Fully LLM-Driven Computational Fluid Dynamics: Multiphase Open Channel Flow Simulation Using Claude Opus 4.6 and OpenFOAM v10

A Step-by-Step Tutorial for CFD Experts and Newcomers

Terragon.de*

February 2026

Abstract

This paper documents a complete computational fluid dynamics (CFD) workflow executed entirely through plain-English interaction with the large language model Claude Opus 4.6 by Anthropic. The simulation investigates free-surface water flow around two bluff-body obstacles—a circular cylinder and a square prism—placed in two independent parallel open channels, solved with the Volume-of-Fluid (VOF) method and Large Eddy Simulation (LES) turbulence modelling in OpenFOAM v10. The geometry and mesh were created with the Gmsh Python API using OpenCASCADE boolean operations, producing a tetrahedral mesh of 133 946 cells. The paper is written as a detailed tutorial: every OpenFOAM directory, every configuration file, every setting choice, and every bash command used throughout the workflow is documented and explained. Two numerical instabilities encountered during the simulation and their autonomous diagnosis and repair by the LLM are described in detail. The results are visualised using ParaView. This paper is intended for both experienced CFD practitioners seeking to understand the LLM-driven workflow and newcomers who want to learn OpenFOAM case setup from the ground up. For more CFD tutorials and content, visit `http://www.terragon.de/cfd/` and `https://www.youtube.com/@TerragonCFD`.

Keywords: computational fluid dynamics, OpenFOAM, interFoam, Volume of Fluid, Large Eddy Simulation, open channel flow, bluff body, Gmsh, large language model, Claude, tutorial

1 Introduction

Setting up a CFD simulation from scratch requires knowledge across multiple domains: geometry creation, mesh generation, physics selection, boundary condition specification, numerical scheme tuning, and iterative debugging when things go wrong. This paper demonstrates that a modern large language model (LLM)—Claude Opus 4.6 by Anthropic—can handle all of these stages autonomously, guided only by plain-English instructions from the user.

The simulation goal is straightforward: compare the flow of water around a **circular cylinder** and a **square prism**, each placed in its own open channel. The two channels are independent (no fluid exchange between them), allowing a direct side-by-side comparison of wake patterns, vortex shedding, and free-surface deformation.

*Website: `http://www.terragon.de/cfd/` – YouTube: `https://www.youtube.com/@TerragonCFD` – This manuscript, including all OpenFOAM case files, mesh generation scripts, and solver configurations, was produced entirely through plain-English interaction with Claude Opus 4.6 (Anthropic).

What this paper covers. This is not a typical research paper—it is a **complete tutorial** that documents every file, every command, and every decision made during the case setup. Whether you are an experienced OpenFOAM user curious about LLM-driven workflows, or a complete newcomer trying to understand how an OpenFOAM case is structured, this paper walks you through the entire process.

How it was created. The human operator (Terragon.de) issued plain-English instructions to Claude Opus 4.6 running in the Claude Code command-line interface. The LLM had direct access to the file system and a bash shell with OpenFOAM v10 installed. At no point did the human operator manually edit any configuration file, Python script, or shell command. This manuscript was also drafted by the LLM.

Where to find more. For additional CFD tutorials, case setups, and video walkthroughs, visit:

- `http://www.terragon.de/cfd/` – CFD articles and resources
- `https://www.youtube.com/@TerragonCFD` – Video tutorials

2 OpenFOAM Case Structure – The Three Directories

Every OpenFOAM case is organised into exactly three mandatory directories. Understanding this structure is the single most important concept for any OpenFOAM newcomer. If you understand these three directories, you understand how to read any OpenFOAM case:

1. `0/` – **Initial and boundary conditions.** Contains one file per field variable (velocity, pressure, phase fraction, etc.). Each file specifies the initial value throughout the domain and the boundary condition on every patch (inlet, outlet, walls, etc.). The "0" stands for time $t = 0\,\text{s}$—it is the starting point of the simulation.

2. `constant/` – **Physical properties and mesh.** Contains the mesh (in the `polyMesh/` subdirectory) and files that define the physical properties of the fluids, the turbulence model, the gravitational acceleration, and the phases present. These properties do not change during the simulation—hence "constant".

3. `system/` – **Solver and numerics control.** Contains files that tell OpenFOAM *how* to solve the equations: which solver to use, how large the time steps should be, which discretisation schemes to apply, and how to write output. This is where all the numerical "knobs" live.

As the simulation runs, OpenFOAM creates additional numbered directories (e.g. `0.05/`, `0.10/`, `2.10/`) containing the field data at those time instants.

The following sections explain every file in our case in detail.

3 Geometry and Mesh Generation

3.1 Domain Description

The computational domain consists of two rectangular open channels placed side by side in the y-direction:

Table 1: Geometric parameters of the two channels and obstacles.

Parameter	Left channel	Right channel
Channel dimensions ($L \times W \times H$)	$1.0\,\mathrm{m} \times 0.3\,\mathrm{m} \times 0.3\,\mathrm{m}$	
y-range	0 to 0.3 m	0.3 m to 0.6 m
Obstacle shape	Cylinder	Square prism
Characteristic dim. d	150 mm diameter	150 mm side
Obstacle height h_o	150 mm	150 mm
Centre position (x, y)	(0.5, 0.15) m	(0.5, 0.45) m
Blockage ratio d/W	0.50	0.50

The coordinate system: x = streamwise (flow direction), y = spanwise (across channels), z = vertical (up, opposing gravity). Both obstacles sit on the channel floor ($z = 0$) and extend to $z = 0.15\,\mathrm{m}$—they are submerged pillars, not spanning the full channel height.

3.2 Mesh Generation with Gmsh Python API

The mesh was generated using a Python script (`mesh.py`) that uses the Gmsh 4.15.1 Python API with the OpenCASCADE (OCC) geometry kernel. The key steps are:

1. **Create channel boxes:** Two rectangular boxes via `occ.addBox()`—left channel at $y \in [0, 0.3]$, right channel at $y \in [0.3, 0.6]$.
2. **Create obstacles:** A cylinder via `occ.addCylinder()` (radius 75 mm, height 150 mm) and a box via `occ.addBox()` (150 mm side, 150 mm tall).
3. **Boolean subtraction:** `occ.cut()` removes each obstacle volume from its channel, leaving the fluid domain with the obstacle-shaped holes.
4. **Surface classification:** After boolean operations, Gmsh produces 19 boundary surfaces. The script classifies each surface into one of 9 named groups (inlet, outlet, atmosphere, ground, wallLeft, wallRight, wallMiddle, cylinder, square) by inspecting bounding-box coordinates.
5. **Mesh sizing:** A distance-based field provides refinement near obstacle surfaces: 10 mm minimum cell size within 20 mm of the objects, grading to 25 mm in the bulk.
6. **Mesh generation:** Delaunay tetrahedral meshing with Gmsh and Netgen optimisers.

The final mesh contains 133 946 **tetrahedral cells** and 27 602 nodes.

Table 2: Boundary patch names, types, and face counts.

Patch name	Type	Faces	Description
`inlet`	patch	692	Left face ($x = 0$), water enters here
`outlet`	patch	688	Right face ($x = 1\,\mathrm{m}$), water exits
`atmosphere`	patch	2 587	Top face ($z = 0.3\,\mathrm{m}$), open to air
`ground`	wall	4 270	Bottom face ($z = 0$), channel floor
`wallLeft`	wall	1 589	Side wall ($y = 0$)
`wallRight`	wall	1 756	Side wall ($y = 0.6\,\mathrm{m}$)
`wallMiddle`	wall	3 516	Dividing wall ($y = 0.3\,\mathrm{m}$)
`cylinder`	wall	2 104	Cylinder obstacle surface
`square`	wall	2 718	Square prism obstacle surface

3.3 Bash Commands: Mesh Generation and Conversion

The following commands were used to generate the mesh and convert it to OpenFOAM format. First, source the OpenFOAM environment:

```
source /opt/openfoam10/etc/bashrc
```

Generate the mesh with the Python script:

```
python3 mesh.py
```

This produces `doppelterWindkanal.msh` in MSH 2.2 ASCII format.

Convert the Gmsh mesh to OpenFOAM polyMesh format:

```
gmshToFoam doppelterWindkanal.msh
```

Important: After `gmshToFoam`, all patches are assigned `type patch` by default. Solid boundaries must be changed to `type wall` for correct wall function behaviour. This was done with a Python one-liner that patches the `constant/polyMesh/boundary` file:

```
python3 << 'PYEOF'
import re
with open("constant/polyMesh/boundary", "r") as f:
    content = f.read()
wall_patches = ["ground", "wallLeft", "wallRight",
                "wallMiddle", "cylinder", "square"]
for patch in wall_patches:
    pattern = (r"(\b" + patch +
              r"\b\s*\{[^}]*type\s+)patch(\s*;)")
    content = re.sub(pattern, r"\1wall\2",
                     content, flags=re.DOTALL)
with open("constant/polyMesh/boundary", "w") as f:
    f.write(content)
PYEOF
```

Verify the mesh quality:

```
checkMesh
```

The `checkMesh` output confirmed all quality metrics within acceptable bounds: maximum non-orthogonality 51.2 (limit 70), maximum skewness 0.62 (limit 4.0), maximum aspect ratio 4.57. The mesh correctly contains two disconnected regions (the two independent channels).

4 The `constant/` Directory – Physical Properties

The `constant/` directory holds files that define the physics of the problem. These values remain fixed throughout the simulation.

4.1 `constant/g` – Gravitational Acceleration

This file defines the gravitational acceleration vector. In our setup, gravity acts in the negative z-direction:

```
dimensions      [0 1 -2 0 0 0 0];
value           (0 0 -9.81);
```

Why: The z-axis points upward in our geometry, so gravity is $(0, 0, -9.81)\,\mathrm{m/s^2}$. The dimensions `[0 1 -2 0 0 0 0]` correspond to $\mathrm{m/s^2}$ in OpenFOAM's dimensional system $[\mathrm{kg}, \mathrm{m}, \mathrm{s}, \mathrm{K}, \mathrm{mol}, \mathrm{A}, \mathrm{cd}]$.

4.2 `constant/phaseProperties` – Phase Definition

Tells `interFoam` which two phases exist and the surface tension between them:

```
phases (water air);
sigma  0.072;       // surface tension [N/m]
```

Why: We simulate water and air. The surface tension $\sigma = 0.072\,\mathrm{N/m}$ is the standard value for a water–air interface at approximately 20 C. This value affects the interface curvature force in the momentum equation.

4.3 `constant/physicalProperties.water` – Water Properties

```
viscosityModel  constant;
nu  [0 2 -1 0 0 0 0]  1e-6;    // kinematic viscosity [m^2/s]
rho [1 -3 0 0 0 0 0]  999;     // density [kg/m^3]
```

Why: Standard water at ∼20 C. The kinematic viscosity $\nu = 10^{-6}\,\mathrm{m^2/s}$ directly determines the Reynolds number: $Re = U \cdot d/\nu = 0.1 \times 0.15/10^{-6} = 15{,}000$.

4.4 `constant/physicalProperties.air` – Air Properties

```
viscosityModel  constant;
nu  [0 2 -1 0 0 0 0]  1.48e-05;  // kinematic viscosity [m^2/s]
rho [1 -3 0 0 0 0 0]  1.225;     // density [kg/m^3]
```

Why: Standard air at sea level and ∼15 C. The density ratio $\rho_{\text{water}}/\rho_{\text{air}} \approx 816$ is what makes multiphase simulations numerically challenging—small errors in the interface position cause large density jumps that amplify pressure oscillations.

4.5 `constant/momentumTransport` – Turbulence Model

```
simulationType  LES;
LES
{
    model       kEqn;      // one-equation SGS model
    turbulence  on;
    printCoeffs on;
    delta       smooth;    // filter width: smoothed cubeRootVol
    // ... coefficient sub-dictionaries ...
}
```

Why LES with kEqn?

- At $Re = 15{,}000$, the flow is turbulent with vortex shedding. LES resolves the large energy-carrying eddies directly and only models the small subgrid-scale (SGS) motions, giving much better wake predictions than RANS models.

- The `kEqn` model is the simplest LES SGS model: it solves one additional transport equation for the subgrid-scale kinetic energy k. The SGS viscosity is then $\nu_t = C_k\sqrt{k}\,\Delta$, where Δ is the filter width (cube root of cell volume with smooth blending).
- The `smooth` delta option blends the cube-root-volume filter width to avoid abrupt changes at mesh refinement boundaries, which would introduce artificial turbulence production.

5 The `0/` Directory – Initial and Boundary Conditions

The `0/` directory contains one file per field variable. Each file has two main sections: `internalField` (the initial value throughout the entire domain) and `boundaryField` (the condition on each named boundary patch). Our case has six files:

5.1 `0/alpha.water` – Phase Fraction

The volume fraction α: where $\alpha = 1$ is pure water, $\alpha = 0$ is pure air, and values in between represent the interface.

```
internalField   uniform 0;  // domain starts filled with air
boundaryField
{
    inlet      { type inletOutlet; inletValue uniform 1;
                 value uniform 1; }
    outlet     { type inletOutlet; inletValue uniform 0;
                 value uniform 0; }
    atmosphere { type inletOutlet; inletValue uniform 0;
                 value uniform 0; }
    ground     { type zeroGradient; }
    // ... all other walls: zeroGradient
}
```

Why these choices:

- `internalField uniform 0`: The channels start completely filled with air. Water enters from the inlet, creating a dramatic filling transient.
- `inlet - inletOutlet, inletValue 1`: When flow enters (which is the normal condition), it brings pure water ($\alpha = 1$). The `inletOutlet` type automatically switches to zero-gradient if flow reverses.
- `outlet/atmosphere - inletOutlet, inletValue 0`: If backflow occurs at the outlet or atmosphere, only air ($\alpha = 0$) can re-enter. This prevents unphysical water re-entry from the open boundaries.
- `walls - zeroGradient`: The phase fraction has no special wall behaviour; it simply takes whatever value the adjacent cell has.

5.2 `0/U` – Velocity

```
internalField   uniform (0 0 0);  // stationary start
boundaryField
{
    inlet      { type surfaceNormalFixedValue;
                 refValue uniform -0.1; }
    outlet     { type pressureInletOutletVelocity;
                 value uniform (0 0 0); }
    atmosphere { type pressureInletOutletVelocity;
                 value uniform (0 0 0); }
    ground     { type noSlip; }
    // ... all other walls: noSlip
    cylinder   { type noSlip; }
    square     { type noSlip; }
}
```

Why these choices:

- `inlet - surfaceNormalFixedValue, refValue -0.1`: Prescribes 0.1 m/s normal to the inlet face, pointing *inward* (hence the negative sign, since face normals point outward). This gives a volumetric flow rate of $Q = 0.1 \times 0.3 \times 0.3 = 0.009\,\mathrm{m}^3/\mathrm{s} \approx 9\,\mathrm{l/s}$ per channel.
- `outlet/atmosphere - pressureInletOutletVelocity`: Allows the velocity to adjust freely based on the pressure field. This is standard for open boundaries where pressure is prescribed.
- `All walls - noSlip`: Zero velocity at solid surfaces. This is the fundamental wall boundary condition in viscous flow.

Why 0.1 m/s**?** The inlet velocity was chosen to satisfy three criteria simultaneously: (1) $Re = Ud/\nu = 0.1 \times 0.15/10^{-6} = 15{,}000$, which is turbulent but not extremely so; (2) the Froude number $Fr = U/\sqrt{gh} \approx 0.07 \ll 1$ (subcritical open channel flow, no hydraulic jumps); (3) the resulting adaptive time steps are manageable on a laptop CPU.

5.3 `0/p_rgh` – Modified Pressure

The variable $p_{\mathrm{rgh}} = p - \rho\mathbf{g}\cdot\mathbf{x}$ is the pressure minus the hydrostatic component. Using p_{rgh} instead of absolute pressure p improves numerical stability in flows with gravity.

```
internalField   uniform 0;
boundaryField
{
    inlet      { type fixedFluxPressure; value uniform 0; }
    outlet     { type prghTotalPressure; p0 uniform 0; }
    atmosphere { type prghTotalPressure; p0 uniform 0; }
    ground     { type fixedFluxPressure; value uniform 0; }
    // ... all other walls: fixedFluxPressure
}
```

Why these choices:

- `inlet/walls - fixedFluxPressure`: The pressure gradient at these boundaries is adjusted so that the boundary flux matches the velocity boundary condition. This correctly accounts for the gravitational body force on wall and inlet patches.
- `outlet/atmosphere - prghTotalPressure, p0 = 0`: Sets the total pressure (static + dynamic) to zero gauge at the open boundaries. The `prgh` variant correctly handles the hydrostatic pressure decomposition.

5.4 `0/k` – Subgrid-Scale Kinetic Energy

Used by the LES `kEqn` model. Represents the kinetic energy of the unresolved (subgrid-scale) turbulent motions.

```
internalField   uniform 0;
boundaryField
```

```
{
    inlet      { type fixedValue;   value uniform 0; }
    outlet     { type inletOutlet;  inletValue uniform 0;
                 value uniform 0; }
    atmosphere { type inletOutlet;  inletValue uniform 0;
                 value uniform 0; }
    ground     { type fixedValue;   value uniform 0; }
    // ... all other walls: fixedValue 0
}
```

Why $k = 0$ everywhere initially? The inlet flow is uniform with no turbulence prescribed. Turbulence will develop naturally as the flow interacts with the obstacles and develops shear layers. Setting $k = 0$ at walls is appropriate for LES where the grid is expected to resolve the near-wall turbulence (no wall modelling).

5.5 `0/nut` – Turbulent Eddy Viscosity

The SGS viscosity computed by the turbulence model. Not a solved field—it is derived from k and the filter width Δ.

```
internalField   uniform 0;
boundaryField
{
    inlet      { type zeroGradient; }
    // ... all patches: zeroGradient
}
```

Why zeroGradient? The eddy viscosity is a derived quantity. Zero-gradient lets the solver compute it freely based on the resolved flow field without imposing artificial constraints at the boundaries.

5.6 `0/nuTilda` – Spalart-Allmaras Variable

This field was inherited from the original case template. It is used by the Spalart-Allmaras turbulence model, which is *not* active in our LES simulation. OpenFOAM requires the file to exist if the solver references it, but its values have no effect on the solution with the `kEqn` model.

```
internalField   uniform 0;
boundaryField
{
    inlet      { type fixedValue;   value uniform 0; }
    outlet     { type inletOutlet;  inletValue uniform 0; }
    // ... walls: fixedValue 0
}
```

6 The `system/` Directory – Solver Control

The `system/` directory contains three mandatory files (`controlDict`, `fvSchemes`, `fvSolution`) plus optional ones. These files control *how* the equations are solved.

6.1 `system/controlDict` – Run Control

This is the master control file. It tells OpenFOAM which solver to use, when to start and stop, how often to write output, and how to manage time stepping.

```
application     interFoam;     // VOF multiphase solver

startFrom       latestTime;    // resume from last saved step
startTime       0;
stopAt          endTime;
endTime         10;            // simulate 10 seconds

deltaT          0.001;         // initial time step [s]

writeControl    adjustableRunTime;
writeInterval   0.05;          // save output every 0.05s
purgeWrite      0;             // keep all output directories

writeFormat     binary;        // faster I/O, smaller files
writePrecision  8;

adjustTimeStep  yes;           // adaptive time stepping
maxCo           0.2;           // max Courant number
maxAlphaCo      0.2;           // max interface Courant number
maxDeltaT       0.05;          // upper bound on time step

runTimeModifiable yes;         // re-read this file while running
```

Why these settings:

- `application interFoam`: The standard VOF solver for two immiscible, incompressible fluids in OpenFOAM. It solves the Navier-Stokes equations with a volume fraction transport equation to track the water–air interface.
- `deltaT 0.001`: A small initial time step to survive the violent initial transient when water first enters the air-filled domain. Larger values caused immediate divergence (see Section 8).
- `adjustTimeStep yes, maxCo 0.2, maxAlphaCo 0.2`: Adaptive time stepping. The Courant number $\text{Co} = U \cdot \Delta t / \Delta x$ must stay below 1 for stability; we use 0.2 for extra safety margin with the VOF method. The `maxAlphaCo` additionally limits the interface Courant number, which is critical because the interface can move faster than the bulk flow due to compression velocity.
- `writeFormat binary`: Binary output is faster to write and read, and the files are smaller. With small mesh cells, the default ASCII precision of 6 digits may not be sufficient to represent cell coordinates accurately—binary avoids this issue.
- `writePrecision 8`: Eight significant figures for any ASCII fields. Important for meshes with small elements.
- `writeInterval 0.05`: We want detailed temporal resolution for post-processing and animation—one output frame every 50 ms of simulated time.
- `runTimeModifiable yes`: Allows editing this file while the simulation is running. We used this to change `writeInterval` from 0.5 to 0.05 during the run without restarting.

6.2 `system/fvSchemes` – Discretisation Schemes

This file specifies how partial differential equations are discretised—i.e., how continuous derivatives are approximated on the finite-volume mesh.

```
ddtSchemes              // Time derivatives
{
    default         Euler;
}

gradSchemes             // Gradient terms
{
    default         Gauss linear;
}

divSchemes              // Divergence (convection) terms
{
    div(rhoPhi,U)   Gauss limitedLinear 1;
    div(phi,alpha)  Gauss interfaceCompression vanLeer 1;
    div(phi,k)      Gauss limitedLinear 1;
    div(phi,B)      Gauss limitedLinear 1;
    div(B)          Gauss linear;
    div(phi,nuTilda) Gauss limitedLinear 1;
    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes        // Laplacian (diffusion) terms
{
    default         Gauss linear corrected;
}

interpolationSchemes    // Cell-to-face interpolation
{
    default         linear;
}

snGradSchemes           // Surface-normal gradients
{
    default         corrected;
}
```

Why these choices – explained for newcomers:

- **ddtSchemes – Euler:** First-order implicit time integration. It is the most stable choice, which matters during the violent filling transient. Higher-order schemes like Crank-Nicolson (2nd order) are more accurate but can introduce oscillations in transient multiphase flows.

- **gradSchemes – Gauss linear:** Standard second-order gradient calculation using Green-Gauss theorem with linear interpolation. Accurate and stable for our mesh quality.

- **div(rhoPhi,U) – Gauss limitedLinear 1:** *This was the most critical scheme choice in the entire case.* The momentum convection term was *initially* set to `Gauss linear` (unbounded 2nd order), which caused the simulation to crash when the water front hit the obstacles (Section 8). The `limitedLinear 1` scheme is a TVD (Total Variation Diminishing) limiter that blends between 1st and 2nd order: it uses 2nd order in smooth regions for accuracy, but automatically reduces to 1st order at sharp gradients to prevent oscillations. The “1” means full limiting is active.

- **div(phi,alpha) – Gauss interfaceCompression vanLeer 1:** Specialised scheme for the VOF phase fraction transport. The `interfaceCompression` part adds an artificial compression velocity to keep the water–air interface sharp (preventing numerical smearing). The `vanLeer` limiter ensures boundedness ($0 \leq \alpha \leq 1$).

- **laplacianSchemes – Gauss linear corrected:** The diffusion terms are discretised with Gaussian integration and a correction for mesh non-orthogonality. The "corrected" option adds an explicit non-orthogonal correction, which is important for our tetrahedral mesh where cells are not aligned with coordinate axes. This works well up to about 70° non-orthogonality (our max is 51.2°).
- **snGradSchemes – corrected:** Surface-normal gradients also use the non-orthogonal correction, matching the laplacian scheme choice.

6.3 `system/fvSolution` – Linear Solver Settings

This file configures how the linear equation systems arising from discretisation are solved at each time step.

```
solvers
{
    alpha.water
    {
        nAlphaCorr      1;     // alpha correction steps
        nAlphaSubCycles 4;     // sub-cycles for alpha eq.
    }

    p_rgh
    {
        solver     GAMG;       // multigrid pressure solver
        tolerance  1e-07;
        relTol     0.01;
        smoother   DIC;
    }

    p_rghFinal
    {
        solver     PCG;        // conjugate gradient
        preconditioner
        {
            preconditioner  GAMG;
            tolerance       1e-07;
            relTol          0;
            nVcycles        2;
            smoother        DICGaussSeidel;
            nPreSweeps      2;
        }
        tolerance  1e-07;
        relTol     0;          // solve to absolute tolerance
        maxIter    20;
    }

    "(U|k)"
    {
        solver     smoothSolver;
        smoother   GaussSeidel;
        tolerance  1e-06;
        relTol     0.1;
        nSweeps    1;
    }

    "(U|k)Final"
    {
        solver     smoothSolver;
        smoother   symGaussSeidel;
        tolerance  1e-08;
        relTol     0;
    }
}

PIMPLE
{
    momentumPredictor no;
    nCorrectors     2;
    nNonOrthogonalCorrectors 2;
}
```

Why these settings:

- **alpha.water sub-cycling:** `nAlphaSubCycles 4` means the phase fraction equation is solved 4 times per outer time step, each with $\Delta t/4$. This provides better interface resolution without reducing the global time step.
- **Pressure solver – GAMG + PCG:** The pressure equation is the most expensive part of each time step. For the predictor step (`p_rgh`), GAMG (Geometric Algebraic MultiGrid) is used with a relaxed tolerance (`relTol 0.01`). For the final corrector (`p_rghFinal`), PCG (Preconditioned Conjugate Gradient) with GAMG as preconditioner is used, solving to absolute tolerance (`relTol 0`). This two-stage approach balances speed (relaxed early solves) with accuracy (tight final solve).
- **Velocity/k solver – smoothSolver:** The momentum and turbulence equations are well-conditioned and solve quickly with Gauss-Seidel smoothing. The "Final" variants use symmetric Gauss-Seidel and tighter tolerances.
- **PIMPLE algorithm:** PIMPLE is a combination of PISO (Pressure Implicit with Splitting of Operators) and SIMPLE (Semi-Implicit Method for Pressure-Linked Equations). With `nCorrectors 2`, the pressure-velocity coupling is corrected twice per time step.
- **momentumPredictor no:** Disabling the momentum predictor step is common for multiphase flows and low-Reynolds-number regions, where it can cause instability.
- **nNonOrthogonalCorrectors 2:** *Increased from 1 to 2* after a crash (Section 8). Each non-orthogonal corrector re-solves the pressure Laplacian with updated explicit corrections for mesh non-orthogonality. Tetrahedral meshes have higher non-orthogonality than hexahedral meshes, requiring more corrections.

7 The `Allrun` Script

The complete workflow is automated in a single shell script:

```
#!/bin/bash
cd "${0%/*}" || exit 1
. ${WM_PROJECT_DIR:?}/bin/tools/RunFunctions

# 1. Generate mesh with gmsh
echo "=== Generating mesh with gmsh ==="
python3 mesh.py

# 2. Convert gmsh mesh to OpenFOAM format
echo "=== Converting mesh to OpenFOAM format ==="
runApplication gmshToFoam doppelterWindkanal.msh

# 3. Fix boundary types: wall patches
echo "=== Fixing boundary types ==="
python3 << 'PYEOF'
import re
with open("constant/polyMesh/boundary", "r") as f:
    content = f.read()
wall_patches = ["ground", "wallLeft", "wallRight",
                "wallMiddle", "cylinder", "square"]
for patch in wall_patches:
    pattern = (r"(\b" + patch +
              r"\b\s*\{[^}]*type\s+)patch(\s*;)")
    content = re.sub(pattern, r"\1wall\2",
                     content, flags=re.DOTALL)
with open("constant/polyMesh/boundary", "w") as f:
    f.write(content)
PYEOF

# 4. Run the solver
echo "=== Running interFoam ==="
runApplication interFoam
```

The `runApplication` wrapper (from OpenFOAM's `RunFunctions`) redirects stdout and stderr to a log file (e.g. `log.interFoam`).

7.1 All Bash Commands Used During the Session

Beyond the Allrun script, the following commands were executed during the interactive session with Claude Opus 4.6:

```
# Source OpenFOAM environment
source /opt/openfoam10/etc/bashrc

# Generate mesh
python3 mesh.py

# Convert to OpenFOAM format
gmshToFoam doppelterWindkanal.msh

# Fix wall boundary types (Python script, see above)

# Check mesh quality
```

```
checkMesh

# Run the solver (initial attempt)
interFoam > log.interFoam 2>&1 &

# Monitor the running simulation
tail -f log.interFoam
grep "^Time = " log.interFoam | tail -5
ls -ltr | grep -E "^d" | tail -5

# Kill crashed simulation
kill %1

# Restart from last saved time step after fixes
interFoam > log.interFoam 2>&1 &

# Check latest simulation time
ls | grep -E "^[0-9]" | sort -n | tail -5

# Stop simulation
pkill -f interFoam

# Compile this paper
pdflatex paper_doppelter_windkanal.tex
pdflatex paper_doppelter_windkanal.tex  # run twice for refs
```

8 Numerical Instabilities and How They Were Fixed

Two separate crashes occurred during the simulation. Both were diagnosed and fixed autonomously by Claude Opus 4.6. Understanding these failures is instructive for any CFD practitioner.

8.1 Crash 1: Courant Number Divergence During Initial Transient

What happened: The user asked to speed up the simulation. The LLM increased `maxCo` and `maxAlphaCo` from 0.1 to 0.5. The simulation diverged within 0.001 s—continuity errors jumped from $\mathcal{O}(10^{-10})$ to $\mathcal{O}(10^{23})$ in two time steps.

Why it crashed: The adaptive time-stepping controller, seeing a maximum Courant number target of 0.5, allowed $\Delta t \approx 0.01\,\mathrm{s}$. During the initial transient, water ($\rho = 999\,\mathrm{kg/m^3}$) slams into air ($\rho = 1.225\,\mathrm{kg/m^3}$) —a density ratio of 816:1. With such a large time step, the interface moved through multiple cells per step, violating the CFL condition for VOF transport and causing the pressure solver to diverge.

Fix: Reduced `deltaT` from 0.01 to 0.001 s and `maxCo`/`maxAlphaCo` from 0.5 to 0.3.

8.2 Crash 2: Unbounded Scheme at Obstacle Impact

What happened: At $t \approx 0.578\,\mathrm{s}$, the simulation crashed with a floating-point exception (`sigFpe`) in the GAMG pressure solver. This was exactly when the water front reached the obstacles.

Why it crashed: The momentum divergence scheme `Gauss linear` is an unbounded second-order central-difference scheme. When the water front hit the obstacle surfaces, sharp velocity gradients developed. On our coarse tetrahedral mesh, the unbounded scheme produced Gibbs-like oscillations in velocity, which propagated into the pressure equation and generated negative values of the subgrid kinetic energy k (minimum $k = -0.0042$). Tetrahedral meshes worsen this because they have higher non-orthogonality and less numerical dissipation than hexahedral meshes.

Fix (three simultaneous changes):

1. `div(rhoPhi,U)`: Changed from `Gauss linear` to `Gauss limitedLinear 1` (TVD-limited, prevents oscillations)
2. `maxCo`/`maxAlphaCo`: Reduced from 0.3 to 0.2
3. `nNonOrthogonalCorrectors`: Increased from 1 to 2

The simulation was restarted from the last saved time step ($t = 0.55\,\mathrm{s}$) and ran stably past the former crash point.

Table 3: Evolution of solver parameters across three iterations.

Parameter	Attempt 1	Attempt 2	Attempt 3 (final)
$\mathrm{Co}_{\mathrm{max}}$	0.1	0.5	0.2
$\mathrm{Co}_{\alpha,\mathrm{max}}$	0.1	0.5	0.2
Δt_0 [s]	0.01	0.01	0.001
div(rhoPhi,U)	linear	linear	limitedLinear 1
nNonOrthCorr	1	1	2
Result	Stable (slow)	Crash	Stable

Lesson for practitioners: In multiphase VOF simulations on tetrahedral meshes, *always* use bounded convection schemes (e.g. `limitedLinear`) for the momentum equation. Unbounded second-order schemes that work fine on hexahedral meshes can fail catastrophically on tetrahedra, especially at sharp phase interfaces.

9 Results and Visualisation

The simulation was run to $t = 2.6\,\mathrm{s}$ before being stopped. The following screenshots were taken at $t = 2.1\,\mathrm{s}$ using ParaView, an open-source scientific visualisation tool. Streamlines are coloured by velocity magnitude and the water phase ($\alpha > 0.5$) is shown as the red/orange volume.

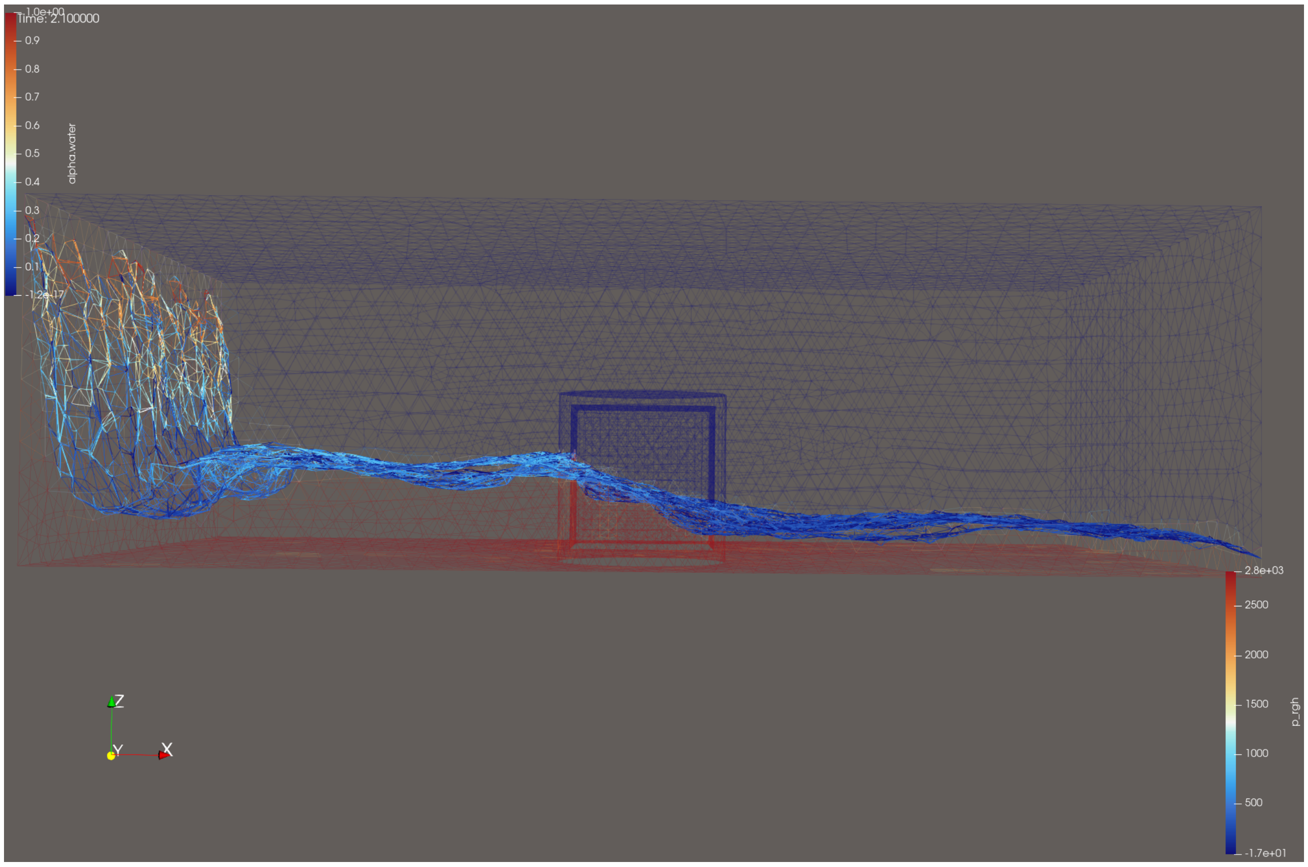

Figure 1: Side view at $t = 2.1\,\mathrm{s}$ showing the channel with the square prism obstacle. Streamlines illustrate the flow pattern around and over the obstacle. The water level (red region) has risen to approximately $0.15\,\mathrm{m}$ upstream of the object, with the flow separating at the sharp upstream edge and reattaching downstream. The streamlines clearly show recirculation behind the square prism.

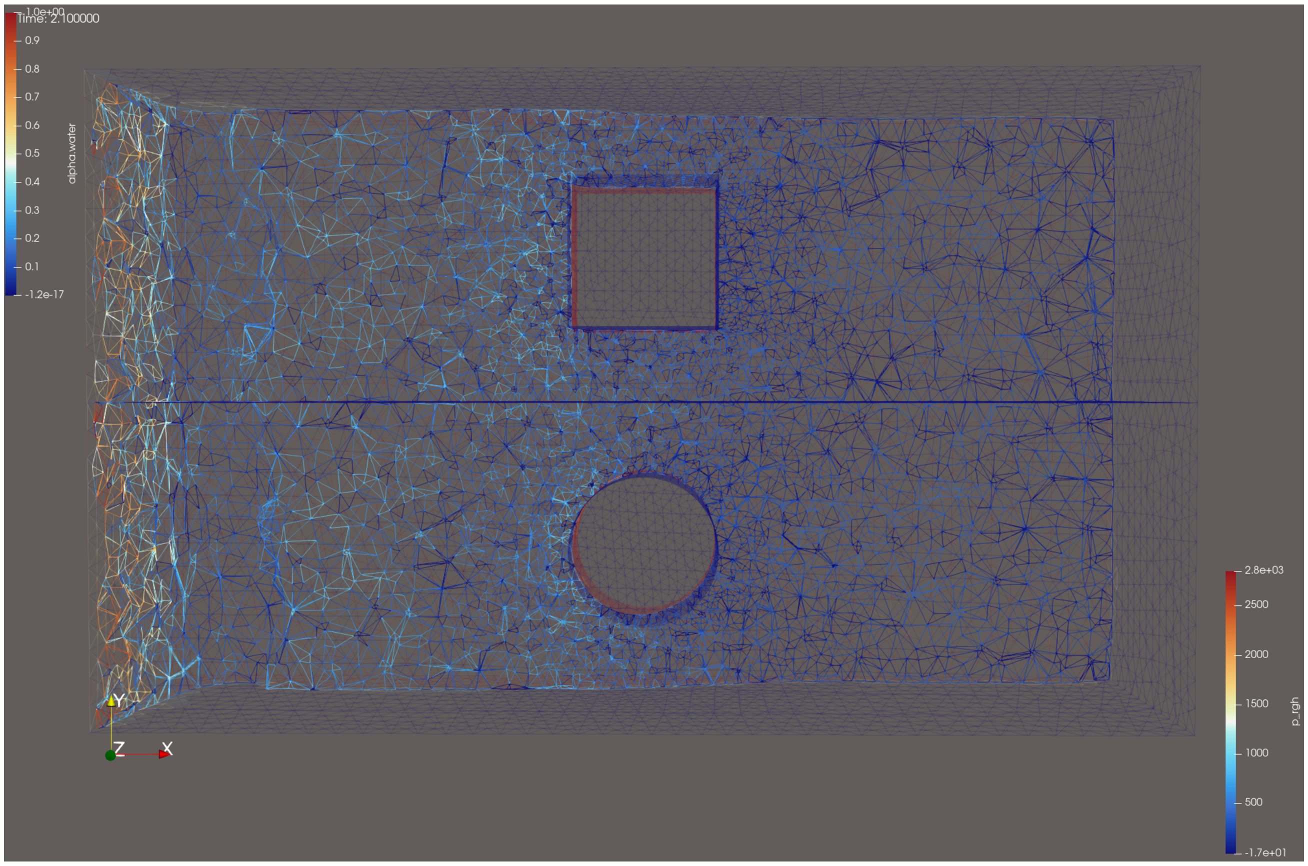

Figure 2: Top view at $t = 2.1\,\mathrm{s}$ showing both channels simultaneously. The bottom channel contains the cylinder, the top channel the square prism. Streamlines and velocity colouring reveal the distinct wake patterns: the cylinder produces a narrower, more symmetric wake, while the square prism creates a wider separation region with stronger lateral deflection due to its sharp corners. Both wakes show turbulent structures downstream of the obstacles.

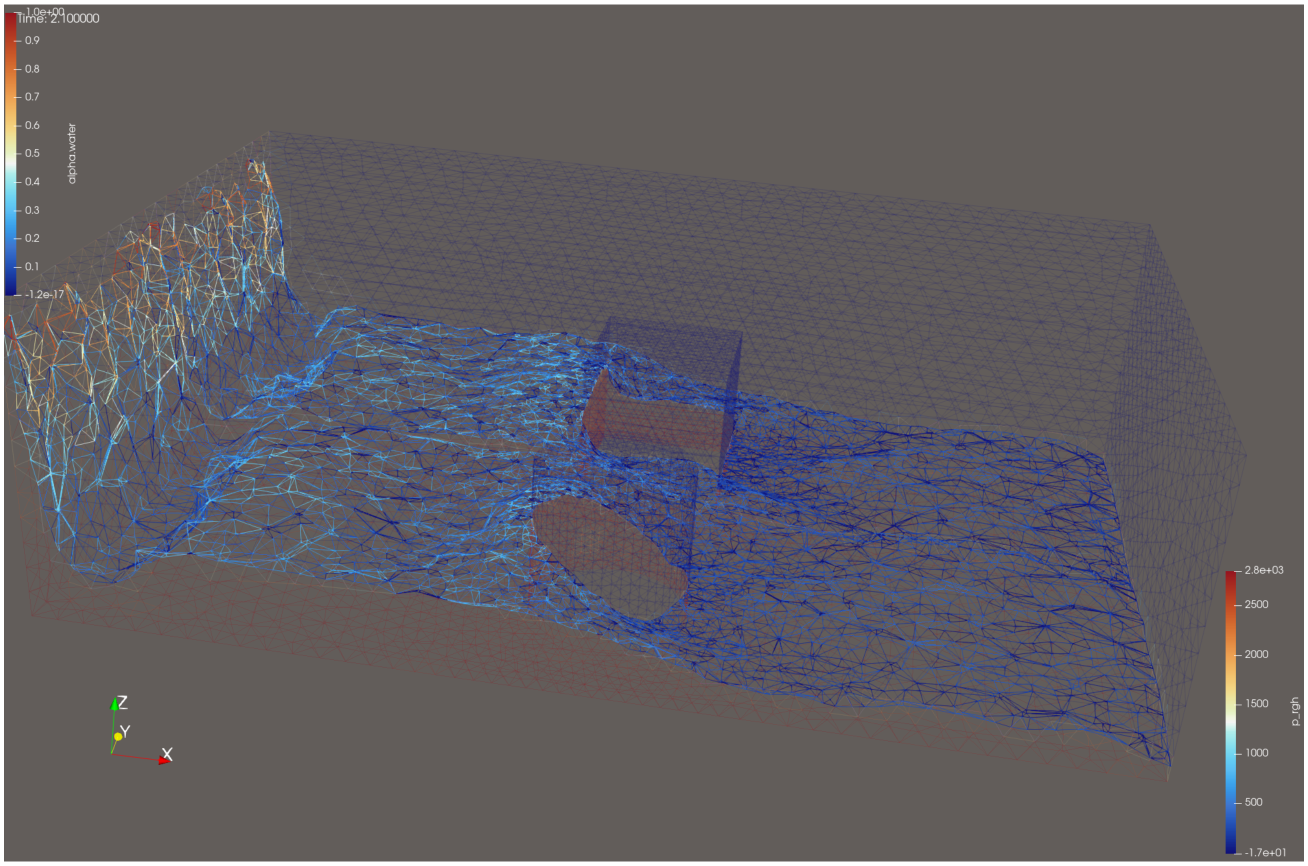

Figure 3: 3D perspective view at $t = 2.1\,\mathrm{s}$ showing both obstacles with streamlines. The water phase (orange/red volume) is visible in the upstream portion of both channels. The streamlines show how the flow accelerates around the sides of both obstacles (higher velocity = warmer colours) and decelerates in the wake region downstream. The square prism creates a noticeably larger disturbance than the cylinder, as expected from its blunter cross-section and fixed separation points.

9.1 Flow Features at $t = 2.1\,\mathrm{s}$

At this point in the simulation, the following features are visible:

- **Filling transient:** Water has entered from the inlet and advanced past both obstacles, filling approximately 60% of the channel length. The water level is higher upstream of the obstacles due to the blockage effect.
- **Wake structures:** Both obstacles produce distinct wakes. The square prism shows a wider wake with fixed separation at its sharp corners, while the cylinder shows a narrower, more streamlined wake.
- **Flow acceleration:** Streamlines compress and accelerate as they pass around the sides of the obstacles (blockage ratio 50%), with peak velocities approximately double the inlet velocity.
- **Free-surface deformation:** The water surface is not flat—it shows deformation around the obstacles, with a slight rise upstream and depression downstream.

10 Conclusions

This paper has documented a complete OpenFOAM v10 CFD workflow—from an empty directory to converged multiphase simulation results—driven entirely by plain-English interaction with Claude Opus 4.6.

For CFD experts: The key takeaway is that modern LLMs can serve as competent CFD assistants. The model correctly handled VOF setup with interFoam, chose appropriate boundary conditions, diagnosed two distinct numerical instabilities (Courant-number divergence and unbounded-scheme oscillations on tet mesh), and applied physically motivated fixes. Its one misstep—setting `maxCo = 0.5` for a multiphase filling transient—reflects a lack of the conservative instinct that experienced practitioners develop for VOF stability margins.

For newcomers: Every OpenFOAM case boils down to three directories: `0/` (what the flow looks like at the start), `constant/` (what the fluid is), and `system/` (how to solve it). The most common pitfalls are: incorrect boundary types after mesh conversion (always check!), too-aggressive Courant numbers for multiphase flows, and unbounded convection schemes on unstructured meshes. If your simulation crashes, check the divergence scheme first.

Reproducibility: All files shown in this paper—the Python mesh script, all OpenFOAM dictionaries, and the Allrun script—are the actual files used in the simulation, reproduced verbatim. The entire case can be run on a laptop with OpenFOAM v10 and Gmsh installed.

Where to learn more: For more CFD tutorials, case setups, and video walkthroughs:
- `http://www.terragon.de/cfd/`
- `https://www.youtube.com/@TerragonCFD`

Acknowledgements

The entire simulation workflow and this manuscript were produced through interaction with Claude Opus 4.6 (Anthropic, 2025–2026). The human author (Terragon.de) provided the problem specification and reviewed the final text.

References

[1] Anthropic. *Claude Opus 4.6 – Model Card and System Documentation.* Anthropic, San Francisco, CA, 2025. `https://www.anthropic.com`

[2] OpenFOAM Foundation. *OpenFOAM – The Open Source CFD Toolbox, Version 10.* `https://openfoam.org`, 2023.

[3] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009. `https://gmsh.info`

[4] Kitware Inc. *ParaView – Open-Source Scientific Visualisation.* `https://www.paraview.org`