Both the Foam-Agent knowledge base and the FoamBench-Basic cases derive from the OpenFOAM tutorial corpus, so a distributional overlap exists and must be discussed explicitly. Three points bound its effect. First, retrieval returns *structural templates, not solutions*: the agents must still adapt a retrieved tutorial to the requested geometry, boundary conditions, and physical parameters, and the Reviewer must still close the residual gap. The ablation of Section 3.3 shows that retrieval alone caps M_{exec} at 57.3% on FoamBench-Basic. Second, the 16 FoamBench-Advanced cases were hand-crafted by the CFDLLMBench authors and are *not* present in our knowledge base, so the Advanced-tier results are out-of-distribution (OOD) evidence. Third, the non-tutorial case studies of Section 3.4 and Section 3.8 (user-supplied geometries and a multi-physics burner) provide further non-tutorial evidence.

3.1.1. Extended Baseline Comparison

Table 1 reports M_{exec} and M_{NMSE} for Foam-Agent, MetaOpenFOAM, and the zero-shot LLM baseline on both tiers under the matched-backbone setting. The zero-shot row is the framework-free lower bound and isolates the agentic-orchestration contribution from raw LLM capability: Foam-Agent's M_{exec} is roughly 14× the zero-shot value on Basic and 37× on Advanced, whereas MetaOpenFOAM is 7–9×. Foam-Agent leads MetaOpenFOAM by +0.327 M_{exec} and +0.304 M_{NMSE} on Basic, and by +0.500 and +0.281 on Advanced.

Because the OpenFOAMGPT (Pandey et al., 2025) source code is not publicly available, it cannot be re-run under the FoamBench protocol, and its published evaluation (six hand-picked tutorials scored by a binary success criterion) is not directly portable to our metrics. To approximate its dependency-handling behavior we ran an *OpenFOAMGPT-shaped proxy*, our pipeline with sequential, dependency-aware file generation replaced by parallel generation (no predecessor context), while retaining the multi-index retrieval and the Reviewer, which scores 0.845 M_{exec} / 0.409 M_{NMSE} on FoamBench-Basic. Full Foam-Agent leads this proxy by +0.037 M_{exec} and +0.068 M_{NMSE}, isolating the contribution of dependency-aware generation. Beyond the experimental baselines above, AutoFEA-like systems (Hou et al., 2025) and traditional scripting pipelines lack adaptive retrieval, natural-language-to-domain classification, and iterative repair, and are therefore not directly comparable on the agentic-correctness axis this benchmark measures.

3.2. Overall Performance Comparison

Figure 2 and Table 1 present the comparative performance. Foam-Agent substantially outperforms the baseline across all tested backbones. With Claude 3.5 Sonnet, Foam-Agent reaches an M_{exec} of 88.2% on FoamBench-Basic, significantly surpassing MetaOpenFOAM's 55.5%; the gap is even larger with GPT-4o, where Foam-Agent reaches 59.1% against the baseline's 17.3%. Because crash-free execution is not the same as physical correctness, we co-report the bucketed field-level fidelity metric: on FoamBench-Basic Foam-Agent attains $M_{\text{NMSE}} = 0.477$, against 0.173 for MetaOpenFOAM.

Execution success versus solution fidelity. The headline 88.2% M_{exec} and the co-reported $M_{\text{NMSE}} = 0.477$ are reconciled by the bucket distribution. The 110 Basic-tier cases distribute as 46/13/51 over the fidelity buckets $\{1, 0.5, 0\}$, so the *strict* complete-success rate is $46/110 = 41.8\%$, still well above the upper bound that MetaOpenFOAM's $M_{\text{NMSE}} = 0.173$ places on its own complete-success rate. The 51 bucket-0 cases decompose into two distinct failure modes: 13 are execution failures (NMSE undefined, scored 0 by convention), while the remaining 38 *ran to completion but produced a field-level NMSE above* 0.3. This exec-pass / fidelity-fail population is the operative residual gap; it

Table 2
2×2 Reviewer × RAG ablation on FoamBench-Basic (n=110, LLM temperature T=0.6).

Reviewer	RAG variant	M_{exec}
off	single index	0.445
off	multi-index	0.573
on	single index	0.845
on	multi-index	**0.882**

is analyzed case by case in the failure taxonomy of Section 3.3.3, where it is found to be dominated by solver- and parameter-setup errors rather than by file-generation or execution faults.

To complement these aggregate metrics with a qualitative view, we visually compare the outputs against ground-truth solutions for three representative cases from FoamBench: *CounterFlowFlame*, *wedge*, and *forwardStep* (Figure 2b–d). In the *CounterFlowFlame* case (Figure 2b), which depicts the CH_4 mass fraction at $t = 0.5$ s, Foam-Agent accurately reproduces the sharp concentration gradients characteristic of flame fronts, whereas the baseline yields a diffuse, inaccurate transition region. Similarly, for the *wedge* case (Figure 2c), Foam-Agent captures the correct temperature field near the angled wall, while the competing framework fails to reconstruct even the fundamental geometry of the domain. Finally, in the *forwardStep* scenario (Figure 2d), the velocity-magnitude field generated by Foam-Agent is virtually indistinguishable from the ground truth, whereas the baseline predicts erroneously low velocities throughout the domain. These results underscore the framework's capability to simulate canonical problems with high precision.

3.3. Ablation Studies

Having shown that Foam-Agent outperforms the baseline, we examine the contribution of each component, focusing on two components: the Reviewer and dependency-aware file generation. Dependency-aware generation orders file creation so that base files precede the dependent files that reference them (Figure 1). All ablation experiments use Claude 3.5 Sonnet (Anthropic, 2024), and Figure 3 summarizes the results. The Reviewer is the single most important factor: it raises the execution-success rate from roughly 50% to over 80% across all configurations. This points to iterative, execution-grounded self-correction (Shinn et al., 2023) as the decisive mechanism. We repeat the experiment at two LLM temperatures (T) to confirm that this holds. With the Reviewer disabled, dependency-aware generation gives the largest single improvement in execution success: from 48.2% to 56.4% at $T = 0.0$ and from 45.4% to 57.3% at $T = 0.6$. With the Reviewer enabled the difference is small, because the loop corrects most initial-generation errors regardless; the main benefit of dependency-aware generation is then faster convergence: it lowers the mean Reviewer-iteration count (from 0.90 to 0.79 at $T = 0.0$, and from 1.87 to 0.96 at $T = 0.6$) and thus the API cost and runtime.

The ablation study on Foam-Agent's RAG system is summarized in Figure 3. We first performed an experiment without the reviewer to isolate the effect of multi-index retrieval, which uses a multi-index of case metadata (e.g., case name, domain, category, solver) to retrieve precise context. This multi-index approach outperforms single-index approaches: it reduces noise and improves retrieval precision, narrowing the semantic gap between natural language and technical terminology.

Reviewer × RAG ablation. To isolate the structural-retrieval signal from the iterative-repair loop, we ran a 2×2 ablation on FoamBench-Basic (n=110) crossing the Reviewer (on/off) with the RAG variant (flat single index versus multi-index); Table 2 reports the result. Under Reviewer-off, the multi-index lifts M_{exec} by +12.8 percentage points (44.5% → 57.3%) over the flat baseline, a direct measure of the structural-decomposition contribution, independent of the repair loop. Under Reviewer-on this narrows to +3.7 points (84.5% → 88.2%), consistent with the repair loop absorbing most retrieval-quality variance. Retrieval and iterative repair are thus complementary: retrieval alone caps M_{exec} at 57.3%, and the Reviewer is what carries the system to 88.2%.

3.3.1. Retrieval Case Studies

Two concrete cases, each run with ten LLM-paraphrased prompts, illustrate the two regimes the ablation aggregates. *Successful retrieval* (`BenardCells`, a buoyancy-driven natural-convection case): under Reviewer-off the flat single index pulled a structurally adjacent tutorial whose patches were typed as plain `wall`, so every paraphrase aborted with `patch type 'wall' not constraint type`; the multi-index instead surfaced the buoyancy-flagged `hotRoom` convection tutorial directly, and the Input Writer wrote matching constraint-type patches on the first call ($0/10 \rightarrow 10/10$). Here one structural-retrieval decision replaced an entire Reviewer loop's worth of repair. *Retrieval mismatch recovered by the Reviewer* (`damBreakWithObstacle`, `interFoam`): the multi-index retrieved the closest volume-of-fluid tutorial, but its file inventory lacked `momentumTransport` and `phaseProperties`, so the first pass tripped a `cannot find file` error; the Reviewer parsed the missing-dictionary error, mapped it through the dependency graph, and regenerated each missing file, so all ten paraphrases reached the `End` marker (Reviewer-off $0/10 \rightarrow$ Reviewer-on $10/10$). Together these cases show retrieval quality and the repair loop as complementary mechanisms.

3.3.2. Reviewer Convergence and Iteration Count

Because the Reviewer dominates performance, we characterize its dynamics on FoamBench-Basic (n=110, iteration cap 10, loop index 0-indexed, consistent with the reviewer-loop counts plotted in Figure 3, where loop 0 is a case that needs no Reviewer correction). *Iteration count.* The distribution is heavily front-loaded: 74 cases finish at loop 0, 17 within loops 1–2, 16 within 3–5, 2 within 6–8, and 1 hits the cap; median 0, mean 0.96. Thus 67.3% of cases finish at the first loop and the cap-hit rate is only 0.9%: in iteration count the Reviewer is a long-tail safety net, even though it dominates wall-clock because each loop brackets a full OpenFOAM run (Section 3.7). *Convergence versus oscillation.* Measuring the rewrite-plan size $|\Delta\mathcal{S}_t|$ at each loop and classifying consecutive pairs as oscillation ($|\Delta\mathcal{S}_t| > |\Delta\mathcal{S}_{t-1}|$), stagnation ($|\Delta\mathcal{S}_t| = |\Delta\mathcal{S}_{t-1}|$), or decrease ($|\Delta\mathcal{S}_t| < |\Delta\mathcal{S}_{t-1}|$), we find that, among cases with at least two Reviewer iterations, 47.8% on Basic and 66.7% on Advanced contain at least one oscillation step yet still converge under the cap. The empirical pattern is therefore *closing-by-displacement* rather than monotone error-set decrease: each fix can expose a previously masked downstream error from a later OpenFOAM stage, consistent with the trajectory-conditioned reviewer framing of Section 2.1.5.

3.3.3. Failure Taxonomy

To locate the residual gap we label every Reviewer-off failure on FoamBench-Basic (n=110 per RAG arm) with a ten-category symptom taxonomy, recording each category's *origin* (reasoning, tool-chain, or numerical) and its *Reviewer recoverability*, the fraction of those failures recovered by the Reviewer-on rerun of the same case (Table 3). Three readings stand out. First, multi-index retrieval rescues reasoning errors upstream of the Reviewer: the no-Reviewer failure pool shifts from 29 reasoning / 24 tool-chain / 8 numerical under the single index to 10/30/7 under the multi-index, and category 3 (boundary-condition name mismatch) collapses from 10 cases to 0. Second, the Reviewer is highly effective on dictionary-grade errors: it recovers 100% of syntax and boundary-condition errors and 95.5% of missing-file errors, the largest single class. Third, the hardest residual class is solver/model mismatch (category 5, dominated by `mpirun` option errors), where recovery is only 0–12.5% because this runner-script-level evidence is not diagnosable from the solver log seen by the extractor Φ (Section 2.1.4). Aggregate Reviewer recovery on initial failures is 80.9% under the multi-index; the remaining $\approx$ 19% defines the current diagnostic frontier. This taxonomy is also the lens for the 38 exec-pass / fidelity-fail cases identified in the overall comparison: they pass M_{exec} but, dominated by categories 5 and 6, fall short on field-level fidelity.

3.4. External Mesh

Processing externally developed mesh files is a *supported capability* of Foam-Agent: it is a deliberate human–AI design boundary that lets a user supply an expert mesh when the in-the-loop meshing tools fall short for a given geometry. The dominant added bottleneck of this compute form is mesh conversion and boundary-patch consistency; Foam-Agent handles it by converting the supplied mesh to the OpenFOAM `polyMesh` format, verifying the patch inventory, and escalating to patch renaming or plan revision when the patch names do not match the prompt. Meshes are provided as `.msh` files, with the boundary names and flow scenario described in the prompt. We demonstrate this capability on two cases: 1) Multi-element airfoil (Ltd.): This case describes the flow around multiple airfoils within the domain, with the flow of one affecting the flow of another. This 2D simulation uses an inlet velocity of 9 m/s and a fluid kinematic viscosity of 1.5×10^{-5} m^2/s. The simulation is set to use the `simpleFoam` solver and Spalart-Allmaras turbulence model (Spalart and Allmaras, 1992), with a timestep of 1.0 s and a final time of 1000 s. 2) Tandem wing configuration (Ltd.): This case describes the flow around a tandem configuration, with one wing located in the wake of the other. This 3D simulation uses an inlet velocity of 9 m/s and a fluid kinematic viscosity of 1.5×10^{-5} m^2/s. The simulation is set to use the `simpleFoam` solver and the Spalart-Allmaras turbulence model, with a timestep of 1.0 s and a final time of 1000 s. The prompts along with the path to the mesh are provided to Foam-Agent for simulating the flow

Table 3
Ten-category taxonomy of Reviewer-off failures on FoamBench-Basic (n=110 per RAG arm). "cnt" is the number of Reviewer-off failures in that category under the single-index / multi-index arm; "rec%" is the fraction of those failures recovered by the Reviewer-on rerun. "—" marks an empty category.

#	Category	Origin	cnt (single)	cnt (multi)	rec% (single)	rec% (multi)
1	Syntax error	reasoning	7	7	100.0	100.0
2	Missing file	tool-chain	16	22	93.8	95.5
3	Dependency inconsistency (BC name)	reasoning	10	0	100.0	—
4	BC error (wrong type for regime)	reasoning	11	3	100.0	100.0
5	Solver/model mismatch (incl. `mpirun`)	tool-chain	8	8	0.0	12.5
6	Convergence failure (FPE / thermo bounds)	numerical	8	7	50.0	85.7
7	Mesh quality (`checkMesh`)	tool-chain	0	0	—	—
8	Geometry/meshing mismatch	reasoning	1	0	0.0	—
9	Unsupported workflow (LES / overset)	tool-chain	0	0	—	—
10	Unrecoverable reasoning (catch-all)	reasoning	0	0	—	—
Total			61	47	77.0	80.9

field; the complete user prompts for this and the other case studies are reproduced in Section B. The corresponding mesh and the contour of the velocity magnitude at the final timestep for these cases is shown in Figure 4. Further, we also compare Foam-Agent results with human expert-generated simulation results. The simulation results from Foam-Agent closely match with the result from a human OpenFOAM expert as shown in the velocity magnitude contour plots in Figure 4. Inspecting the case files generated by the agent against those prepared by humans, we found them to closely match in key aspects such as physical parameters, turbulence model selection, boundary condition assignment, and solver choice. Minor differences were observed in the numerical schemes, but these did not significantly affect the outcomes, as confirmed by the velocity contours.

3.5. Tool-Based Mesh Generation

We demonstrate the mesh-generation capabilities of Foam-Agent using the Gmsh (Geuzaine and Remacle, 2009) Python library on two cases: 1) Flow over a cylinder. 2) Flow over two square obstacles. The simulation is run for 10 s using the `pisoFoam` (Jasak et al., 2007) solver. The mesh generated by the agent and the contour of velocity magnitude at the final timestep for the two cases is shown in Figure 5. To underscore the necessity of a specialized meshing agent that leverages external tools such as Gmsh, we compare meshes generated with OpenFOAM's native meshing utilities against those produced via the Gmsh Python API. The native meshing modules were unable to capture the intended geometry of the flow scenario, whereas the Gmsh-based approach successfully generated the overall domain and accurately represented the obstacles within it. The meshing agent creates the mesh in the form of `.msh` file, which is then converted to OpenFOAM compatible format and then further used for simulation.

The dominant added bottleneck of the Gmsh pathway is geometry-to-`.geo` generation together with Gmsh syntax or geometry repair and the `.msh`→`polyMesh` conversion; Foam-Agent handles it by having the Meshing Agent generate and repair the `.geo` script, with Reviewer iterations re-entering the meshing loop whenever a conversion or patch check fails. A per-mesh-type reliability breakdown on the FoamBench-Advanced tier is reported separately in Section E.

3.6. HPC Execution

We demonstrate the capabilities of the HPC Runner Agent by instructing the framework to perform a 3D lid-driven cavity flow with one million cells, following the setup used in (OpenFOAM High Performance Computing Technical Committee, 2025). The agent generates the necessary OpenFOAM case files along with a Slurm submission script for the HPC platform *Perlmutter* (National Energy Research Scientific Computing Center, 2021). The full Slurm script produced by the agent is reproduced in Section C. To ensure strict adherence to platform-specific scheduling policies, which often vary significantly across clusters, we adopt a documentation-guided approach. Instead of relying solely on the LLM's parametric knowledge, relevant documentation regarding partition limits, module naming conventions, and header syntax is injected into the agent's context (via RAG or direct prompting). This context-aware generation reduces the risk of hallucinated directives and improves the reliability of the generated executable scripts. The dominant added bottleneck of HPC execution is not introduced by Foam-Agent but inherited from cluster computing at large: scheduler

Table 4
End-to-end runtime of Foam-Agent versus a senior OpenFOAM user on three standard cases.

Case	Human (min)	Foam-Agent (min)	Speed-up
Lid-driven cavity (`icoFoam`)	16	1.1	14.5×
Cylinder flow (incompressible)	34	3.9	8.7×
`forwardStep` (`rhoCentralFoam`)	15	7.3	2.1×

Table 5
Per-agent wall-clock attribution on FoamBench, from LLM-call timestamps. The Reviewer share includes the OpenFOAM solver execution it brackets.

	Basic		Advanced	
Agent	%	mean (s)	%	mean (s)
Reviewer	74.7	276.4	83.7	597.7
Input Writer	11.9	44.1	7.4	53.1
Architect	7.1	26.2	5.7	40.8
Runner	6.3	23.3	3.1	22.3
Total	—	370.0	—	713.9

and queue latency, job stage-in and stage-out time, and allocation-specific configuration; Foam-Agent confines the agent's responsibility to generating a valid Slurm script, while the user-supplied partition, account, node-count, and wall-time fields define the human–AI boundary. The resulting highly refined mesh, the velocity contours at the final timestep (mid-z slice), and corresponding streamlines are presented in Figure 6.

3.7. Computational Cost

We characterize the computational cost of Foam-Agent along four dimensions: end-to-end runtime against a human-expert baseline, per-agent wall-clock attribution, token and dollar cost, and scalability with problem size.

Runtime versus manual setup. Three standard cases were independently set up by a senior OpenFOAM user (about five years' experience) under a controlled protocol: start from the closest OpenFOAM v10 tutorial, consult only the OpenFOAM user guide (no LLM tools), and meet the same complete-success criterion as Foam-Agent. The human time covers code writing, review, and iterative editing but excludes the solver wall-clock incurred between iterations; the Foam-Agent time is the measured end-to-end wall-clock and *includes* solver execution between Reviewer iterations, so the speed-up of Table 4 is conservative on the Foam-Agent side. Foam-Agent is 2.1–14.5× faster than the human expert across the three cases.

Per-agent wall-clock. Table 5 attributes end-to-end wall-clock to each agent, computed from LLM-call timestamps. The Reviewer step dominates (74.7% of attributed time on FoamBench-Basic and 83.7% on Advanced) because each Reviewer pass brackets a full OpenFOAM solver run between writing the patched files and the next error-analysis call; the Architect, Input Writer, and Runner together account for the remainder. Per-case end-to-end wall-clock is 370 s mean / 148 s median on Basic and 714 s mean / 302 s median on Advanced. The Mesh and Visualization agents do not appear in this trace: meshing shells out to `blockMesh`, `snappyHexMesh`, or Gmsh without an LLM call, and visualization runs locally.

Token and dollar cost. Table 6 reports token usage and dollar cost at a rate of $3 per million input tokens and $15 per million output tokens. The Basic-tier median cost ($0.29 per case) sits well below the mean ($1.07), a gap driven by a long tail of multi-iteration cases (maximum $8.97 per case), consistent with the long-tailed Reviewer-iteration distribution of Section 3.3.2.

Scalability with problem size. Because the six agents operate on configuration files (`system/`, `constant/`, `0/`, `Allrun`) rather than on mesh data, their LLM-call time is independent of mesh resolution by construction. Solver

Table 6
Token usage and dollar cost on FoamBench, at $3/M input and $15/M output tokens.

Tier	Cases	Calls	Input	Output	Mean $	Median $
Basic	110	1 928	36.13 M	0.65 M	1.07	0.29
Advanced	16	557	5.20 M	0.40 M	1.35	1.27

wall-clock, in contrast, follows the standard CFD cost model (linear in cell count for explicit schemes, super-linear for implicit). Agent overhead is therefore a *shrinking* fraction of end-to-end cost as the problem grows. In this regime each saved Reviewer iteration avoids a full expensive run, so dependency-aware generation (which empirically lowers the iteration count) becomes disproportionately valuable on large cases. A full stratified HPC scaling grid is left as a dedicated benchmarking study.

3.8. Multi-Physics Case Study: A Multi-Region Burner

The FoamBench-Advanced tier already exercises advanced turbulence (LES, Spalart–Allmaras, k–ε, k–ω SST), complex geometries, and partially multi-physics cases. To complement that breadth with an in-depth multi-physics walkthrough, we present a *multi-region burner* case that Foam-Agent configured and ran end-to-end from a natural-language description, with no manual intervention to the generated case files.

Case description. The case is an axisymmetric, two-region conjugate-heat-transfer combustion chamber solved with `chtMultiRegionFoam` (OpenFOAM 10). A gas region and a surrounding solid wall region exchange heat across their shared interface, representative of the thermal loading of industrial furnace walls. Methane enters through a central fuel inlet and air through an annular co-flow inlet, and combustion products exit downstream. Chemical kinetics use the single-step Westbrook–Dryer mechanism integrated with an Euler-implicit stiff ODE solver, and turbulence–chemistry interaction is handled by the Partially Stirred Reactor (PaSR) model. The turbulence closure is k–ω SST, thermodynamic properties are computed from JANAF polynomials with a compressible perfect-gas equation of state, and thermal radiation is modeled with the P1 approximation. The case therefore couples reactive multi-species flow, turbulence–chemistry interaction, conjugate heat transfer, thermal radiation, and compressible thermodynamics, a combination that demands expert-level configuration in a standard OpenFOAM workflow.

Result. Figure 7 compares the steady-state temperature field of an expert reference simulation against the case generated autonomously by Foam-Agent. Both resolve the core flame structure: a high-temperature reaction zone ($T \approx 2100\,\mathrm{K}$) anchored near the fuel inlet that expands radially into the chamber, with the surrounding gas near the inlet temperature ($T \approx 300\,\mathrm{K}$). The field-level NMSE between the two solutions at the final time step is 0.0398, placing the case in the top fidelity bucket (M_{NMSE}=1) and confirming the qualitative agreement quantitatively. Foam-Agent thus navigated the full multi-physics setup (reactive flow, conjugate heat transfer, radiation, and compressible thermodynamics) without expert oversight.

3.9. Interoperability demonstration

To show that Foam-Agent can be driven as a component rather than only as a standalone pipeline, we expose its agent functions over MCP and let an external orchestrator sequence them. The MCP server design is shown in Figure 9 (Section D). In this study, we utilize the cursor environment as the orchestrator to perform a complete end-to-end simulation of the flow over a NACA 0012 airfoil (Figure 8). Specifically, the orchestrator first generates the mesh using Gmsh by invoking `generate_mesh()`. Next, it creates the input files via `generate_file_content()` (the Input Write agent). Once the files are ready, it targets an HPC node by calling `generate_hpc_script()` to produce `Allrun` and Slurm scripts, and then submits the job with `run_simulation()`. Finally, it renders the velocity magnitude by invoking `generate_visualization()`, which triggers the Visualization node.

4. Conclusion

This work introduced Foam-Agent as a domain-structured harness for automating OpenFOAM-based CFD workflows from natural language. The central goal is not merely to generate case files, but to make the latent structure of CFD case construction explicit enough that it can be retrieved, executed, inspected, and repaired. This design is not tied to a particular retrieval backend, language model, or orchestration protocol. Rather, it rests on three methodological components: multi-index retrieval over solver, physics, boundary-condition, and file-level knowledge; dependency-aware file generation through a topological traversal of the OpenFOAM case dependency graph; and a trajectory-conditioned reviewer loop that repairs failed runs through execution-grounded self-correction. Around these components, Foam-Agent organizes specialist agents for mesh generation, case setup, execution, debugging, and post-processing, and exposes the resulting workflow through a Model Context Protocol deployment surface for use by external orchestrators. The results on CFDLLMBench show that this structure significantly improves automated CFD case construction. Foam-Agent achieves execution-success rates of 88.2% on the 110 Basic-tier cases and 62.5% on the out-of-distribution Advanced tier, 1.6× and 5× those of the prior MetaOpenFOAM framework, both with Claude 3.5 Sonnet. At the same time, we also report bucketed field-level fidelity, because crash-free execution is only a necessary condition for useful CFD automation. This separation is important for evaluating scientific agents: execution success measures whether the workflow can produce a runnable simulation, while field-level fidelity tests whether the resulting flow is physically and numerically credible for the intended problem.

The main lesson from these experiments is that many failures in LLM-driven CFD automation are structural rather than purely linguistic. Incorrect simulations often arise from inconsistent dependencies across case files, incompatible solver and boundary-condition choices, insufficiently constrained mesh generation, or local repair actions that fix one error while damaging the global case structure. Foam-Agent addresses these issues by combining specialist agents with an explicitly serial dependency structure aligned with OpenFOAM practice. This design provides modularity without allowing unconstrained inter-agent negotiation to dominate a workflow that is naturally ordered by physical and numerical dependencies. The present system remains limited in several respects. It is most reliable for simple-to-moderate geometries generated through `blockMesh`, `snappyHexMesh`, or Gmsh, although it can ingest external expert meshes when needed. Its repair capability is centered on mesh regeneration, mesh-parameter adjustment, execution diagnosis, and case-level correction, rather than closed-loop modification of CAD or the pre-meshing geometry definitions. Finally, the inter-file dependency structure is currently implemented as a fixed topological order derived from the OpenFOAM directory convention, whereas future versions could infer case-specific dependencies and compile them into reusable workflows, directly addressing known multi-file consistency failures in language-model agents for scientific computing (Duston, Xin, Sun, Zan, Li, Xin, Shen, Chen, Sun, Zhang, Liu, Zhou, Liu, Pu, Wang, Ge, Tong, Ye, Zhao, Han, Cao, Zhao, Ren, Long, Liu, Huang, Du, Rong and Peng, 2025).

Future efforts should target expanding both the physical fidelity of the computational harness and the cognitive autonomy of the multi-agent architecture. On the CFD side, code-level modification can enable LLM-driven development of low-level components of CFD solvers. For example, turbulence models for LES (Yang, Bin, Shi and Yang, 2025). On the application side, real-world OpenFOAM workflow that requires multi-physics and advanced geometric understanding, such as fluid–structure interaction, overset and sliding meshes, solve-time adaptive mesh refinement, is not fully represented in the current harness. Broader validation on non-tutorial benchmarks, especially industrial geometries and case sets stratified by mesh-generation pathway, is needed to test whether the same design principles still remain effective beyond tutorial-derived examples. It would be interesting to see the role of RAG and behavior of LLM agents when confronting completely unseen industrial grade problems. On the LLM side, while user prompt is sufficient to determine the simulation in this work, it should be clarified in practice, as human can be naturally ambiguous about their intent when interacting with LLM (Somasekharan, Hassan, Lin, Panapitiya, Emami, Acharya, Horawalavithana and Pan, 2026a). On the agentic side, decentralized organizations may be complementary for less structured open-ended discovery settings (Yue, Ko, Chen, Di and Pan, 2026), while a vision-language model could inspect generated visualizations, compare them with expected physical patterns, and provide result-level feedback (Somasekharan, Pathak, Dhanakoti, Zhang, Yue, Zhu and Pan, 2026b). Furthermore, human-agent collaborative paradigms (Shao, Wang, Qian, Pan, Liu and Wang, 2026) could enable the system to gracefully solicit human intervention when encountering unrecoverable states, thereby ensuring reliability while preserving a substantial speedup over purely manual baselines. Taken together, these directions position Foam-Agent as domain-level CFD harness engineering: a composable and empirically testable workflow harness that distills expert OpenFOAM practice into a reusable component for automated numerical experiments of fluid flows.

CRediT authorship contribution statement

Ling Yue: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review and editing. **Nithin Somasekharan:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Writing – review and editing. **Tingwen Zhang:** Data curation, Methodology, Software, Validation, Writing – review and editing. **Yadi Cao:** Conceptualization, Project administration, Supervision, Writing – review and editing. **Zhangze Chen:** Validation, Visualization, Writing – review and editing. **Shimin Di:** Conceptualization, Supervision, Writing – review and editing. **Shaowu Pan:** Conceptualization, Project administration, Funding acquisition, Resources, Supervision, Writing – review and editing.

Declaration of competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work the authors used ChatGPT (OpenAI) to assist with LaTeX formatting and language editing. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

Data statement

The datasets generated and analyzed during the current study are available at `https://github.com/csml-rpi/Foam-Agent/tree/main/database`. The evaluation datasets and associated scripts are based on the CFDLLMBench framework, which is publicly available at `https://github.com/NLR-Theseus/cfdllmbench`. Source data are provided with this paper.

Code availability

The complete Foam-Agent code base, together with all system and user prompts, run configurations, evaluation scripts, and execution logs, is publicly available at `https://github.com/csml-rpi/Foam-Agent` under the MIT license. This supports full reproduction of the analyses and results reported in this study.

Acknowledgements

This work received funding support from the U.S. Department of Energy with ID DE-SC0025425. Shaowu Pan is supported by the Google Research Scholar Program. Computing resources are supported by the Lambda research grant program, NSF-ACCESS-PHY240112 and by the National Energy Research Scientific Computing Center under award NERSC DDR-ERCAP0030714.

References

, 2005. The CFD General Notation System – Standard Interface Data Structures. Recommended Practice AIAA R-101A-2005. American Institute of Aeronautics and Astronautics. doi:10.2514/4.479304.001.

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al., 2023. GPT-4 technical report. arXiv preprint arXiv:2303.08774 .

Aliyar, S., Bredmose, H., Roenby, J., Tomaselli, P.D., Sarlak, H., 2025. Directional focused wave group response of a floating wind turbine: Harmonic separation in experiments and CFD. Renewable Energy , 123516.

Anthropic, 2024. Claude 3.5 Sonnet. `https://www.anthropic.com/news/claude-3-5-sonnet`. Accessed: 2025-10-07.

Anthropic, 2024. Model Context Protocol (MCP) Specification. Technical Specification. Anthropic. URL: `https://modelcontextprotocol.io`. accessed: 2026-01-03.

Ayachit, U., 2015. The ParaView Guide: A Parallel Visualization Application. Kitware, Inc., Clifton Park, NY.

Bravo-Mosquera, P.D., Cerón-Muñoz, H.D., Díaz-Vázquez, G., Catalano, F.M., 2018. Conceptual design and CFD analysis of a new prototype of agricultural aircraft. Aerospace Science and Technology 80, 156–176.

Buehler, M., 2024. MechGPT: A language-based strategy for mechanics and materials modeling that connects knowledge across scales, disciplines, and modalities. Appl. Mech. Rev. 76, 021001.

Buehler, M.J., 2023. MeLM, a generative pretrained language modeling framework that solves forward and inverse mechanics problems. Journal of the Mechanics and Physics of Solids 181, 105454. doi:10.1016/j.jmps.2023.105454.

Bungartz, H.J., Lindner, F., Gatzhammer, B., Mehl, M., Scheufele, K., Shukaev, A., Uekermann, B., 2016. preCICE – A fully parallel library for multi-physics surface coupling. Computers & Fluids 141, 250–258. doi:10.1016/j.compfluid.2016.04.003.

Chen, Y., Zhu, X., Zhou, H., Ren, Z., 2025. MetaOpenFOAM 2.0: Large language model driven chain of thought for automating CFD simulation and post-processing. arXiv preprint arXiv:2502.00498 .

Colvin, S., Jolibois, E., Ramezani, H., Garcia Badaracco, A., Dorsey, T., Montague, D., Matveenko, S., Trylesinski, M., Runkle, S., Hewitt, D., Hall, A., Plot, V., 2025. Pydantic validation. URL: https://github.com/pydantic/pydantic. version v2.12.0a1+dev. License: MIT.

Deotale, R., Srinivasan, A., Golestanian, M., Tian, Y., Zhang, T., Vlachos, P., Gomez, H., 2026. ALL-FEM: Agentic large language models fine-tuned for finite element methods. Computer Methods in Applied Mechanics and Engineering , 118985doi:10.1016/j.cma.2026.118985.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.E., Lomeli, M., Hosseini, L., Jégou, H., 2025. The Faiss library. IEEE Transactions on Big Data .

Duston, T., Xin, S., Sun, Y., Zan, D., Li, A., Xin, S., Shen, K., Chen, Y., Sun, Q., Zhang, G., Liu, J., Zhou, H., Liu, J., Pu, Z., Wang, Y., Ge, B.X., Tong, X., Ye, F., Zhao, Z.C., Han, W., Cao, Z., Zhao, Y., Ren, W., Long, Q., Liu, Y.X., Huang, A., Du, Y., Rong, Y., Peng, J., 2025. AInsteinBench: Benchmarking coding agents on scientific repositories. URL: https://arxiv.org/abs/2512.21373, arXiv:2512.21373.

Geuzaine, C., Remacle, J.F., 2009. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. International journal for numerical methods in engineering 79, 1309–1331.

Ghafarollahi, A., Buehler, M.J., 2024. ProtAgents: protein discovery via large language model multi-agent collaborations combining physics and machine learning. Digital Discovery 3, 1389–1409. doi:10.1039/D4DD00013G.

Ghafarollahi, A., Buehler, M.J., 2025. Rapid and automated alloy design with graph neural network-powered large language model-driven multi-agent AI. MRS Bulletin 50, 1309–1324. URL: https://doi.org/10.1557/s43577-025-00953-4, doi:10.1557/s43577-025-00953-4.

Guo, J., Park, C., Qian, D., Hughes, T.J.R., Liu, W.K., 2026. Large language model-empowered next-generation computer-aided engineering. Computer Methods in Applied Mechanics and Engineering 450, 118591. doi:10.1016/j.cma.2025.118591.

Hartwanger, D., Horvat, A., 2008. 3D modelling of a wind turbine using CFD, in: NAFEMS Conference, United Kingdom.

Höpken, J., Mooney, K.G., 2012. The OpenFOAM® Technology Primer. ShareAlike.

Hou, S., Johnson, R., Makhija, R., Chen, L., Ye, Y., 2025. AutoFEA: Enhancing AI copilot by integrating finite element analysis using large language models with graph neural networks, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 24078–24085.

Jasak, H., Jemcov, A., Tukovic, Z., et al., 2007. OpenFOAM: A C++ library for complex physics simulations, in: International workshop on coupled methods in numerical dynamics, IUC Dubrovnik Croatia. pp. 1–20.

Kong, F., Kheyfets, V.O., Finol, E.A., Cai, X.C., 2018. Simulation of unsteady blood flows in a patient-specific compliant pulmonary artery with a highly parallel monolithically coupled fluid-structure interaction algorithm. International Journal for Numerical Methods in Biomedical Engineering 35. URL: https://api.semanticscholar.org/CorpusID:52953903.

Kumar, P.M., Moharana, Naik, P.B.K., Prof, Singh, K., Patel, 2020. Computational fluid dynamics. Radial Flow Turbocompressors URL: https://api.semanticscholar.org/CorpusID:1594848.

LangChain Inc., . LangGraph documentation. URL: https://langchain-ai.github.io/langgraph/. accessed: August 2025.

Lanzafame, R., Mauro, S., Messina, M., 2013. Wind turbine CFD modeling using a correlation-based transitional model. Renewable Energy 52, 31–39.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., et al., 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in neural information processing systems 33, 9459–9474.

Ltd., O., . OpenFOAM tutorials. https://github.com/openfoamtutorials/. Accessed: 2025-08-24.

M. Bran, A., Cox, S., Schilter, O., Baldassari, C., White, A.D., Schwaller, P., 2024. Augmenting large language models with chemistry tools. Nature Machine Intelligence 6, 525–535.

National Energy Research Scientific Computing Center, 2021. Perlmutter supercomputer. https://www.nersc.gov/systems/perlmutter/. U.S. DOE Office of Science User Facility, Contract No. DE-AC02-05CH11231.

Ni, B., Buehler, M.J., 2024. MechAgents: Large language model multi-agent collaborations can solve mechanics problems, generate new data, and integrate knowledge. Extreme Mechanics Letters 67, 102131. doi:10.1016/j.eml.2024.102131.

OpenFOAM High Performance Computing Technical Committee, 2025. OpenFOAM HPC benchmark suite. https://develop.openfoam.com/committees/hpc/-/tree/develop. Accessed: August 5, 2025.

Ouyang, C., Yue, L., Di, S., Zheng, L., Yue, L., Pan, S., Yin, J., Zhang, M.L., 2026. Code2MCP: Transforming code repositories into MCP services. URL: https://arxiv.org/abs/2509.05941, arXiv:2509.05941.

Pandey, S., Xu, R., Wang, W., Chu, X., 2025. OpenFOAMGPT: A retrieval-augmented large language model (LLM) agent for OpenFOAM-based computational fluid dynamics. Physics of Fluids 37.

Shao, E., Wang, Y., Qian, Y., Pan, Z., Liu, H., Wang, D., 2026. SciSciGPT: advancing human–AI collaboration in the science of science. Nature Computational Science 6, 301–315.

Shinn, N., Cassano, F., Labash, B., Gopinath, A., Narasimhan, K., Yao, S., 2023. Reflexion: language agents with verbal reinforcement learning, in: Neural Information Processing Systems. URL: https://api.semanticscholar.org/CorpusID:258833055.

Slotnick, J.P., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., Mavriplis, D.J., 2014. CFD vision 2030 study: a path to revolutionary computational aerosciences. Technical Report. NASA. See "Poor mesh generation performance and robustness" at Section 5.3.

Somasekharan, N., Hassan, Y., Lin, S., Panapitiya, G., Emami, P., Acharya, A., Horawalavithana, S., Pan, S., 2026a. SCICONVBENCH: Benchmarking LLMs on multi-turn clarification for task formulation in computational science. arXiv preprint arXiv:2605.18630 .

Somasekharan, N., Pathak, R., Dhanakoti, M., Zhang, T., Yue, L., Zhu, A., Pan, S., 2026b. AI CFD scientist: Toward open-ended computational fluid dynamics discovery with physics-aware AI agents. arXiv preprint arXiv:2605.06607 .

Somasekharan, N., Yue, L., Cao, Y., Li, W., Emami, P., Bhargav, P.S., Acharya, A., Xie, X., Pan, S., 2026c. CFDLLMBench: A benchmark suite for evaluating large language models in computational fluid dynamics. Journal of Data-centric Machine Learning Research 3, 1–40. URL: https://openreview.net/forum?id=kTcH1MnkjY.

Spalart, P., Allmaras, S., 1992. A one-equation turbulence model for aerodynamic flows, in: 30th aerospace sciences meeting and exhibit, p. 439.
Steinman, D.A., 2002. Image-based computational fluid dynamics modeling in realistic arterial geometries. Annals of biomedical engineering 30, 483–497.
Sullivan, C., Kaszynski, A., 2019. PyVista: 3D plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK). Journal of Open Source Software 4, 1450.
Tu, J., Yeoh, G.H., Liu, C., Tao, Y., 2023. Computational fluid dynamics: a practical approach. Elsevier.
Wang, F.Y., Marom, L., Pal, S., Luu, R.K., Lu, W., Berkovich, J.A., Buehler, M.J., 2026. Autonomous agents coordinating distributed discovery through emergent artifact exchange. https://arxiv.org/abs/2603.14312. Preprint at https://arxiv.org/abs/2603.14312.
Wang, Z.J., 2014. High-order computational fluid dynamics tools for aircraft design. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 372.
Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., et al., 2025. The rise and potential of large language model based agents: A survey. Science China Information Sciences 68, 121101.
Yang, Z., Bin, Y., Shi, Y., Yang, X.I., 2025. Large language model driven development of turbulence models. Flow 5, E40.
Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K.R., Cao, Y., 2023. ReAct: Synergizing reasoning and acting in language models, in: The eleventh international conference on learning representations.
Yue, L., Ko, C.Y., Chen, P.Y., Di, S., Pan, S., 2026. Building mcp-native hierarchical ai scientist ecosystems: a perspective on scaling multi-agent scientific discovery. Frontiers in Artificial Intelligence 9, 1820375.
Zhang, J., Zhong, L., Su, B., Wan, M., Yap, J.S., Tham, J.P.L., Chua, L.P., Ghista, D.N., Tan, R.S., 2014. Perspective on CFD studies of coronary artery disease lesions and hemodynamics: A review. International Journal for Numerical Methods in Biomedical Engineering 30. URL: https://api.semanticscholar.org/CorpusID:205686247.
Zhang, T., Liu, Z., Xin, Y., Jiao, Y., 2025. MooseAgent: A LLM based multi-agent framework for automating moose simulation. arXiv preprint arXiv:2504.08621 .

A. System and User Prompts

This appendix presents all system and user prompts used in the Foam-Agent framework for various components. These prompts are organized by agent role and function within the multi-agent architecture.

A.1. Architect Agent Prompts

The Architect Agent interprets user requirements into a structured simulation plan and breaks down complex tasks into manageable subtasks.

A.1.1. Case Description Prompts

Case Description System Prompt

Please transform the following user requirement into a standard case description using a structured format. The key elements should include case name, case domain, case category, and case solver.

Case Description User Prompt

User requirement: {user_requirement}.

user_requirement here refers to the input simulation request from the user. Example, see Figure 1 in the main manuscript.

A.1.2. Task Decomposition Prompts

Task Decomposition System Prompt

You are an experienced Planner specializing in OpenFOAM projects. Your task is to break down the following user requirement into a series of smaller, manageable subtasks. For each subtask, identify the file name of the OpenFOAM input file (example: U, system etc.) and the corresponding folder name where it should be stored. Your final output must strictly follow the JSON schema below and include no additional keys or information:
{
“subtasks”: [
{

"file_name": "<string>",
"folder_name": "<string>"
}
// ... more subtasks
]
}
Make sure that your output is valid JSON and strictly adheres to the provided schema. Make sure you generate all the necessary files for the user's requirements.

Task Decomposition User Prompt

User Requirement: {user_requirement}
Reference Directory Structure (similar case): {dir_structure}
{dir_counts_str}
Make sure you generate all the necessary files for the user requirements. Please generate the output as structured JSON.

dir_structure refers to the folders and files that need to be created for simulating the requested scenario. For example, `0/U`, `constant/momentumTransport`, `system/controlDict` etc. dir_counts_str refers to the number of directories to be created (`0`, `constant`, `system` etc.).

A.2. Input Writer Agent Prompts

The Input Writer Agent generates OpenFOAM configuration files and ensures consistency across interdependent files based on the user requirements and the simulation plan obtained from the architect agent. The architect agent provides the information of the files to be written and the folder it has to be saved into to the input writer agent, which writes the content of these files. Below, the instructions for the Input Writer Agent are outlined for generating configuration files, terminal commands, and scripts.

A.2.1. File Generation Prompts

File Generation System Prompt

You are an expert in OpenFOAM simulation and numerical modeling. Your task is to generate a complete and functional file named: <file_name>{file_name}</file_name> within the <folder_name>{folder_name}</folder_name> directory. Ensure all required values are present and match with the files content already generated. Before finalizing the output, ensure:

- All necessary fields exist (e.g., if 'nu' is defined in 'constant/transportProperties', it must be used correctly in '0/U').
- Cross-check field names between different files to avoid mismatches.
- Ensure units and dimensions are correct for all physical variables.
- Ensure case solver settings are consistent with the user requirements.

Available solvers are: {state.case_stats['case_solver']}. Provide only the code – no explanations, comments, or additional text.

File Generation User Prompt

User requirement: {state.user_requirement}. Refer to the following similar case file content to ensure the generated file aligns with the user requirement: <similar_case_reference>{similar_file_text}</similar_case_reference>. Similar case reference is always correct. If you find the user requirement is very consistent with the similar case reference, you should use the

similar case reference as the template to generate the file. Just modify the necessary parts to make the file complete and functional. Please ensure that the generated file is complete, functional, and logically sound. Additionally, apply your domain expertise to verify that all numerical values are consistent with the user requirements, maintaining accuracy and coherence.
You should ensure that the new file is consistent with the previous files. Such as boundary conditions, mesh settings, etc.

A.2.2. Command Generation Prompts

Command Generation System Prompt

You are an expert in OpenFOAM. The user will provide a list of available commands. Your task is to generate only the necessary OpenFOAM commands required to create an Allrun script for the given user case, based on the provided directory structure. Return only the list of commands – no explanations, comments, or additional text.

Command Generation User Prompt

Available OpenFOAM commands for the Allrun script: {commands}
Case directory structure: {state.dir_structure}
User case information: {state.case_info}
Reference Allrun scripts from similar cases: {state.allrun_reference}
Generate only the required OpenFOAM command list—no extra text.

The commands passed in the user prompt to this agent form a pre-curated list of OpenFOAM commands, saved within Foam-Agent. The user need not provide any command within the input.

A.2.3. Allrun Script Generation Prompts

Allrun Script Generation System Prompt

You are an expert in OpenFOAM.
Generate an Allrun script based on the provided details.
Available commands with descriptions: {commands_help}
Reference Allrun scripts from similar cases: {state.allrun_reference}

Allrun Script Generation User Prompt

User requirement: {state.user_requirement}
Case directory structure: {state.dir_structure}
User case information: {state.case_info}
All run scripts for these similar cases are for reference only and may not be correct, as you might be a different case solver or have a different directory structure. You need to rely on your OpenFOAM and physics knowledge to discern this, and pay more attention to user requirements, as your ultimate goal is to fulfill the user's requirements and generate an allrun script that meets those requirements. Generate the Allrun script strictly based on the above information. Do not include explanations, comments, or additional text. Put the code in ``` tags.

A.3. Reviewer Agent Prompts

The Reviewer Agent analyzes simulation errors and proposes corrections to resolve issues.

A.3.1. Error Analysis Prompts

Error Analysis System Prompt

You are an expert in OpenFOAM simulation and numerical modeling. Your task is to review the provided error logs and diagnose the underlying issues. You will be provided with a similar case reference, which is a list of similar cases that are ordered by similarity from high to low. You can use this reference to help you understand the user requirement and the error. When an error indicates that a specific keyword is undefined (for example, 'div(phi,(p|rho)) is undefined'), your response must propose a solution that simply defines that exact keyword as shown in the error log. Do not reinterpret or modify the keyword (e.g., do not treat `div(phi,(p|rho))` as meaning "or"); instead, assume it is meant to be taken literally. Propose ideas on how to resolve the errors, but do not modify any files directly. Please do not propose solutions that require modifying any parameters declared in the user requirement, try other approaches instead. Do not ask the user any questions. The user will supply all relevant foam files along with the error logs, and within the logs, you will find both the error content and the corresponding error command indicated by the log file name.

Error Analysis User Prompt (Initial Error)

<similar_case_reference>{state.tutorial_reference}</similar_case_reference>
<foamfiles>{str(state.foamfiles)}</foamfiles>
<error_logs>{state.error_logs} </error_logs>
<user_requirement>{state.user_requirement}</user_requirement>
Please review the error logs and provide guidance on how to resolve the reported errors. Make sure your suggestions adhere to user requirements and do not contradict it.

Error Analysis User Prompt (Subsequent Errors)

<similar_case_reference>{state.tutorial_reference}</similar_case_reference>
<foamfiles>{str(state.foamfiles)}</foamfiles>
<current_error_logs>{state.error_logs} </current_error_logs>
<history> {chr(10).join(state.history_text)} </history>
<user_requirement>{state.user_requirement}</user_requirement>
I have modified the files according to your previous suggestions. If the error persists, please provide further guidance. Make sure your suggestions adhere to user requirements and do not contradict them. Also, please consider the previous attempts and try a different approach.

A.3.2. File Correction Prompts

File Correction System Prompt

You are an expert in OpenFOAM simulation and numerical modeling. Your task is to modify and rewrite the necessary OpenFOAM files to fix the reported error. Please do not propose solutions that require modifying any parameters declared in the user requirement; try other approaches instead. The user will provide the error content, error command, reviewer's suggestions, and all relevant foam files. Only return files that require rewriting, modification, or addition; do not include files that remain unchanged. Return the complete, corrected file contents in the following JSON format:
{
list of foamfile: [{file_name: 'file_name',
folder_name: 'folder_name',
content: 'content'}].
}
Ensure your response includes only the modified file content with no extra text, as it will be parsed using Pydantic.

File Correction User Prompt

<foamfiles>{str(state.foamfiles)}</foamfiles>
<error_logs>{state.error_logs}</error_logs>
<reviewer_analysis>{review_content}</reviewer_analysis>
<user_requirement>{state.user_requirement}</user_requirement>

Please update the relevant OpenFOAM files to resolve the reported errors, ensuring that all modifications strictly adhere to the specified formats. Ensure all modifications adhere to user requirement.

A.4. History Tracking Format

The system tracks modification history using a structured format for each iteration attempt:

History Tracking Format

<Attempt {attempt_number}>
<Error_Logs> {state.error_logs} </Error_Logs>
<Review_Analysis> {review_content} </Review_Analysis>
</Attempt>

A.5. Example User Requirements

Below is an example of a user requirement used to test the Foam-Agent system:

Example User Requirement

Perform a 3D Benard Cell simulation using OpenFOAM. Run a two-dimensional Rayleigh Benard convection cell simulation in a rectangular cavity that is 9 units wide and 1 unit tall, extruded a single cell in the spanwise direction with the front and back patches treated as empty to enforce 2D. Use 90 cells in x direction, 10 cells in y and 1 in z. Use the buoyantFoam solver with gravity acting downward in the negative y direction. The reference state should be air-like with density 1.0 kg/m^3, reference temperature 300 K, thermal expansion coefficient 3.3e-3 1/K, kinematic viscosity 1.5e-5 m^2/s, and thermal diffusivity 2.1e-5 m^2/s (Pr = 0.7). Temperature of bottom wall is fixed at 301 K and the top wall at 300 K, while the vertical side walls are adiabatic. Apply no-slip velocity conditions on all solid walls, initialize the flow at rest with a uniform temperature field of 300 K. Use a timestep of 1s, simulate up to 1000 s of physical time, and write results every 50 steps. Use k-epsilon turbulence model.

B. User Prompts In Case Studies

B.1. Multi Element Airfoil

Example User Requirement

Perform a 2D incompressible flow over a multi element airfoil setup. The mesh is provided as a `.msh` file. The `.msh` file contains 4 boundaries named "inlet", "outlet", "walls", "airfoil" and "frontAndBack". The "inlet" and "outlet" are of type freestream with the freestream velocity being 9 m/s. The "walls" and "airfoil" have a no-slip boundary condition (velocity equal to zero at the wall). The "frontAndBack" faces are designated as 'empty'. The simulation runs from time 0 to 1000 with a time step of 1.0 units, and results are output every 1 time step. The viscosity (nu) is set as constant with a value of 1.5×10^{-5} m^2/s. Use simpleFoam solver. Use SpalartAllmaras turbulence model. Further visualize the magnitude of velocity along the Z plane.

B.2. Tandem Wing

Example User Requirement

Perform a 3D incompressible flow over a tandem wing configuration. The mesh is provided as a `.msh` file. The msh file contains 4 boundaries named "inlet", "outlet", "walls", "airfoil" and "frontAndBack". The "inlet" and "outlet" are of type freestream with the freestream velocity being 9 m/s. The "walls" and "airfoil" have a no-slip boundary condition (velocity equal to zero at the wall). The "frontAndBack" faces are also of type wall. The simulation runs from time 0 to 1000 with a time step of 1.0 units, and results are output every 1 time step. The viscosity ('nu') is set as constant with a value of 1.5×10^{-5} m^2/s. Use simpleFoam solver. Use SpalartAllmaras turbulence model. Further visualize the magnitude of velocity along the mid Z section at the final time.

B.3. Flow Over Cylinder

Example User Requirement

Simulate incompressible flow over a circular cylinder. Use Gmsh to create the computational mesh. The computational domain extends from -2.5 to 2.5 in the x-direction, -1 to 1 in the y-direction, and 0 to 0.2 in the z-direction. The cylinder is positioned at (-1, 0) with a radius of 0.1 units. Use a structured mesh with approximately 20x10 cells in the x-y plane and 1 cell in the z-direction. The inlet boundary named "inlet" (left boundary at x = -2.5) has a uniform velocity of 1 m/s in the positive x-direction. The right boundary at x=+2.5 is the outlet named "outlet". The top and bottom walls named "topWall" and "bottomWall" respectively (y = +1 and y=-1) use slip boundary conditions. The cylinder surface named "cylinder" uses a no-slip boundary condition (velocity equal to zero at the wall). The front and back faces named "frontAndBack" are located at z = 0 and z = 0.2 respectively, and are designated as 'empty' for 2D simulation. Use base mesh size of 0.5 on cylinder and size of 1.0 elsewhere. The simulation runs from time 0 to 2 seconds with a time step of 0.001 units, and results are output every 100 time steps. The kinematic viscosity (nu) is set as constant with a value of 1×10^{-5} m^2/s. Use pisoFoam solver for incompressible flow. Visualize the magnitude of velocity ('U') along the x-y plane.

B.4. Flow Over Two Square Obstacles

Example User Requirement

Simulate incompressible flow over two square obstacles. Use Gmsh to create the computational mesh. The computational domain spans 0 to 5 in x direction and 0 to 2.5 in y direction and 0 to 0.1 in the z direction. One of the square obstacle is of size 0.25 unit x 0.25 unit x 0.1 unit centered at 1.5 x 1.25 x 0.0 and the other square obstacle is of size 0.25 unit x 0.25 unit x 0.1 unit centered at 3.5 x 1.25 x 0.0. Use one cell in z direction making the geometry effectively 2D. Use a structured mesh with approximately 50x25 cells in the x-y plane and 1 cell in the z-direction. The inlet boundary named "inlet" (left boundary at x = 0) has a uniform velocity of 1 m/s in the positive x-direction. The right boundary at x = 5 is the outlet named "outlet". The top and bottom walls named "topWall" and "bottomWall" respectively (y = 2.5 and y = 0) use slip boundary conditions. The square obstacle surfaces named "square1" and "square2" use no-slip boundary conditions (velocity equal to zero at the walls). The front and back faces named "frontAndBack" are located at z = 0 and z = 0.1 respectively, and are designated as 'empty' for 2D simulation. Use base mesh size of 0.5 on squares and size of 1.0 elsewhere. The simulation runs from time 0 to 10 seconds with a time step of 0.001 units, and results are output every 100 time steps. The kinematic viscosity (nu) is set as constant with a value of 1×10^{-5} m^2/s. Use pisoFoam solver for incompressible flow. Visualize the magnitude of velocity ('U') along the x-y plane.

B.5. 3D cavity HPC Case

Example User Requirement

Do an incompressible 3D lid driven cavity flow using icoFoam solver. The cavity is a cube of dimension [0, 0.1] × [0, 0.1] × [0,0.1]. Use simple grading with 100 ×100×100 in x, y and z direction. The top wall ('movingWall') moves in the x-direction with a uniform velocity of 1 m/s. The 'fixedWalls' have a no-slip boundary condition (velocity equal to zero at the wall). The simulation runs from time 0 to 0.015 with a time step of 0.001 units, and results are output every 10 time steps. The viscosity (nu) is set as constant with a value of 0.01 m^2/s. Perform a HPC run for this case in perlmutter cluster. My account is ACCOUNTNAME. Do a parallel run for this case by splitting it into 32 subdomains.

C. Slurm Script Generated by the Agent

Slurm Script Generated by the Agent

```
#!/bin/bash
#SBATCH -A xxxx
#SBATCH -C cpu
#SBATCH -q regular
#SBATCH -N 1
#SBATCH -n 32
#SBATCH –ntasks-per-node=32
#SBATCH -t 02:00:00
#SBATCH –mem=128GB
#SBATCH -J Sim
#SBATCH -o %j.out
#SBATCH -e %j.err

# Initialize error handling
set -e

# Change to case directory
cd ./runshpcnode || exit 1

# Create log directory if it doesn’t exist
mkdir -p logs

# Run the simulation
echo “Starting OpenFOAM simulation at $(date)”
./Allrun -parallel
if [ $? -eq 0 ]; then
echo “Simulation completed successfully at $(date)”
exit 0
else
echo “Simulation failed at $(date)”
exit 1
fi
```

Table 7
The core functions of the Foam-Agent Model Context Protocol (MCP). Each function represents a decoupled capability within the CFD workflow, featuring strongly-typed inputs and outputs to ensure reliable interaction with orchestrating agents.

Function Name	Description	Input Schema	Output Schema
`create_case`	Initializes a new CFD simulation case and its workspace.	`{user_prompt: str}`	`{case_id: str}`
`plan_simulation_structure`	(Architect Agent) Plans the required file and directory structure based on the user prompt.	`{case_id: str}`	`{plan: List[{file, folder}]}`
`generate_file_content`	(Input Writer Agent) Generates the content for a single specified configuration file.	`{case_id, file, folder}`	`{content: str}`
`generate_mesh`	(Meshing Agent) Asynchronously generates the computational mesh using a specified method.	`{case_id, mesh_config: Dict}`	`{job_id: str}`
`generate_hpc_script`	(HPC Agent) Generates a job submission script (e.g., Slurm) for a high-performance computing cluster.	`{case_id, hpc_config: Dict}`	`{script_content: str}`
`run_simulation`	(Runner Agent) Asynchronously executes the simulation either locally or by submitting to an HPC cluster.	`{case_id, environment: str}`	`{job_id: str}`
`check_job_status`	Checks the status of any asynchronous job (meshing, simulation, visualization).	`{job_id: str}`	`{status: Dict}`
`get_simulation_logs`	Retrieves detailed logs for a failed job to enable error diagnosis.	`{case_id, job_id}`	`{logs: Dict}`
`review_and_suggest_fix`	(Reviewer Agent) Analyzes error logs and proposes corrective actions.	`{case_id, logs}`	`{suggestions: Dict}`
`apply_fix`	Applies suggested modifications to the relevant case files.	`{case_id, modifications: List}`	`{status: str}`
`generate_visualization`	(Visualization Agent) Asynchronously generates a visualization of the simulation results.	`{case_id, quantity, ...}`	`{job_id: str}`

D. MCP Functions

The purpose of each function in the MCP compliant architecture of Foam-Agent is shown in Table 7; the overall MCP design, in which Foam-Agent's functionality is decomposed into basic functions and exposed to an external orchestrator, is shown in Figure 9.

Protocol-level characteristics. Two protocol-boundary concerns are documented here for completeness. *Latency.* Per-call round-trip overhead is small in practice (single-digit milliseconds for local STDIO transports and tens of milliseconds for HTTP transports) and is dominated end-to-end by OpenFOAM solver wall-clock per Reviewer iteration (Section 3.7). *Protocol-layer failure modes.* Four error classes arise at the protocol boundary: schema-validation errors on malformed argument payloads, call timeouts when a long-running solver invocation exceeds the orchestrator-side timeout, solver-crash propagation when an OpenFOAM signal exit is surfaced through a tool-call

Table 8
Prompt-sensitivity robustness on the GPT-5.3-codex backbone: three LLM paraphrases per dataset over the 27 FoamBench datasets.

Robustness check	Result
M_{NMSE} bucket agreement across all 3 paraphrases	18/27
M_{exec} agreement across all 3 paraphrases	22/27
Advanced-tier bucketed M_{NMSE} mean variation	0
Basic-tier bucketed M_{NMSE} mean variation	≤ 0.09

Table 9
Per-mesh-type success on FoamBench-Advanced (n=16), Claude Sonnet 4.6 backbone. "buckets" are the case counts in fidelity buckets 1/0.5/0.

Mesh kind	Cases	M_{exec}	M_{NMSE}	buckets (1/0.5/0)
`blockMesh`	11	0.727	0.091	1/0/10
`snappyHexMesh`	5	0.800	0.000	0/0/5

return, and orchestrator session drops during long Reviewer loops. Each is returned as a structured error payload via the MCP error contract, preserving Foam-Agent's standalone error semantics across the boundary.

E. Supplementary Experiments on Additional Backbones

The two studies reported in this appendix were run on backbones other than the Claude 3.5 Sonnet model used for the headline FoamBench evaluation—a prompt-sensitivity study on GPT-5.3-codex and a per-mesh-type reliability breakdown on Claude Sonnet 4.6—and are presented separately, as relative within-study comparisons.

E.1. Prompt Sensitivity

Because Foam-Agent is driven by natural language, we test robustness to prompt phrasing. For each of the 27 FoamBench datasets (11 Basic-tier scenarios and 16 Advanced-tier cases) we generated three LLM paraphrases of the prompt under a syntactic-drift guard that preserves all numerics, single-quoted boundary-condition names, and OpenFOAM solver names; the median embedding-space faithfulness to the original prompt is 0.959 (cosine similarity). Table 8 summarizes the stability: the bucketed M_{NMSE} agrees across all three paraphrases for 18/27 datasets and M_{exec} for 22/27, the Advanced-tier bucketed M_{NMSE} mean shows zero variation, and the Basic-tier mean varies by at most 0.09. The 5/27 datasets with M_{exec} disagreement concentrate in known-hard cases (e.g. `forwardStep` and `wedge` on Basic, `Rectangular_Obstacle` variants on Advanced), indicating case difficulty rather than phrasing sensitivity. We read this as evidence of semantic-equivalence robustness; lexical-diversity robustness across model families and manual rewrites is a complementary evaluation.

E.2. Per-Mesh-Type Reliability

To characterize mesh-type reliability we instrument a FoamBench-Advanced run (n=16) with per-invocation mesh-type metadata (Table 9); FoamBench-Basic is uniformly `blockMesh`, so the Basic numbers already report blockMesh performance. Two observations follow. The `snappyHexMesh` Advanced cases reach $M_{\text{exec}} = 0.80$ but cluster in fidelity bucket 0: generating a topologically valid hex-dominant mesh is not the bottleneck on this path—the residual is solver-side physical accuracy. The Advanced `blockMesh` cases are bimodal, with one complete-success case and the remainder split between divergent and execution-failure outcomes. In both pathways the limiting factor on the Advanced tier is solver fidelity and parameter setup, not mesh generation. The absolute values in Table 9 are reported for the within-table mesh-type comparison.

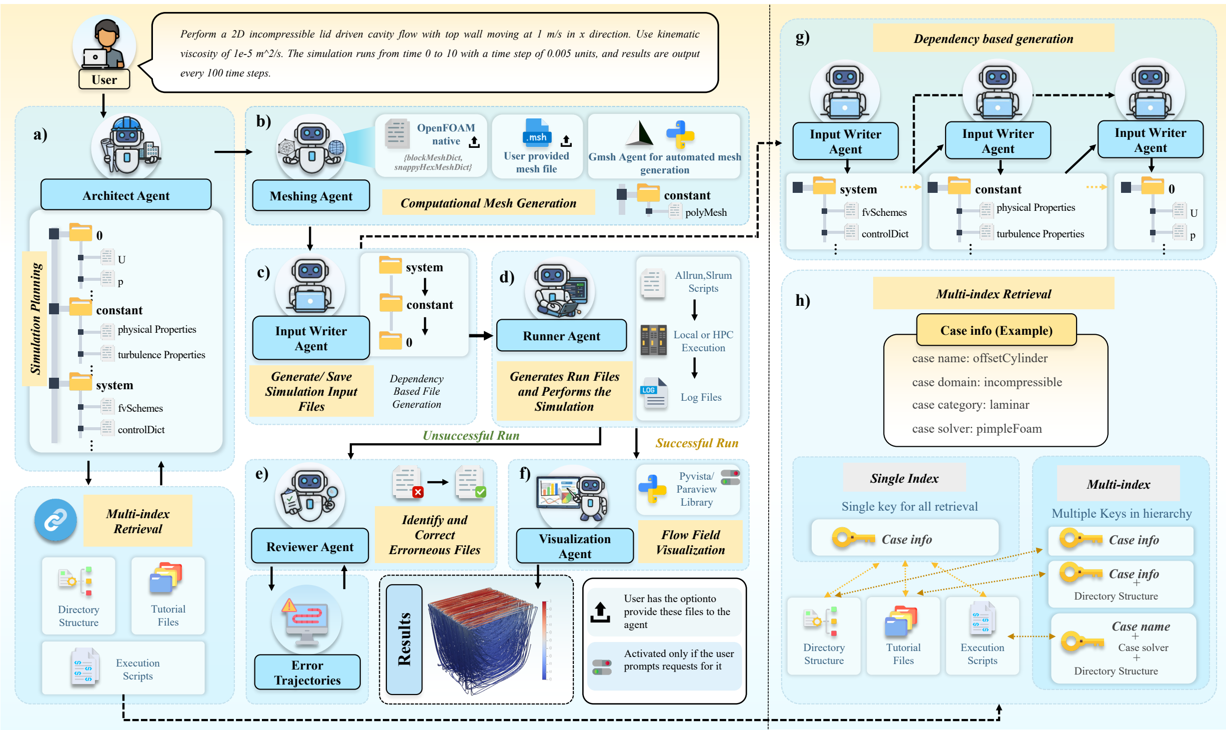


Figure 1: Foam-Agent architecture. Starting from a natural-language request about CFD tasks, the framework plans the case structure, generates or imports the mesh, writes OpenFOAM files, executes the case, debugs failed runs, and produces post-processing visualizations. Panels (a)–(f) show the six agents; panel (g) shows dependency-aware file generation; panel (h) shows multi-index retrieval.

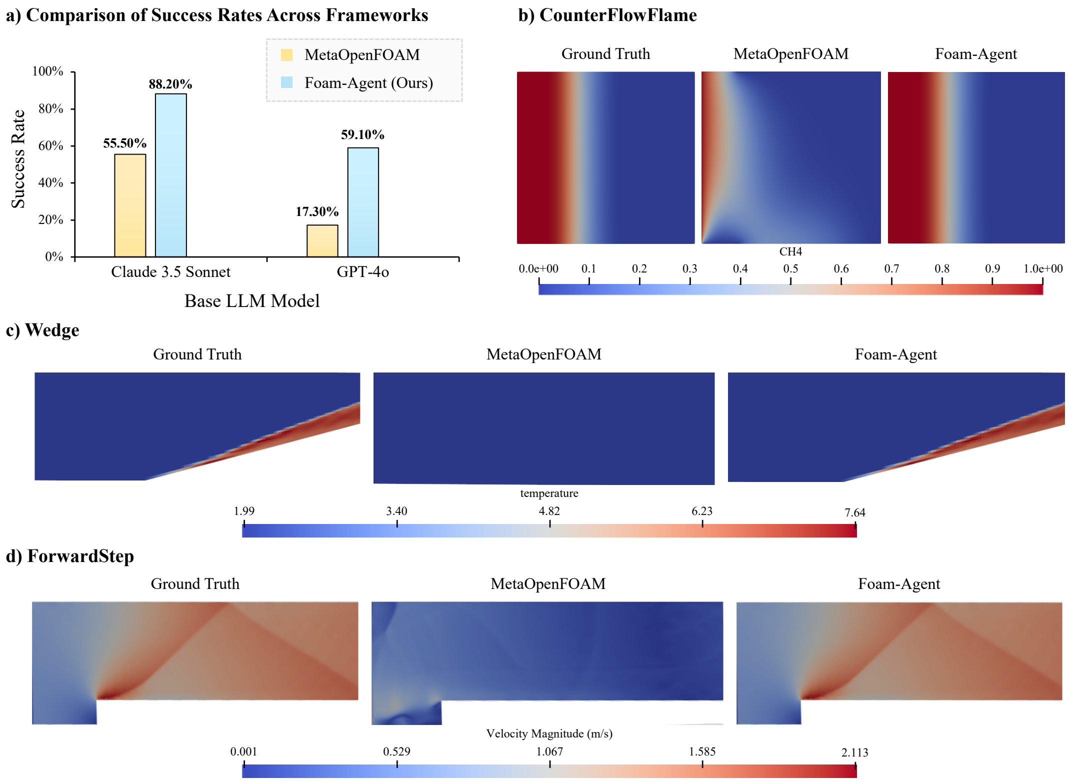


Figure 2: Performance comparison between MetaOpenFOAM and Foam-Agent. (a) Execution-success rate (M_{exec}) with Claude 3.5 Sonnet and GPT-4o on the CFDLLMBench dataset; the corresponding bucketed field-level fidelity (M_{NMSE}) is reported in Table 1. (b)–(d) Qualitative comparison of simulation contours against ground truth, complementing the quantitative M_{NMSE} metric: (b) CH_4 mass fraction at $t = 0.5$ s for the *CounterFlowFlame* case; (c) temperature at $t = 0.2$ s for the *wedge* case; (d) velocity magnitude at $t = 4.0$ s for the *forwardStep* case. Ground truth is generated by a human expert. Foam-Agent visuals are automatically generated by the agent via Python scripts, which are also used to render the baselines for fair comparison.

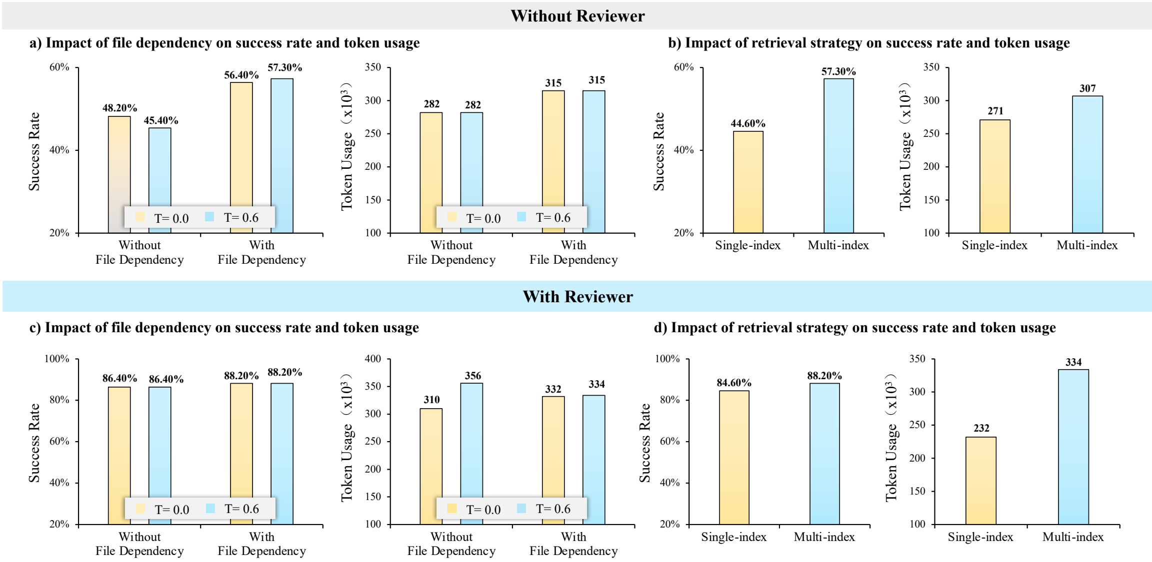


Figure 3: Ablation study of strategies of file generation, retrieval and iterative reviewer used in Foam-Agent. We also study the effect of LLM temperature parameters by setting it to a low value of 0 and high value of 0.6. (a) Impact of file dependency on execution success rate and token usage without the reviewer agent, for LLM temperature values of 0 and 0.6. (b) Impact of retrieval strategy (multi-index versus baseline single-index RAG) on execution success rate and token usage without the reviewer agent, for LLM temperature values of 0 and 0.6. (c) Impact of file dependency on execution success rate, token usage, and average reviewer loops with the reviewer agent, for LLM temperature values of 0 and 0.6. (d) Impact of retrieval strategy (multi-index versus baseline single-index RAG) on execution success rate and token usage with the reviewer agent, for LLM temperature values of 0 and 0.6. Claude 3.5 Sonnet is the prompt model here.

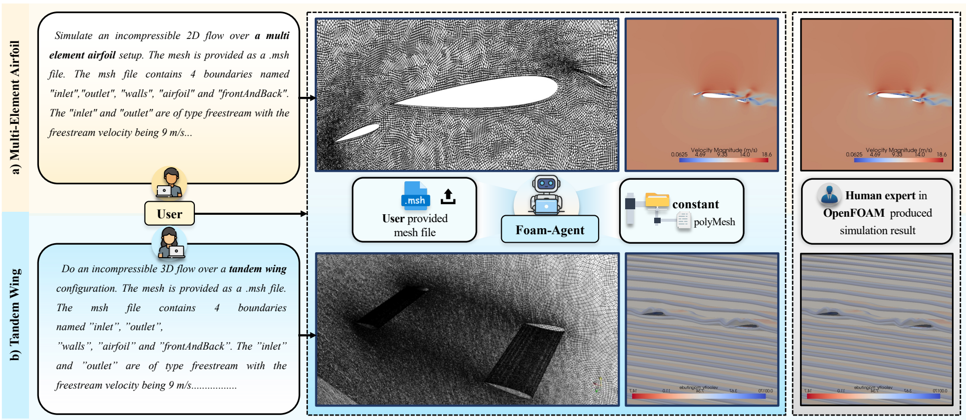


Figure 4: External mesh cases. Foam-Agent converts user-supplied `.msh` files to OpenFOAM format, generates the remaining case files, and produces final velocity-magnitude visualizations for (a) a multi-element airfoil and (b) a tandem wing. The same visualization script is used for the agent and expert solutions to ensure a consistent qualitative comparison.

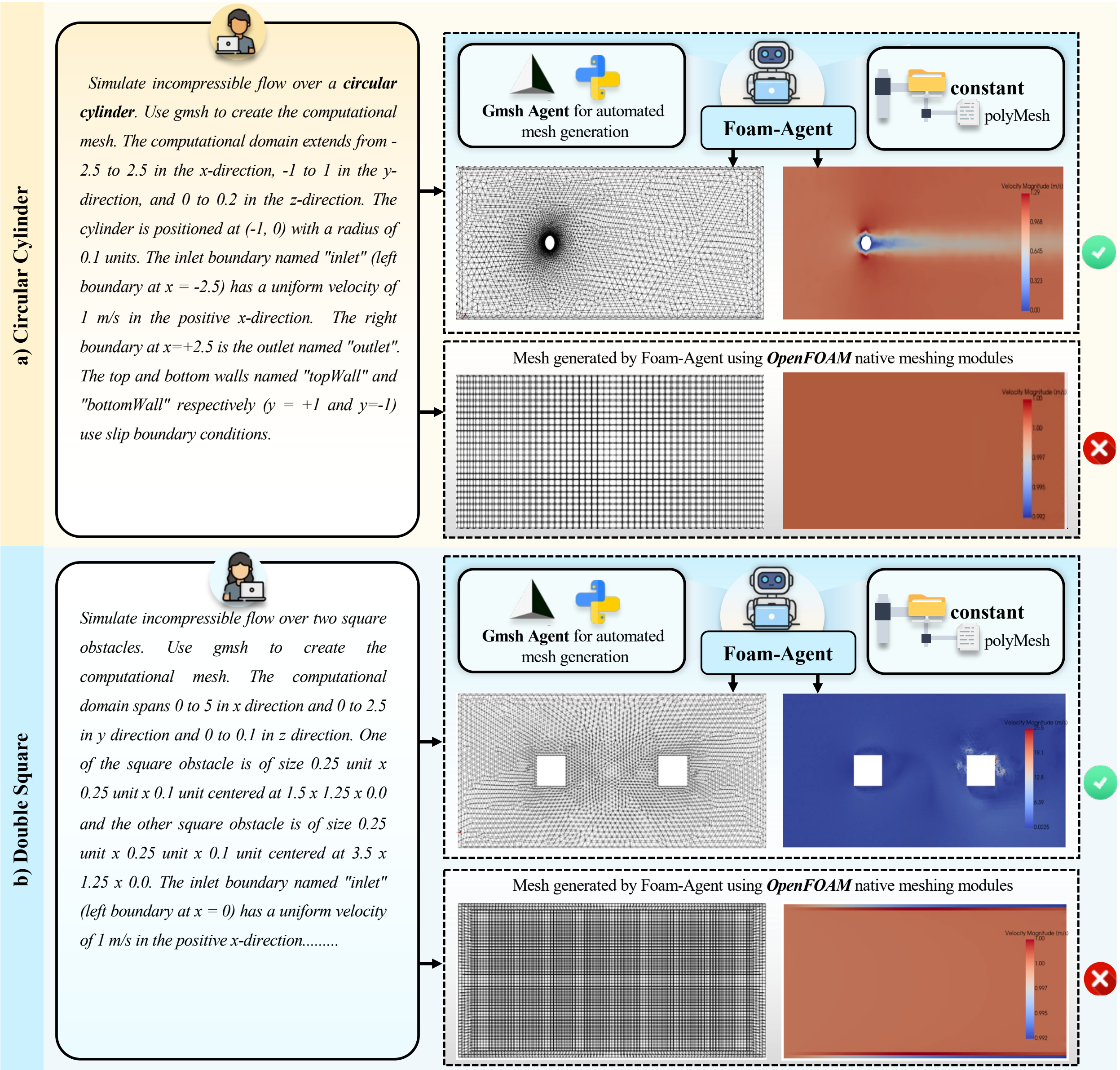


Figure 5: Tool-based mesh generation with Gmsh. For (a) flow over a cylinder and (b) flow over two square obstacles, Foam-Agent generates a Gmsh Python script, converts the resulting `.msh` file to OpenFOAM format, runs the case, and visualizes the final velocity magnitude. OpenFOAM native meshing does not represent the obstacles correctly in these examples.

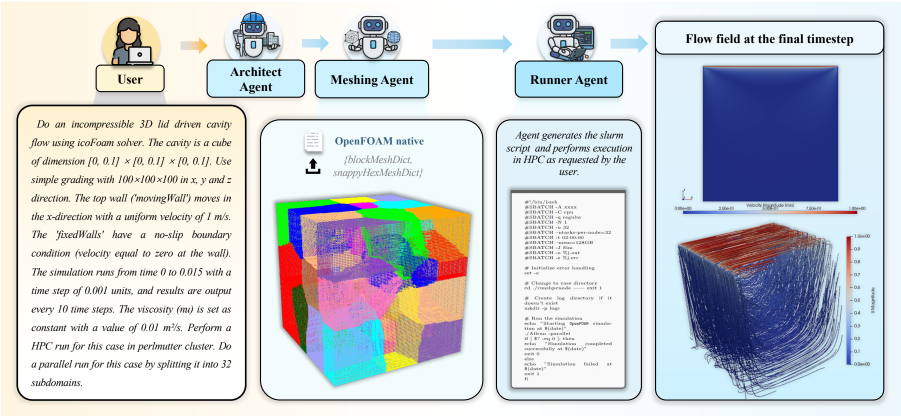
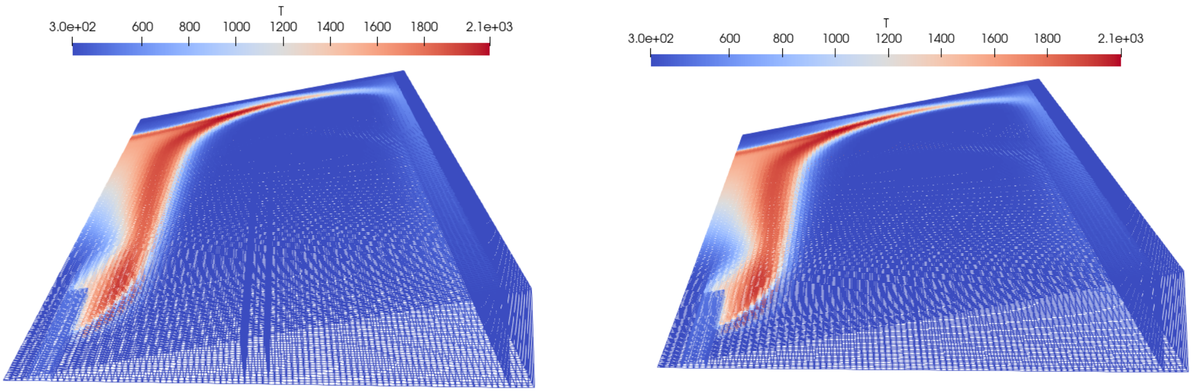


Figure 6: HPC execution of a 3D lid-driven cavity case. Foam-Agent generates the OpenFOAM setup, domain decomposition, and Slurm script for Perlmutter, runs the case on 32 subdomains, and produces the final mid-z velocity contour. The 3D streamlines are included only for qualitative illustration.



(a) Expert reference

(b) Foam-Agent

Figure 7: Steady-state temperature field T [K] for the multi-region burner case (`chtMultiRegionFoam`; k–ω SST + PaSR + Westbrook–Dryer kinetics + conjugate heat transfer + P1 radiation). (a) Expert reference configured by a senior OpenFOAM user; (b) Foam-Agent configured autonomously from a natural-language description. Both resolve a $\approx 2100\,\mathrm{K}$ flame zone anchored at the fuel inlet; the field-level NMSE between (a) and (b) at the final time step is 0.0398 (M_{NMSE}=1).

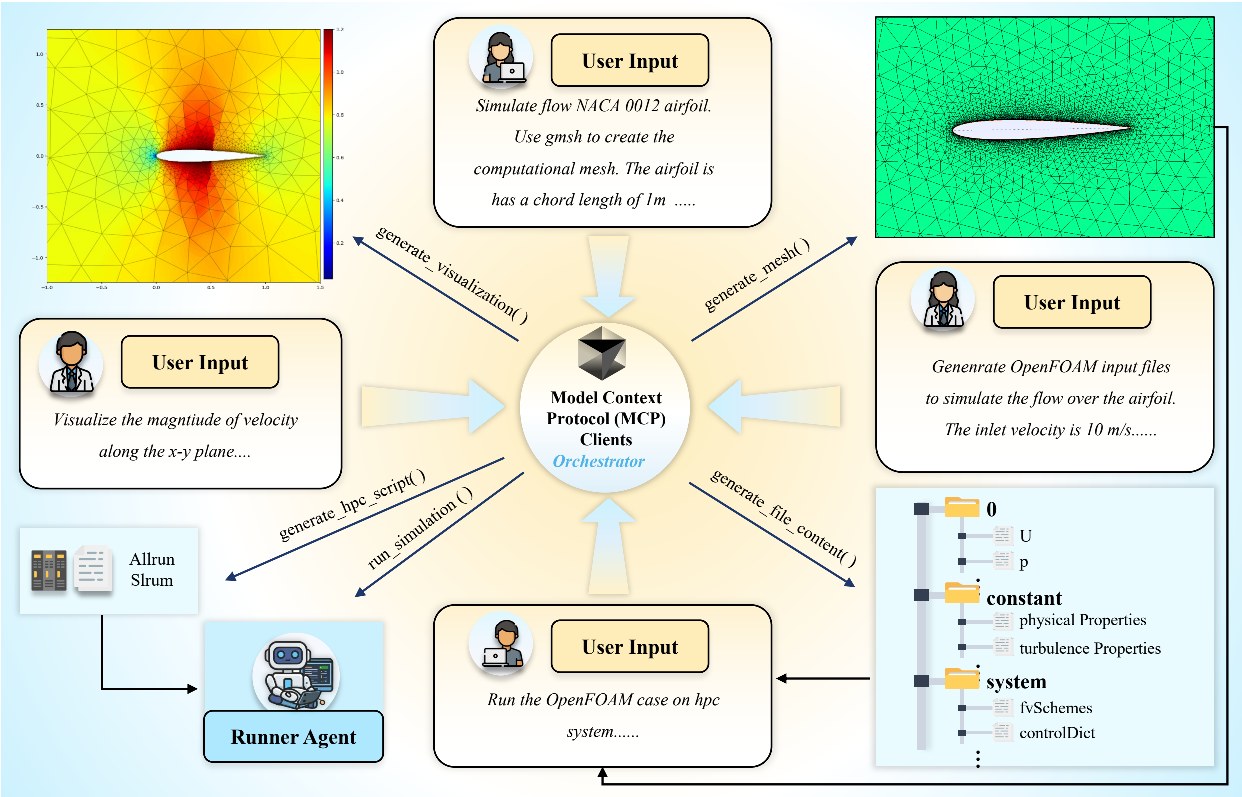


Figure 8: MCP-orchestrated end-to-end simulation of NACA0012 flow. The orchestrator invokes `generate_mesh()`, `generate_file_content()`, `generate_hpc_script()`, `run_simulation()`, and `generate_visualization()` to complete the workflow from natural-language request to final result.

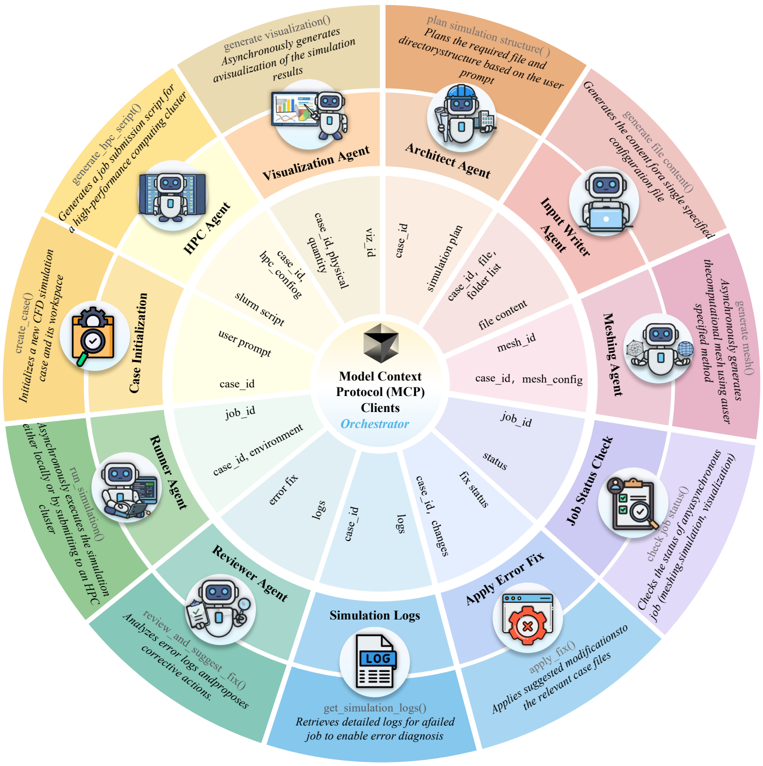


Figure 9: Design diagram for the MCP version of the Foam-Agent. Functionality of Foam-Agent is broken down to basic functions and exposed to an MCP orchestrator. The orchestrator can then decide which function to be called at anytime, given the user request.