A COMPLETE CFD AUTOMATION PIPELINE

End-to-End Scripted Simulation Management for Industrial CFD via the Autodesk CFD Python API

Pedro Henrique Rodrigues de Carvalho Castello

Politecnico di Milano · Milan, Italy · 2025

ABSTRACT

This paper presents a complete Python automation suite developed against the Autodesk CFD API that replaces the conventional manual simulation workflow with an end-to-end scripted pipeline. The suite comprises six modules: a Design Study Manager for scenario organisation and batch renaming; a Materials and Boundary Conditions Copilot for fluid property assignment and parametric flow condition sweeps; a Mesher and Convergence Copilot implementing the N-times-one-cycle adaptive meshing strategy; a Solver for autonomous iteration execution and convergence monitoring; a Data Gatherer for automated pressure extraction and CSV export; and a Logger providing stage-specific timestamped audit records of every state change across the pipeline. Together, these modules reduce analyst engagement time per simulation campaign by approximately ninety percent and produce a fully auditable and reproducible record from geometry setup through result extraction.

Keywords: *Autodesk CFD / Python automation / pipeline / adaptive meshing / boundary conditions / audit logging / K-ω SST*

MODULES

`DS-Manager`	Scenario organisation and batch renaming
`Materials & BCs`	Fluid, environment, parametric BCs
`Mesher`	N×1-cycle adaptive meshing, convergence
`Solver`	Autonomous execution, dialog handling
`Data-Gatherer`	Pressure extraction, CSV export
`Logger`	Stage-specific timestamped audit trail

CONTENTS

01 INTRODUCTION

The manual operation of a commercial CFD software package across a multi-scenario industrial campaign is a labour-intensive and error-prone process. For each scenario, the engineer must open the correct design, verify fluid material properties, configure boundary conditions, generate the mesh, launch the solver, monitor convergence, extract results, and record data. When a campaign spans eight to twenty-four scenarios — as is typical in industrial valve characterization for DN250 to DN400 class hardware — the cumulative analyst time becomes a significant fraction of the project schedule, and the probability of a parameter inconsistency rises with each manual operation.

This paper describes a complete Python automation suite developed against the Autodesk CFD API that eliminates each of these manual steps. The suite consists of six modules operating in sequence: DesignStudy-Manager, Materials and Boundary Conditions Copilot, Mesher and Convergence Copilot, Solver, Data-Gatherer, and Logger. Together they constitute a fully autonomous, reproducible, and auditable simulation pipeline from geometry organisation through result export.

The suite was developed over several months of active consulting work at an Italian valve manufacturer in the Rho district of Milan, where the need was to accelerate and de-risk CFD characterization of butterfly and shut-off valves under EN 12266-1 and ISO 5208 protocols. The performance gains documented here — approximately ninety percent reduction in analyst engagement time, near-zero parameter error rate, full reproducibility — were measured against the manual baseline in that environment. The methodology is general-purpose and applies to any Autodesk CFD design study structured as a scenario sweep.

02 PIPELINE ARCHITECTURE AND SHARED DESIGN PRINCIPLES

2.1 Motivation for a Multi-Module Architecture

A single monolithic script executing all pipeline stages in one pass would be simpler in structure but inflexible in practice. Different stages have different timing and re-run requirements: geometry organisation happens once per study; boundary conditions may need adjustment after initial solve inspection; meshing may be repeated with tightened parameters; and the solver is always the longest stage and must be independently restartable. A module-based architecture makes each stage individually addressable without side-effecting the others.

2.2 Shared Structural Conventions

All six modules share three structural conventions. The operator parameter block always occupies the top of the file, bounded by comment delimiters, and contains every value the engineer is expected to edit. All loops include explicit time.sleep() calls at API interaction points, compensating for GUI event queue latency. Every scenario loop iteration ends with study.save(), guaranteeing a valid save state even if the script is interrupted mid-campaign.

2.3 The Logger as Cross-Cutting Concern

The Logger is not a stage in the simulation sequence but a cross-cutting concern invoked inline at designated event points within each module. Rather than being imported as a library, it writes directly to an append-mode log file at the moment each event occurs. Section 8 describes the Logger in detail, including the stage-specific schemas that define exactly what is recorded at each pipeline point.

03 DESIGNSTUDY-MANAGER

3.1 Purpose

Before any simulation work begins, the design study must be organised into a consistently named scenario set. CAD imports typically arrive with auto-generated design names carrying no operational meaning. The DesignStudy-Manager script performs batch renaming across any subset of designs using a two-axis parameterisation: a design selection pattern (continuous or discrete) and a naming pattern (linear numerical, closed-suffix, or specific string list). All combinations are handled with explicit length and index validation before any mutation occurs.

3.2 Selection and Naming Patterns

The dptr variable accepts Continuous — iterating over all designs from dsns[0] to dsns[1] inclusive — or Discrete, applying the operation to an explicitly listed index set. The nptr variable accepts Linear_numerical, Closed (appends - Chiuso to the current name), or Specific (applies an ordered string list). The operator block for a discrete-specific campaign:

```
dptr = 'Discrete'
dsns = [1, 3, 5, 7]
nptr = 'Specific'
nrng = ['10deg', '30deg', '50deg',
'70deg']
```

Fig. 1 — Discrete-Specific mode: designs at indices 1, 3, 5, 7 renamed to opening-angle labels.

The script validates len(dsns) == len(nrng) before executing and checks all requested indices against the total design count. The discrete-specific rename loop:

```
j = 0
for i in dsns:
    design = design_list[i-1]
    print(f'Opened geometry at position
{i}')
    design.name = str(nrng[j])
    print(f'Position {i} renamed to
{nrng[j]}')
    j += 1
    time.sleep(0.5)
    study.save()
```

Fig. 2 — Discrete rename loop. Index j tracks position in nrng independently of design index i.

3.3 Logger Events — Manager Stage

The Logger records each renaming transaction with full before-and-after state: study name, timestamp, design index, old name, new name, and the pattern used. If the validation checks fail, an ERROR event is logged with the specific mismatch before any design is touched:

```
[2025-06-12 09:14:02] [INFO] [DS-Manager]
[Valve_DN300_Campaign_01]
  Design index : 1
  Old name     : Design 1
  New name     : 10deg
  Pattern      : Discrete / Specific
  Status       : OK
[2025-06-12 09:14:03] [INFO] [DS-Manager]
[Valve_DN300_Campaign_01]
  Design index : 3   |   Old name: Design 3
|   New name: 30deg
  Status       : OK
```

Fig. 3 — Logger output, DS-Manager stage. Old and new names are both recorded per transaction.

04 MATERIALS AND BOUNDARY CONDITIONS COPILOT

4.1 Fluid Material and Environment

The BC Copilot opens each scenario, assigns the fluid material, configures environmental reference conditions, and applies inlet, outlet, and symmetry boundary conditions. The operator block for a high-temperature air campaign:

```
fluid     = 'Air'
temp      = 383
tempUnit  = 'Celsius'
press     = 101325
pressUnit = 'Pa'
```

Fig. 4 — Fluid and environment parameter block.

These are applied via volume selection and the materialEnv interface:

```
scenario.selectionMode = 'Volume'
scenario.selectAll()
scenario.applyMaterial(fluid)
scenario.deselectAll()

env = scenario.materialEnv()
env.setProperty('temperature', temp)
env.setProperty('tempUnits',   tempUnit)
env.setProperty('pressure',    press)
env.setProperty('pressUnits',
pressUnit)
```

Fig. 5 — Material and environment assignment. selectAll assigns the fluid to the full flow domain volume.

4.2 Inlet Flow: Constant and Proportional Modes

In Constant mode, the effective flow is Max_Inlet_flow * Symmetry_coefficient for all scenarios. In Proportional mode the flow scales linearly across the opening range:

```
angular_step =
((Proportional_boundaries[1]
                -
Proportional_boundaries[0])
              / (num_scenarios - 1))

# Per scenario i in loop:
Inlet_effective_flow = (
    Max_Inlet_flow
    * ((Proportional_boundaries[0]
       + (i-1) * angular_step)
       / Proportional_boundaries[1])
    * Symmetry_coefficient
)
```

Fig. 6 — Proportional flow: linear interpolation across the opening range, scaled by the symmetry fraction.

The inlet BC object is constructed and applied to the designated surface:

```
inlet_bc       =
Setup.BoundaryCondition('Volume Flow
Rate')
inlet_bc.value = Inlet_effective_flow
inlet_bc.units = flowUnit
inlet_ents     = Setup.EntityIdList()
inlet_ents.append(Inlet_surface)
scenario.applyBoundaryCondition(
    inlet_bc, inlet_ents, ent_type)
```

Fig. 7 — Inlet BC application via EntityIdList.

4.3 Outlet and Symmetry BCs

The outlet is a zero gauge pressure condition. All symmetry surfaces are accumulated into one EntityIdList before a single applyBoundaryCondition call:

```
outlet_bc           =
Setup.BoundaryCondition('Pressure')
outlet_bc.value     = Outlet_pressure
outlet_bc.units     = pressureUnit
outlet_bc.pressureType = 'Gage'
outlet_ents         =
Setup.EntityIdList()
outlet_ents.append(Outlet_surface)
scenario.applyBoundaryCondition(
    outlet_bc, outlet_ents, ent_type)

symmetry_bc   =
Setup.BoundaryCondition('Slip/Symmetry')
sym_ents      = Setup.EntityIdList()
for surf_id in Symmetry_surfaces:
    sym_ents.append(surf_id)
scenario.applyBoundaryCondition(
    symmetry_bc, sym_ents, ent_type)
```

Fig. 8 — Outlet and symmetry BC application. All symmetry surface IDs load into one EntityIdList.

4.4 Logger Events — BC Stage

The BC Logger records every state transition in full: fluid old/new, temperature old/new, pressure reference old/new, then per-surface records for each BC including surface ID, mode, previous and new effective values with units. This gives a complete before-and-after record for every change:

```
[2025-06-12 09:22:11] [INFO] [BC-Copilot]
[Valve_DN300_Campaign_01]
  Scenario     : 17  (Design: 10deg)
  Fluid        : Air          (prev: Water)
  Temperature  : 383 Celsius  (prev: 25
Celsius)
  Pressure ref : 101325 Pa    (prev: 101325
Pa)
  --- Inlet BC ---
  Surface ID   : 32
  Mode         : Proportional
  Max flow     : 20332.31 m3/h  |  Sym.
coeff: 0.25
  Eff. flow    : 1270.77 m3/h   (prev: 0.0
m3/h)
  --- Outlet BC ---
```

```
  Surface ID   : 28  |  Type: Gage Pressure
  Value        : 0 Pa  (prev: 0 Pa)
  --- Symmetry BCs ---
  Surfaces     : [34, 25, 27, 31, 26, 30]
|  Status: OK
```

Fig. 9 — Logger output, BC stage. Every field records previous and new values, units, and surface IDs.

05 MESHER AND CONVERGENCE COPILOT

5.1 The N-Times-1-Cycle Strategy

The Mesher implements the adaptive meshing strategy described in detail in the companion paper. The explicit removeMesh() call before each mesh operation resets accumulated refinement state, enabling the adaptation engine to start from a clean, parameter-consistent baseline every cycle. The complete mesh generation block per scenario:

```
scenario.numberOfLayers =
Number_Of_Layers
scenario.layerFactor    = Layer_Factor
scenario.layerGradation =
Layer_Gradation

scenario.removeMesh()             #
explicit state reset
meshy    = scenario.mesh()
ad       = meshy.advancedControls()
ad.resolutionFactor        =
Resolution_Factor
ad.edgeGrowthRate          =
Edge_Growth_Rate
ad.minPointsOnEdge         =
Minimum_Points_On_Edge
ad.numPointsOnLongestEdge  =
Points_On_Longest_Edge
ad.surfLimitingAspectRatio =
Surface_Limiting_Aspect_Ratio
ad.volumeGrowthRate        =
Volume_Growth_Rate
ad.enableVolumeGrowthRate  = True
scenario.automaticSize()
```

Fig. 10 — Full mesh generation block. Wall layer parameters are applied before removeMesh(); advanced controls go to the fresh mesh object returned by scenario.mesh().

5.2 Convergence Criteria

Four convergence criteria are applied through the Intelligent Solution Control interface. The instantaneous slope, time-averaged slope, time-averaged concavity, and field fluctuation thresholds must all be satisfied simultaneously. Together they prevent both premature acceptance and false convergence from asymptotic approach:

```
scenario.intelligentSolutionControl =
'On'
scenario.instantaneousConvergenceCurveSl
ope
    =
Instantaneous_Convergence_Curve_Slope
scenario.timeAvgConvergenceCurveSlope
    =
Time_Averaged_Convergence_Curve_Slope
scenario.timeAvgConvergenceCurveConcavit
y
    =
Time_Averaged_Convergence_Curve_Concavit
y
scenario.fieldFluctuation =
Field_Fluctuations
scenario.turbModel = 'SST k-omega'
scenario.flow      = 'On'
```

Fig. 11 — Convergence and physics configuration. SST k-omega is set alongside the four-criterion convergence gate.

5.3 Logger Events — Mesher Stage

The Mesher Logger records the complete mesh parameter state before and after each operation: previous cell count (zero after removeMesh()), new cell count after automaticSize(), all parameter old/new values, and mesh generation time. This enables cell count delta tracking across the campaign:

```
[2025-06-12 09:35:44] [INFO] [Mesher]
[Valve_DN300_Campaign_01]
  Scenario       : 2  (Design: 10deg)
  removeMesh()   : OK  |  prev cell count:
847,312
  Wall layers    : 3   (prev: 3)
  Layer factor   : 0.45  (prev: 0.45)
  Edge growth    : 1.05  (prev: 1.05)
  Aspect ratio   : 20    (prev: 20)
  New cell count : 912,640  |  Mesh time:
00:02:31
  Convergence    : Inst 0.00001 / TimeAvg
0.0001 / Fluct 1e-7
  Turbulence     : SST k-omega  |  Status:
OK
```

Fig. 12 — Logger output, Mesher stage. Previous and new cell counts both recorded; parameter changes are flagged when they differ from prior scenario.

06 SOLVER

6.1 Autonomous Execution and Dialog Handling

The Solver re-applies convergence criteria before launching as a redundancy measure, then sets the iteration count. Autodesk CFD presents GUI dialog boxes at solver launch that cannot be dismissed through the API. The module uses pyautogui to send Enter keystrokes in the correct sequence. FAILSAFE is disabled because the solver runs are long and unattended, and cursor position cannot be controlled during execution:

```
study = Setup.DesignStudy.Create()
```

```
pyautogui.FAILSAFE = False

# Per scenario:
scenario.intelligentSolutionControl =
'On'
scenario.instantaneousConvergenceCurveSl
ope
    =
Instantaneous_Convergence_Curve_Slope
scenario.timeAvgConvergenceCurveSlope
    =
Time_Averaged_Convergence_Curve_Slope
scenario.fieldFluctuation =
Field_Fluctuations
scenario.turbModel = 'SST k-omega'
scenario.iterations =
number_of_iterations
time.sleep(1)
pyautogui.press('enter')  # confirm run
parameters
time.sleep(1)
scenario.run()
pyautogui.press('enter')  # acknowledge
result overwrite
time.sleep(1)
pyautogui.press('enter')  # confirm
solver start
```

Fig. 13 — Full solver launch sequence. pyautogui bridges the gap between the API and GUI dialogs the API cannot reach.

After each run, the module activates the results interface for the Data-Gatherer:

```
results         = scenario.results()
results.activate()
decision_center  = DC.DecisionCenter()
design_results   = DC.DesignResults()
study.save()
```

Fig. 14 — Results priming after solve. This ensures post-processing is available to the Data-Gatherer without manual GUI interaction.

6.2 Logger Events — Solver Stage

The Solver Logger captures the most granular time-series data in the pipeline: convergence criteria in effect at launch, solver start and end timestamps, iteration count at termination, stop reason, final residual, and total solve time. Non-convergent cases receive a WARN event:

```
[2025-06-12 09:38:17] [INFO] [Solver]
[Valve_DN300_Campaign_01]
  Scenario      : 2  (Design: 10deg)
  Iter. ceil.   : 5000  |  Inst slope:
0.0001  |  Fluct: 1e-6
  Solver start  : 09:38:17  |  Solver end:
09:52:44
  Iterations    : 1143  |  Stop:
Convergence criteria satisfied
  Final resid.  : 8.4e-6  |  Solve time:
00:14:27  |  Status: OK
```

```
[2025-06-12 11:04:33] [WARN] [Solver]
[Valve_DN300_Campaign_01]
  Scenario      : 5  (Design: 50deg-closed)
  Iterations    : 5000  |  Stop: Iteration
ceiling reached
  Final resid.  : 2.1e-4  |  Status: WARN -
review result manually
```

Fig. 15 — Logger output, Solver stage. WARN events flag non-convergent scenarios without aborting the campaign.

07 DATA-GATHERER

7.1 Wall Pressure Extraction

After all scenarios have been solved, the Data-Gatherer iterates over the scenario range and extracts wall-averaged static pressure at inlet and outlet surfaces. The deselect-select-calculate sequence is mandatory before each reading to prevent residual selection state from contaminating the result:

```
wall_results =
Results.WallResults(scenario)

wall_results.deselectAll()
wall_results.select(Inlet_wall)
wall_results.calculate()
Inlet_p_raw =
wall_results.pressure(Inlet_wall)
Inlet_p_Pa  = Inlet_p_raw / 10  #
dyne/cm2 -> Pa

wall_results.deselectAll()
wall_results.select(Outlet_wall)
wall_results.calculate()
Outlet_p_raw =
wall_results.pressure(Outlet_wall)
Outlet_p_Pa  = Outlet_p_raw / 10
```

Fig. 16 — Wall pressure extraction. The deselectAll / select / calculate cycle is repeated for each surface. Autodesk CFD internal units are dyne/cm²; dividing by 10 converts to Pa.

7.2 CSV Export

Results are accumulated in lists during the loop and written in a single pass at the end, preventing partial files if the script is interrupted. The study name drives the filename:

```
study_name   = study.name.replace(' ',
'_')
desktop_path = os.path.join(
    os.path.expanduser('~'), 'Desktop')
csv_filename = os.path.join(
    desktop_path,
    f'{study_name}_results.csv')

with open(csv_filename, 'w', newline='')
as f:
    writer = csv.writer(f)
```

```
    writer.writerow(['Iterations'] +
iterations_list)
    writer.writerow(['Inlet Pressure']
+ inlet_press_list)
    writer.writerow(['Outlet Pressure']
+ outlet_press_list)
```

Fig. 17 — CSV export. Single-pass write after loop completion; study name used as filename stem.

7.3 Logger Events — Data-Gatherer Stage

The Data-Gatherer Logger records both raw and converted pressure values, the computed differential pressure, the wall IDs used, and the output file path. Recording raw values alongside converted ones ensures any future unit system change can be detected retrospectively:

```
[2025-06-12 12:11:05] [INFO] [Data-
Gatherer] [Valve_DN300_Campaign_01]
  Scenario     : 9  (Design: 10deg)
  Inlet  wall  : 24  |  Outlet wall: 19
  Inlet  (raw) : 12847.3 dyne/cm2  ->
1284.73 Pa
  Outlet (raw) :  1024.1 dyne/cm2  ->
102.41 Pa
  Delta-P      : 1182.32 Pa
  Iterations   : 1143
  CSV output   :
~/Desktop/Valve_DN300_Campaign_01_results.c
sv
  Status       : OK
```

Fig. 18 — Logger output, Data-Gatherer stage. Raw and converted values both logged; differential pressure computed and recorded directly.

08 LOGGER

8.1 Architecture

The Logger writes to an append-mode file derived from the study name and date. Each module run opens a session header block containing the module name, study name, run date and time, and scenario range. This separates entries from different module runs within the same log file:

```
#
============================================
====================
# MODULE : Mesher & Convergence Copilot
# STUDY  : Valve_DN300_Campaign_01
# DATE   : 2025-06-12  |  TIME: 09:33:01
# RANGE  : Scenarios 2 to 5
#
============================================
====================
```

Fig. 19 — Log session header. Each module run creates a new block, making the file readable as a full campaign chronicle.

8.2 Event Classes and Stage-Specific Schemas

Events are classified into four severity levels: INFO for normal progression; WARN for non-fatal anomalies such as iteration ceiling; ERROR for incomplete scenario results; CRITICAL for pipeline aborts. Each module has a distinct schema defining which fields are recorded. The schemas differ substantively across stages:

→ DS-Manager: timestamp, study name, design index, old name, new name, pattern, status.

→ BC Copilot: timestamp, study name, scenario, design name, fluid old/new, temperature old/new, pressure ref old/new, per-surface records (ID, type, mode, old effective value, new effective value, units).

→ Mesher: timestamp, study name, scenario, design name, removeMesh result, previous cell count, all parameter old/new pairs, new cell count, mesh time, convergence criteria, turbulence model, status.

→ Solver: timestamp, study name, scenario, design name, iteration ceiling, convergence criteria at launch, start time, end time, iterations run, stop reason, final residual, solve time, status.

→ Data-Gatherer: timestamp, study name, scenario, design name, wall IDs, raw inlet/outlet pressures, converted pressures, differential pressure, iteration count, output CSV path, status.

8.3 Campaign Summary Block

At the end of each module run the Logger appends a summary block: total scenarios, event counts by severity, elapsed time, and a list of any scenario indices generating WARN or ERROR events:

```
#
============================================
====================
# SUMMARY : Solver
[Valve_DN300_Campaign_01]
# Date     : 2025-06-12
# Scenarios processed : 9
# INFO events   : 9  |  WARN: 2 (sc. 5,
8)  |  ERROR: 0
# Total solve time    : 02:17:44
#
============================================
====================
```

Fig. 20 — Campaign summary block. Two WARN events flag scenarios 5 and 8 for manual review.

09 DATA FLOW AND INTER-MODULE DEPENDENCIES

9.1 Sequential Dependency Chain

The five simulation modules form a strict sequential dependency chain: the DS-Manager must run first, then BC Copilot, then Mesher, then Solver, then Data-Gatherer. The Logger has no fixed position. Each module leaves the study in a saved state via study.save(), making every stage independently restartable from the last save point without re-running earlier modules.

Script	Inputs	Outputs
DS-Manager	Design indices, naming pattern	Renamed designs, saved study
Materials & BCs	Fluid props, surface IDs, flow	BC-configured scenarios
Mesher	Layer / growth / convergence	Meshed scenarios, physics set
Solver	Iteration count, convergence	Converged solutions, saved
Data-Gatherer	Scenario range, wall IDs	CSV on Desktop
Logger	All runtime events	Timestamped audit log file

Table 2 — Data flow: inputs consumed and outputs produced by each module.

10 COMPARATIVE ANALYSIS

The following table contrasts the full scripted pipeline against the conventional manual workflow across nine operational dimensions, measured against the pre-deployment baseline at the Rho facility.

Capability	Manual GUI	Script pipeline
Setup time	~45 min / scenario	*<3 min / campaign*
Param consistency	Human-dependent	*Script-enforced*
Convergence check	Visual inspection	*Criteria-driven, logged*
Result extraction	Manual copy/paste	*Auto CSV on Desktop*
Audit trail	None	*Full timestamped log*
Reproducibility	Operator-skill bound	*Version-controlled*
BC sweep	Manual recalculation	*Computed per scenario*
Error handling	Silent / undetected	*Logged and flagged*

Table 3 — Manual workflow versus full automation pipeline.

The audit trail entry reflects the most structurally significant difference. In the manual workflow, no record exists of what parameter values were actually applied to a given scenario. The Logger provides a complete before-and-after record for every mutable parameter in every scenario, making the simulation record independently verifiable and enabling mid-campaign error detection of the kind documented in Section 11.3.

11 INDUSTRIAL APPLICATION AND RESULTS

11.1 Campaign Structure

The suite was deployed for a leakage and flow coefficient characterization campaign covering three DN300 butterfly valve variants at nine opening positions each, yielding twenty-seven scenarios. Under the manual workflow this required approximately twenty hours of analyst time across two working days. Under the pipeline, total analyst engagement was approximately one hour: parameter block setup and final CSV review. The remaining nineteen hours ran unattended.

11.2 Convergence Performance

Across the twenty-seven scenarios, the Intelligent Solution Control criteria caused early termination in twenty-four cases. The three non-convergent cases — all at the most closed valve position — reached the five-thousand-iteration ceiling. The Logger recorded WARN events for all three. Post-processing confirmed pressure differentials had stabilised to within one percent of their asymptotic values; results were retained with a noted qualification.

11.3 Logger Utility in Practice

During the campaign the Logger identified one scenario where the BC Copilot had applied a symmetry coefficient of 0.5 instead of 0.25 due to a parameter block edit error made mid-campaign. The Logger recorded the previous effective flow alongside the new one, making the discrepancy immediately visible when the inlet pressure result appeared anomalously low. The scenario was re-run with the correct coefficient without re-running any other module or scenario. Without the Logger this error would have required a full campaign review.

12 EXTENSIBILITY AND FURTHER DEVELOPMENT

12.1 Additional Result Quantities

The Data-Gatherer is immediately extensible to mass flow rate (conservation verification), wall shear stress (erosion assessment), and temperature quantities (steam valve characterization). Each requires only additional WallResults API calls within the existing loop, with corresponding new fields in the Logger Data-Gatherer schema.

12.2 Parametric Mesh Sweeps

The Mesher parameter block can accept lists rather than scalars for any mesh control. A second outer loop over the list would produce a full mesh sensitivity sweep. The Logger Mesher schema would record which parameter set index is active for each scenario, making the sensitivity study output fully traceable from the log file alone.

12.3 Cross-Platform Transferability

The pipeline architecture — sequential addressable modules, operator parameter block, explicit state reset, shared study file, append-mode logger with stage-specific schemas — is a methodological template applicable to any CFD platform exposing a Python API. ANSYS Fluent via PyFluent and STAR-CCM+ via its Java macro layer are direct candidates, with different API calls but identical structural logic.

13 CONCLUSIONS

The manual operation of a commercial CFD software package across a multi-scenario industrial campaign is a labour-intensive and error-prone process. For each scenario, the engineer must open the correct design, verify fluid material properties, configure boundary conditions, generate the mesh, launch the solver, monitor convergence, extract results, and record data. When a campaign spans eight to twenty-four scenarios — as is typical in industrial valve characterization for DN250 to DN400 class hardware — the cumulative analyst time becomes a significant fraction of the project schedule, and the probability of a parameter inconsistency rises with each manual operation.

This paper describes a complete Python automation suite developed against the Autodesk CFD API that eliminates each of these manual steps. The suite consists of six modules operating in sequence: DesignStudy-Manager, Materials and Boundary Conditions Copilot, Mesher and Convergence Copilot, Solver, Data-Gatherer, and Logger. Together they constitute a fully autonomous, reproducible, and auditable simulation pipeline from geometry organisation through result export.

The suite was developed over several months of active consulting work at an Italian valve manufacturer in the Rho district of Milan, where the need was to accelerate and de-risk CFD characterization of butterfly and shut-off valves under EN 12266-1 and ISO 5208 protocols. The performance gains documented here — approximately ninety percent reduction in analyst engagement time, near-zero parameter error rate, full reproducibility — were measured against the manual baseline in that environment. The methodology is general-purpose and applies to any Autodesk CFD design study structured as a scenario sweep.

A complete end-to-end Python automation pipeline has been developed and deployed against the Autodesk CFD API, comprising six modules that together replace the conventional manual simulation workflow with a fully autonomous, reproducible, and auditable process from geometry organisation through result export.

The central technical contributions are: the N-times-1-cycle adaptive meshing decomposition, which transforms native meshing from an opaque atomic operation into a transparent, parameter-controllable sequence; the Proportional inlet flow mode, which eliminates manual flow recalculation across opening-position sweeps; the pyautogui-augmented Solver, which resolves the limitation that CFD solver launch dialogs cannot be dismissed through the API alone; and the stage-specific Logger, which records every mutable parameter state transition throughout the pipeline with before-and-after values, timestamps, study name, design index, and computed quantities — providing a complete forensic record of every simulation.

In the industrial application at the Rho valve manufacturer, the pipeline reduced analyst engagement time per twenty-seven-scenario campaign from approximately twenty hours to approximately one hour, confirmed parameter consistency across all scenarios, correctly extracted and exported results for all twenty-four convergent scenarios, flagged three non-convergent scenarios with WARN events, and identified one parameter error mid-campaign before it could propagate. The pipeline is presented not merely as an automation convenience but as a layer of engineering rigour applied systematically on top of a commercial tool — one that makes the full simulation record traceable, version-controlled, and independently verifiable.

Pedro Henrique Rodrigues de Carvalho Castello

Politecnico di Milano · Milan, Italy · 2025