- Building Simulation & Calibration UI
  - Project Overview
  - Main Features
    - Workflow and Interface Outline
  - Building Archetypes
  - Interface Outline
- 1. Database Usage
  - 1) All Possible filter_by Values and Which Key They Use
  - 2) Table of filter_by → Expected filter_criteria Format
  - 3) Example JSON Schema for the DB Filter
    - Example Payload
- 2. IDF Creation
  - 2.1 Geometry Building
    - 1) All Possible Combinations of Matching Fields
    - 2) Override Parameters and How They Are Assigned
      - (A) Overriding perimeter_depth
      - (B) Overriding has_core
    - 3) JSON Schema for Geometry Overrides
      - Example user_config override list
  - 2.2 Shading
    - 1. All Possible Combinations of Overrides (Shading)
    - 2. Values Allowed for Each Criterion (Shading)
    - 3. Clear JSON Schema for Possible Overrides (Shading)
    - 4. Example JSON Configuration (Shading)
- 3. Non-Geometry
  - 3.1 DHW
    - 1) All Possible Combinations of Matching Fields
    - 2) Possible param_name Values and How They Are Assigned
    - 3) JSON Schema for DHW Overrides
  - 3.2 Elec Lighting
    - 1) All Possible Combinations of Matching Fields
    - 2) Possible param_name Values & How They Are Assigned
    - 3) JSON Schema for Lighting Overrides
      - Example lighting.json Override List
  - 3.2 Elec lighting
    - 1) All Possible Combinations of Matching Fields
    - 2) Possible param_name Values & How They Are Assigned
    - 3) JSON Schema for Lighting Overrides
      - Example lighting.json Override List
  - 3.3 Elec_equipment
    - 2) Which param_name Values Can Be Overridden?
    - Putting it All Together
  - 3.3 Envelope Parameters (Fenez)
    - 1) All Possible Combinations of Matching Fields
    - 2) Possible param_name Values & How They Are Assigned
    - 3) JSON Schema for Fenestration Overrides
      - Example fenestration.json Override List
  - 3.4 Heating and Cooling (HVAC)
    - 1) All Possible Combinations of Matching Fields
    - 2) Possible param_name Values
    - 3) JSON Schema for HVAC Overrides
      - Sample JSON array of overrides
  - 3.5 Ventilation (HVAC)
    - 1) All Possible Combinations of Matching Fields
    - 2) Valid param_name Fields and Possible Overrides
      - Examples of fixed_value vs. min_val/max_val

    - 3) JSON Schema for Ventilation Overrides
        - Example user_config_vent Overrides
- 3.6 Attached Shading
    - 1) All Possible Combinations of Matching Fields
    - 2) Table of Recognized Shading Parameters for Override
    - 3) JSON Schema for Shading Overrides
        - Example user_config_shading in JSON
- 3.7 Ground Temperature
    - 1) All Possible Combinations of Matching Fields
        - 2) Possible param_name - Equivalent: The 12 Months
            - 3) JSON Schema for Ground - Temp Overrides
- 3.8 Sizing
    - 3. Clear JSON Schema for Possible Overrides
    - Example JSON Config
        - Example JSON Overrides
- 5. Output
    - 2. Table for Values Allowed for Each Criterion
    - 3. JSON Schema for Possible Overrides
        - Practical Example JSON Based on the Schema
- 6. Post-process
    - 1. All Possible Combinations for Overrides
    - 2. Allowed Values for Each Criterion
        - (Alternate Enumeration View)
    - 3. JSON Schema for Possible Overrides
        - Practical Example JSON Configuration Based on the Schema
- 7. Advanced Processings
    - 7.1 Scenario Development
    - 7.2 Validation
        - validation_base
        - validation_scenarios
        - Example snippet
    - 7.3 Sensitivity
    - 7.4 Surrogate
    - 7.5 Calibration
- 8. Main Config
- 9. Result Visualizations

Building Simulation & Calibration UI

Project Overview

https://github.com/amin-jalilzadeh-tu/E_Plus_2030_py

This repository automates the entire process of creating, running, and post-processing EnergyPlus simulations for one or more buildings. It combines geometry generation, HVAC/DHW/fenestration parameter assignment, advanced scenario or calibration methods, and a REST API (Flask/Gunicorn) for managing simulation jobs.

Main Features

Workflow and Interface Outline

1. **Main Data & Lookup Tables**

- We begin with the **main building data** (e.g., function, age, area, etc.) and **various lookup tables** (fenestration, HVAC, DHW, etc.).
- We also have **user configurations** that specify how to override or randomize parameters.
- **Partial overrides** (from `.json`) come next, overriding or setting ranges for specific buildings, scenarios, or parameters.
- The final picks (whether random or fixed) are logged in `assigned_fenez_params.csv` with the correct "range used" or "fixed value" notation, and then the final assigned and range values are saved in a CSV file.

2. **Create IDF File**

   - For each building (row in our main data), we **create an IDF** using the building data + lookup tables + user overrides.
   - During this process, we also **record the assigned values** (e.g., a specific WWR pick from a 0.25–0.35 range, or a DHW setpoint).

3. **Run Simulations**

   - After generating the IDF(s), we **run EnergyPlus simulations**.
   - The outputs (CSV) provide **simulated results** such as electricity consumption, heating demand, or DHW energy usage.

4. **Validation**

   - We compare the simulated results against **real data** (e.g., measured energy use).
   - A **validation report** is generated (showing MBE, CV(RMSE), pass/fail, etc.).

5. **Decision: Next Step?**

   - If the validation indicates acceptable accuracy, we may **stop**.
   - Otherwise, we **iterate** by generating *new sets of IDF files* with updated parameters (e.g., refined occupant densities, improved infiltration rates, etc.).

6. **Generate More IDF Files & Run Again**

   - In this "iterate" path, we create multiple scenarios based on the base IDF file and user config, using various value-assigning strategies to generate more IDF files.
   - Possibly based on post-calibration or updated values gleaned from the previous step.
   - We run **simulations** again, **record assigned values** for each scenario, and then **validate again**.

7. **Analysis (Sensitivity / Surrogate / Optimization)**

   - After a few calibration runs—or if we want to explore more systematically—we might:
     - **Perform a sensitivity analysis** (which parameters strongly affect the outputs?).
     - **Create/update a surrogate model** (a faster approximation of E+ results).
     - **Run an optimization** loop, using either the surrogate or direct E+ simulations to find the best parameter set.

8. **Reporting**

   - Throughout each **iteration** (pre-calibration, iteration 1, iteration 2, post-calibration, etc.), we keep a record of results.
   - Final outputs can include:
     - **Assigned parameter logs** (so you know exactly what was used).
     - **Simulation results** (EnergyPlus outputs).
     - **Validation reports** (MBE, CV(RMSE)).
     - Any **optimization** or **surrogate** analysis results.
   - These can also be sent via email, and you can visualize them. There will be possibilities to input your own report and visualize it without re-running the model.

9. **Update Post-Calibration Values**

  - Once satisfied with calibration or optimization outcomes, you can **update** your base assumptions or default dictionaries with the final "best fit" values.
  - This closes the loop, feeding new values back into **user configs** or **lookup tables**, improving future simulations.

> **P.S.** For validation and calibration, we currently don't have real data. The loop includes a placeholder in case you have data. You can provide your CSV file to calibrate the model for a specific archetype.

> Some options or objects are not active yet in this initial digital twin launch, so you may see them as deactivated. They will appear in the interface but do nothing at this stage. Some options will also be locked, yet still provide information, or may become editable later.

> We plan to create an interface for three types of users:
>
> - **Non-experts**: A minimal interface to avoid confusion (e.g., municipalities).
> - **Moderate**: An intermediate set of options.
> - **Expert developers**: Access to all options.

Building Archetypes

```
{
  "buildingFunctions": {
    "Residential": {
      "types": [
        "Corner House",
        "Apartment",
        "Terrace or Semi-detached House",
        "Detached House",
        "Two-and-a-half-story House"
      ]
    },
    "Non-Residential": {
      "types": [
        "Meeting Function",
        "Healthcare Function",
        "Sport Function",
        "Cell Function",
        "Retail Function",
        "Industrial Function",
        "Accommodation Function",
        "Office Function",
        "Education Function",
        "Other Use Function"
      ]
    }
  },
  "ageRanges": [
    "2015 and later",
    "2006 - 2014",
    "1992 - 2005",
    "1975 - 1991",
    "1965 - 1974",
    "1945 - 1964",
    "< 1945"
  ]
}
```

Interface Outline

- **Top-Level Choice**: Run a new simulation or view existing results.
  - If it's existing results, the user goes to a dashboard to visualize them.
  - If it's a new run, the user selects the **expertise level** (Expert, Moderate, or Non-expert), leading to different dashboards.
- **Selecting Buildings**:

    - First, select the area. You can either provide your own building info data (`df_buildings.csv`) or use the database to select buildings or a district from a map.
    - For the CSV file, we'll have a placeholder (we'll ask Tim how to provide it via API).
    - For the database, you can select buildings by ID, postcode, bounding box (lat/lon or x/y), or by drawing a polygon.

- **User Overrides**:

    - We'll display all possible matching-field combinations, which `param_name` values can be overridden, and a JSON schema for such overrides.
    - All matching fields (e.g., `building_id`, `dhw_key`) are optional, but `param_name` is required.
    - One of (`fixed_value` or `min_val`/`max_val`) must be provided.

- **Shading Example**:

    1. Match on (`building_id`, `shading_type_key`) if those exist in your override row.
    2. Apply any numeric ranges or string overrides to the default shading data in `shading_lookup`.
    3. Pick the final numeric value from any `_range` fields (midpoint, random, or min) based on the chosen strategy.
    4. Log the final picks in `assigned_shading_log[window_id]`.

1. Database Usage

1) All Possible `filter_by` Values and Which Key They Use

The function `load_buildings_from_db(filter_criteria, filter_by)` can take one of these `filter_by` strings:

1. **"meestvoorkomendepostcode"**Expects `filter_criteria["meestvoorkomendepostcode"]` to be a list of strings (for example, `["4816BL"]`).
2. **"pand_id"**Expects `filter_criteria["pand_id"]` to be a list (even if it's just one item), such as `["0383100000001369", "0383100000001370"]`.
3. **"pand_ids"**Similar to `"pand_id"`, but the code uses a separate field from `filter_criteria["pand_ids"]`.
4. **"bbox_xy"**Expects `filter_criteria["bbox_xy"] = [minx, miny, maxx, maxy]` (4 floats).
5. **"bbox_latlon"** Expects `filter_criteria["bbox_latlon"] = [min_lat, min_lon, max_lat, max_lon]`. Meaning: only buildings whose (lat, lon) are inside that bounding box.

---

2) Table of `filter_by` → Expected `filter_criteria` Format

`filter_by`	Field(s) in `filter_criteria`	Example Value(s)
`"meestvoorkomendepostcode"`	`"meestvoorkomendepostcode"` (list)	`["4816BL", "2012AB"]`
`"pand_id"`	`"pand_id"` (list)	`["0383100000001369", "0383100000001370"]`
`"pand_ids"`	`"pand_ids"` (list)	`["XYZ123", "XYZ999"]`
`"bbox_xy"`	`"bbox_xy"` = `[minx, miny, maxx, maxy]`	`[120000.0, 487000.0, 121000.0, 488000.0]`
`"bbox_latlon"`	`"bbox_latlon"` = `[min_lat, ..., ...]`	`[52.35, 4.85, 52.37, 4.92]`

3) Example JSON Schema for the DB Filter

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Database Filter JSON",
  "type": "object",
  "properties": {
    "use_database": {
      "type": "boolean",
      "description": "If false, the code skips the DB and uses user CSV input."
    },
    "filter_by": {
      "type": "string",
      "enum": ["meestvoorkomendepostcode", "pand_id", "pand_ids", "bbox_xy", "bbox_latlon"],
      "description": "Which key in db_filter to use for the WHERE clause."
    },
    "db_filter": {
      "type": "object",
      "properties": {
        "meestvoorkomendepostcode": {
          "type": "array",
          "items": { "type": "string" },
          "description": "List of one or more postcodes for a WHERE b.meestvoorkomendepostcode =
ANY(...) query."
        },
        "pand_id": {
          "type": "array",
          "items": { "type": "string" },
          "description": "List of one or more pand_id(s) => WHERE b.pand_id = ANY(...)"
        },
        "pand_ids": {
          "type": "array",
          "items": { "type": "string" },
          "description": "Alternative list of pand_id(s)."
        },
        "bbox_xy": {
          "type": "array",
          "items": { "type": "number" },
          "minItems": 4,
          "maxItems": 4,
          "description": "[minx, miny, maxx, maxy]."
        },
        "bbox_latlon": {
          "type": "array",
          "items": { "type": "number" },
          "minItems": 4,
          "maxItems": 4,
          "description": "[min_lat, min_lon, max_lat, max_lon]."
        }
      }
    }
  },
  "required": ["filter_by", "db_filter"],
  "additionalProperties": false
}
```

Example Payload

```
{
  "use_database": true,
  "filter_by": "bbox_latlon",
  "db_filter": {
    "meestvoorkomendepostcode": ["4816BL"],
    "pand_id": ["0383100000001369", "0383100000001370"],
    "pand_ids": ["XYZ123", "XYZ999"],
    "bbox_xy": [120000.0, 487000.0, 121000.0, 488000.0],
    "bbox_latlon": [52.35, 4.85, 52.37, 4.92]
  }
}
```

- In this example, we have all possible keys in `db_filter`, but since `"filter_by": "bbox_latlon"`, the code will only use the `[52.35, 4.85, 52.37, 4.92]` bounding box. The rest is effectively ignored by the query function.

- If `"use_database"` is false, the application skips calling `load_buildings_from_db()` entirely.

---

2. IDF Creation

Then we will have the IDF creation part, which includes a few main parts and sub-parts.

We will have geometry and non-geometry. For geometry, we have two or three sub-parts:

1. The geometry of buildings creation
2. The shadow of surrounding buildings
3. The shadow of surrounding trees

In IDF creation, we also need the following configuration, where the user defines the major parameters. The `num_workers` is the number of parallel processes and should be limited by the user, as it can use a lot of CPU and RAM.

```
"idf_creation": {
  "perform_idf_creation": true,
  "scenario": "scenario1",
  "calibration_stage": "pre_calibration",
  "strategy": "B",
  "random_seed": 42,
  "iddfile": "EnergyPlus/Energy+.idd",
  "idf_file_path": "EnergyPlus/Minimal.idf",
  "output_idf_dir": "output_IDFs",
  "run_simulations": true,
  "simulate_config": {
    "num_workers": 8,
    "ep_force_overwrite": true
  }
}
```

2.1 Geometry Building

1) All Possible Combinations of Matching Fields

For geometry overrides, each row in `user_config` can optionally include:

- `building_id`
- `building_type`

Combination #	`building_id` in row?	`building_type` in row?	Meaning in Plain Words
1	✗	✗	Universal: applies to all buildings, all building_types.
2	✓	✗	Applies only to a specific `building_id`, with no requirement on `building_type`.
3	✗	✓	Applies to any building with that exact `building_type`.
4	✓	✓	Most specific: applies only if both `building_id` AND `building_type` match.

`building_id` is one of the building IDs. For `building_type`, it can be, for example, an office or apartment. (Function and age range are not included here.)

> **Note**: Calibration stage and assigning value strategy will be present but deactivated as a placeholder.

---

2) Override Parameters and How They Are Assigned

The `param_name` can be either:

1. **"perimeter_depth"** (a numeric parameter)
2. **"has_core"** (a boolean parameter)

Each override row must include a way to set that parameter, either:

- `fixed_value` (which can be boolean or lock a numeric range to a single value), or
- `min_val` + `max_val` (for numeric range).

(A) Overriding `perimeter_depth`

- If you have `"param_name": "perimeter_depth"`, you can specify a numeric range by:

  ```
  "min_val": 2.0,
  "max_val": 3.0
  ```

  Then the code picks a final value from `[2.0, 3.0]`, depending on the strategy.

- If you want to lock it to a single value, do:

  ```
  "min_val": 2.5,
  "max_val": 2.5,
  "fixed_value": true
  ```

  The code interprets `(2.5, 2.5)` as a zero-width range, so you get exactly `2.5`.

(B) Overriding `has_core`

- If `"param_name": "has_core"`, the code looks for a `fixed_value` that should be boolean (true or false).

- Example:

  ```
  {
    "param_name": "has_core",
    "fixed_value": true
  }
  ```

  This sets `has_core_default = True`.

- There’s no concept of `min_val` / `max_val` for a boolean.

---

3) JSON Schema for Geometry Overrides

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Geometry Override Row",
  "type": "object",
  "properties": {
    "building_id": {
      "type": "integer",
      "description": "Optional exact building ID to match."
    },
    "building_type": {
      "type": "string",
      "description": "Optional exact building_type to match (e.g. 'Office Function')."
    },
    "param_name": {
      "type": "string",
```

```
      "enum": ["perimeter_depth", "has_core"],
      "description": "Which geometry parameter to override."
    },
    "fixed_value": {
      "description": "Used to force a single numeric value OR a boolean for has_core.",
      "anyOf": [
        { "type": "boolean" },
        { "type": "number" },
        { "type": "null" }
      ]
    },
    "min_val": {
      "type": "number",
      "description": "Used with max_val for numeric override of perimeter_depth."
    },
    "max_val": {
      "type": "number",
      "description": "Used with min_val for numeric override of perimeter_depth."
    }
  },
  "required": ["param_name"],
  "additionalProperties": false
}
```

Notes:

- Either you provide `fixed_value` (especially for a boolean `has_core`), or you provide `min_val` and `max_val` for a numeric override. If you do both, the code's logic sets (`min_val`, `min_val`) if `"fixed_value": true` is present, effectively locking the range.
- `building_id` and `building_type` are both optional. If you omit them, the override is universal.
- If `param_name = "has_core"`, the code expects `fixed_value` to be `true` or `false`.
- If `param_name = "perimeter_depth"`, you can do either:
  - `"fixed_value": true` plus `min_val`/`max_val` both the same, or
  - Just `min_val` and `max_val` without a `fixed_value`.

Example `user_config` override list

```
[
  {
    "param_name": "has_core",
    "fixed_value": false
  },
  {
    "building_id": 101,
    "param_name": "perimeter_depth",
    "min_val": 2.0,
    "max_val": 2.0,
    "fixed_value": true
  },
  {
    "building_type": "Apartment",
    "param_name": "perimeter_depth",
    "min_val": 1.5,
    "max_val": 2.0
  }
]
```

- Row 1: Universal override for `has_core = false` (applies to all buildings).
- Row 2: For `building_id = 101`, set `perimeter_depth` exactly = 2.0.
- Row 3: For any building whose `building_type` is `"Apartment"`, pick `perimeter_depth` in `[1.5, 2.0]`.

2.2 Shading

Here's a detailed, structured overview for configuring JSON-based user overrides for your shading post-processing logic (`postproc/merge_results.py`):

1. All Possible Combinations of Overrides (Shading)

The table below demonstrates possible combinations of shading parameters based on your provided logic:

#	Shading Type	Transmittance Schedule
1	SHADING:BUILDING:DETAILED ✓	✓ (schedule name provided)
2	SHADING:BUILDING:DETAILED	✗ (null)
3	SHADING:SITE:DETAILED	✓
4	SHADING:ZONE:DETAILED	✓
5	Custom	✓

2. Values Allowed for Each Criterion (Shading)

Criterion	Allowed Values	Example Values
`shading_type`	`"SHADING:BUILDING:DETAILED"`, `"SHADING:SITE:DETAILED"`, `"SHADING:ZONE:DETAILED"`, or a custom string	`"SHADING:BUILDING:DETAILED"`
`trans_schedule_name`	String or `null` (no schedule)	`"TreeTransSchedule"`, `null`

3. Clear JSON Schema for Possible Overrides (Shading)

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Shading Configuration Schema",
  "type": "object",
  "description": "Schema defining shading configuration overrides for EnergyPlus IDF objects",
  "properties": {
    "shading_type": {
      "type": "string",
      "description": "The EnergyPlus shading object type.",
      "enum": [
        "SHADING:BUILDING:DETAILED",
        "SHADING:ZONE:DETAILED",
        "SHADING:SITE:DETAILED"
      ]
    },
    "trans_schedule_name": {
      "type": ["string", "null"],
      "description": "Schedule controlling transmittance, or null for fully opaque surfaces."
    }
  },
  "required": ["shading_type"],
  "additionalProperties": false
}
```

4. Example JSON Configuration (Shading)

Below is an example JSON configuration that matches your schema:

```
{
  "shading_type": "SHADING:BUILDING:DETAILED",
  "trans_schedule_name": "TreeTransSchedule"
}
```

3. Non-Geometry

3.1 DHW

1) All Possible Combinations of Matching Fields

An override row in `user_config_dhw` can contain any subset of the following fields:

- `building_id`
- `dhw_key`
- `building_function`
- `age_range`

> **Note:** The calibration stage and the value-assigning strategy are present but deactivated as a placeholder.

Combination #	`building_id` in row?	`dhw_key` in row?	`building_function` in row?	`age_range` in row?	Meaning in Plain Words
1	✗	✗	✗	✗	Universal override: applies to all buildings, all DHW keys, all functions, all ages.
2	✓	✗	✗	✗	Applies only to a particular `building_id`; no restrictions on `dhw_key`, function, or age.
3	✗	✓	✗	✗	Applies to any building but only for a specific `dhw_key`.
4	✗	✗	✓	✗	Applies to any building but only if `building_function` matches (e.g., "residential").
5	✗	✗	✗	✓	Applies to any building but only if `age_range` matches (e.g., "2000 - 2010").
6	✓	✓	✗	✗	Applies only if both `building_id` and `dhw_key` match.
7	✓	✗	✓	✗	Applies only if `building_id` and `building_function` both match.
8	✓	✗	✗	✓	Applies only if `building_id` and `age_range` both match.
9	✗	✓	✓	✗	Applies only if `dhw_key` and `building_function` both match.

Combination #	`building_id` in row?	`dhw_key` in row?	`building_function` in row?	`age_range` in row?	Meaning in Plain Words
10	✗	✓	✗	✓	Applies only if `dhw_key` and `age_range` both match.
11	✗	✗	✓	✓	Applies only if `building_function` and `age_range` both match.
12	✓	✓	✓	✗	Applies if `building_id`, `dhw_key`, and `building_function` all match.
13	✓	✓	✗	✓	Applies if `building_id`, `dhw_key`, and `age_range` all match.
14	✓	✗	✓	✓	Applies if `building_id`, `building_function`, and `age_range` all match.
15	✗	✓	✓	✓	Applies if `dhw_key`, `building_function`, and `age_range` all match.
16	✓	✓	✓	✓	Most specific: must match that exact `building_id`, `dhw_key`, `building_function`, and `age_range`.

2) Possible `param_name` Values and How They Are Assigned

Each override row must have:

- `param_name`: the parameter we want to override.
- One of these assignment methods:
  - `fixed_value` => sets both min and max to that single value, **or**
  - `min_val` **and** `max_val` => sets a numeric range.

The valid `param_name` entries recognized by `assign_dhw_parameters()` are:

1. `occupant_density_m2_per_person`
2. `liters_per_person_per_day`
3. `default_tank_volume_liters`
4. `default_heater_capacity_w`
5. `setpoint_c`
6. `usage_split_factor`
7. `peak_hours`
8. `sched_morning`
9. `sched_peak`
10. `sched_afternoon`

11. `sched_evening`

Depending on which you select, you either provide:

- **Option A:** `"fixed_value": 52.0`→ The code treats it as a single-value range: (52.0, 52.0).
- **Option B:** `"min_val": 50.0`, `"max_val": 60.0` → The code picks either the midpoint (strategy "A"), a random number (strategy "B"), or the `min_val` (strategy "else").

For instance:

```
{
  "building_id": 4136730,
  "dhw_key": "Apartment",
  "param_name": "liters_per_person_per_day",
  "fixed_value": 52.0
}
```

This would force a single-value override (52 L/person/day) only for `building_id`=4136730 and `dhw_key`="Apartment".

Or:

```
{
  "building_function": "residential",
  "age_range": "1992 - 2005",
  "param_name": "setpoint_c",
  "min_val": 58.0,
  "max_val": 60.0
}
```

This would override `setpoint_c` in the range [58, 60] for any building whose `building_function` is "residential" (case-insensitive) and whose `age_range` is "1992 - 2005".

---

3) JSON Schema for DHW Overrides

```
{
  "title": "DHW Override Row",
  "type": "object",
  "properties": {
    "building_id": {
      "type": "integer",
      "description": "Optional exact building ID to match."
    },
    "dhw_key": {
      "type": "string",
      "description": "Optional exact DHW key to match. Example: 'Apartment', 'Office Function', etc."
    },
    "building_function": {
      "type": "string",
      "description": "Optional case-insensitive match. Example: 'residential' or 'non-residential'."
    },
    "age_range": {
      "type": "string",
      "description": "Optional exact string match. Example: '1992 - 2005'."
    },

    "param_name": {
      "type": "string",
      "enum": [
        "occupant_density_m2_per_person",
        "liters_per_person_per_day",
        "default_tank_volume_liters",
        "default_heater_capacity_w",
        "setpoint_c",
        "usage_split_factor",
        "peak_hours",
        "sched_morning",
        "sched_peak",
        "sched_afternoon",
        "sched_evening"
      ],
      "description": "Which DHW parameter you are overriding."
    },

    "fixed_value": {
      "type": "number",
      "description": "If given, sets min_val=max_val=this number."
    },
    "min_val": {
      "type": "number",
      "description": "Used in combination with max_val. If both are present, code picks from that
range."
    },
    "max_val": {
      "type": "number",
      "description": "Used in combination with min_val. If both are present, code picks from that
range."
    }
  },
  "required": ["param_name"],
  "oneOf": [
    {
      "required": ["fixed_value"]
    },
    {
      "required": ["min_val", "max_val"]
    }
  ],
  "additionalProperties": false
}
```

Example JSON

```
[
  {
    "param_name": "peak_hours",
    "fixed_value": 3.0
  },
  {
    "building_function": "Residential",
    "param_name": "liters_per_person_per_day",
    "min_val": 40.0,
    "max_val": 50.0
  },
  {
    "building_id": 4136730,
    "dhw_key": "Apartment",
    "param_name": "setpoint_c",
    "fixed_value": 58.0
  },
  {
    "building_function": "RESIDENTIAL",
    "age_range": "1992 - 2005",
    "param_name": "default_heater_capacity_w",
    "min_val": 3500,
    "max_val": 4500
  }
]
```

- **1st row:** Universal override for `peak_hours = 3.0` for all buildings, all keys.
- **2nd row:** Overrides `liters_per_person_per_day` range 40–50 for any building whose function is "residential" (case-insensitive).
- **3rd row:** Specific override for `building_id`=4136730 + `dhw_key`="Apartment", forcing `setpoint_c` to 58.
- **4th row:** For any "residential" + `age_range`="1992 - 2005", overrides the heater capacity range.

3.2 Elec Lighting

1) All Possible Combinations of Matching Fields

Your `find_applicable_overrides()` function (in `overrides_helper.py`) checks, for each override row:

- `building_id` (exact integer match if present),
- `building_type` (string match, case-insensitive),

- age_range (exact string match if present).

> **Note:** The calibration stage and the value-assigning strategy are present but deactivated as a placeholder.

Comb. #	building_id in row?	building_type in row?	age_range in row?	Meaning
1	✗	✗	✗	Universal override: applies to all buildings, all types, all ages.
2	✓	✗	✗	Applies only to a specific building_id.
3	✗	✓	✗	Applies to any building with that building_type.
4	✗	✗	✓	Applies to any building with that age_range.
5	✓	✓	✗	Must match both the building_id and building_type.
6	✓	✗	✓	Must match both the building_id and age_range.
7	✗	✓	✓	Must match both the building_type and age_range.
8	✓	✓	✓	Most specific override: must match building_id, building_type, and age_range.

2) Possible param_name Values & How They Are Assigned

Each override row must specify:

- param_name
- **Either** a fixed_value **or** both min_val and max_val.

The code (in assign_lighting_parameters()) recognizes the following param_name strings:

1. lights_wm2(Overriding LIGHTS_WM2_range in the lookup dict. Typically 0 W/m² for residential or ~15–30 W/m² for non-res.)
2. parasitic_wm2(Overriding PARASITIC_WM2_range, e.g., 0.28–0.30 W/m² for non-res.)
3. td(Overriding tD_range: typical daytime burning hours, e.g., 2100–2300.)
4. tn(Overriding tN_range: typical nighttime burning hours, e.g., 300–400.)
5. lights_fraction_radiant
6. lights_fraction_visible
7. lights_fraction_replaceable
8. equip_fraction_radiant
9. equip_fraction_lost

Your row can follow either example:

- **Fixed value example:**

```
{
  "building_id": 777,
  "param_name": "lights_wm2",
  "fixed_value": 18.0
}
```

This forces (min_val, max_val) = (18.0, 18.0) internally.

- **Range example:**

  ```
  {
    "building_type": "Office Function",
    "param_name": "td",
    "min_val": 2100,
    "max_val": 2200
  }
  ```

  This sets `tD_range = (2100, 2200)`. Depending on the strategy, the code picks the midpoint (strategy "A"), a random uniform value (strategy "B"), or the minimum (default) from that range.

---

3) JSON Schema for Lighting Overrides

Below is a possible JSON schema that enforces the above requirements. You can use this to validate your `lighting.json` or similar:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Lighting Override Row",
  "type": "object",
  "properties": {
    "building_id": {
      "type": "integer",
      "description": "Optional exact match for building ID."
    },
    "building_type": {
      "type": "string",
      "description": "Optional (case-insensitive) string like 'Residential' or 'Office Function'."
    },
    "age_range": {
      "type": "string",
      "description": "Optional exact match, e.g. '1992 - 2005'."
    },
    "param_name": {
      "type": "string",
      "enum": [
        "lights_wm2",
        "parasitic_wm2",
        "td",
        "tn",
        "lights_fraction_radiant",
        "lights_fraction_visible",
        "lights_fraction_replaceable",
        "equip_fraction_radiant",
        "equip_fraction_lost"
      ],
      "description": "Lighting parameter to override."
    },
    "fixed_value": {
      "type": "number",
      "description": "If present, overrides min_val/max_val with a single fixed value.",
      "nullable": true
    },
    "min_val": {
      "type": "number",
      "description": "Used with max_val if fixed_value not given.",
      "nullable": true
    },
    "max_val": {
      "type": "number",
      "description": "Used with min_val if fixed_value not given.",
      "nullable": true
    }
  },
  "required": ["param_name"],
  "oneOf": [
    { "required": ["fixed_value"] },
    { "required": ["min_val", "max_val"] }
  ],
  "additionalProperties": false
}
```

Example `lighting.json` Override List

Below is a sample array showing multiple rows:

```
[
  {
    "param_name": "lights_fraction_visible",
    "fixed_value": 0.25
  },
  {
    "building_type": "Healthcare Function",
    "param_name": "lights_wm2",
    "min_val": 18.5,
    "max_val": 20.0
  },
  {
    "building_id": 98765,
    "building_type": "Retail Function",
    "param_name": "tD",
    "fixed_value": 3000
  },
  {
    "age_range": "Pre-1980",
    "param_name": "equip_fraction_radiant",
    "min_val": 0.05,
    "max_val": 0.1
  }
]
```

- **1st row:** Universal override for `lights_fraction_visible = 0.25` (applies to all buildings if not more specifically matched by others).
- **2nd row:** Any building with `building_type`=="Healthcare Function" overrides `lights_wm2` range to (18.5–20.0).
- **3rd row:** Must match both `building_id`=98765 and `building_type`=="Retail Function", setting `tD` to a single fixed 3000.
- **4th row:** Any building with `age_range`=="Pre-1980" sets `equip_fraction_radiant` to be in [0.05, 0.1].

3.2 Elec lighting

---

1) All Possible Combinations of Matching Fields

Your `find_applicable_overrides()` function (in `overrides_helper.py`) checks, for each override row:

- building_id (exact integer match if present),
- building_type (string match, case-insensitive),
- age_range (exact string match if present).

Note: Here, calibration stage and assigning value strategy will be there, but deactivated as a placeholder.

Comb. #	building_id in row?	building_type in row?	age_range in row?	Meaning
1	✗	✗	✗	Universal override: applies to all buildings, all types, all ages.
2	✓	✗	✗	Applies only to a specific building_id.
3	✗	✓	✗	Applies to any building with that building_type.
4	✗	✗	✓	Applies to any building with that age_range.
5	✓	✓	✗	Must match both the building_id and building_type.
6	✓	✗	✓	Must match building_id and age_range.

Comb. #	building_id in row?	building_type in row?	age_range in row?	Meaning
7	✗	✓	✓	Must match building_type and age_range.
8	✓	✓	✓	Most specific override: must match building_id, building_type, and age_range.

2) Possible `param_name` Values & How They Are Assigned

Each override row must specify:

- `param_name`
- EITHER a `fixed_value` or both `min_val` and `max_val`.

The code (in `assign_lighting_parameters()`) recognizes the following `param_name` strings:

1. `lights_wm2`(Overriding `LIGHTS_WM2_range` in the lookup dict. Typically 0 W/m² for residential or ~15–30 W/m² for non-res.)
2. `parasitic_wm2`(Overriding `PARASITIC_WM2_range`, e.g., 0.28–0.30 W/m² for non-res.)
3. `td`(Overriding `tD_range`: typical daytime burning hours, e.g., 2100–2300.)
4. `tn`(Overriding `tN_range`: typical night-time burning hours, e.g., 300–400.)
5. `lights_fraction_radiant`
6. `lights_fraction_visible`
7. `lights_fraction_replaceable`
8. `equip_fraction_radiant`
9. `equip_fraction_lost`

Your row can look like either:

- **Fixed value example:**

  ```
  {
    "building_id": 777,
    "param_name": "lights_wm2",
    "fixed_value": 18.0
  }
  ```

  This forces `(min_val, max_val) = (18.0, 18.0)` behind the scenes.

- **Range example:**

  ```
  {
    "building_type": "Office Function",
    "param_name": "td",
    "min_val": 2100,
    "max_val": 2200
  }
  ```

  This sets `tD_range = (2100, 2200)`. Depending on the strategy, the code picks the midpoint (strategy "A"), a random uniform value (strategy "B"), or the minimum (default) from that range.

3) JSON Schema for Lighting Overrides

Below is a possible JSON schema that enforces the above requirements. You can use this to validate your `lighting.json` or similar:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Lighting Override Row",
```

```
  "type": "object",
  "properties": {
    "building_id": {
      "type": "integer",
      "description": "Optional exact match for building ID."
    },
    "building_type": {
      "type": "string",
      "description": "Optional (case-insensitive) string like 'Residential' or 'Office Function'."
    },
    "age_range": {
      "type": "string",
      "description": "Optional exact match, e.g. '1992 - 2005'."
    },

    "param_name": {
      "type": "string",
      "enum": [
        "lights_wm2",
        "parasitic_wm2",
        "td",
        "tn",
        "lights_fraction_radiant",
        "lights_fraction_visible",
        "lights_fraction_replaceable",
        "equip_fraction_radiant",
        "equip_fraction_lost"
      ],
      "description": "Lighting parameter to override."
    },

    "fixed_value": {
      "type": "number",
      "description": "If present, overrides min_val/max_val with a single fixed value.",
      "nullable": true
    },
    "min_val": {
      "type": "number",
      "description": "Used with max_val if fixed_value not given.",
      "nullable": true
    },
    "max_val": {
      "type": "number",
      "description": "Used with min_val if fixed_value not given.",
      "nullable": true
    }
  },
  "required": ["param_name"],
  "oneOf": [
    { "required": ["fixed_value"] },
    { "required": ["min_val", "max_val"] }
  ],
  "additionalProperties": false
}
```

Example `lighting.json` Override List

Here's a sample array showing multiple rows:

```
[
  {
    "param_name": "lights_fraction_visible",
    "fixed_value": 0.25
  },
  {
    "building_type": "Healthcare Function",
    "param_name": "lights_wm2",
    "min_val": 18.5,
    "max_val": 20.0
  },
  {
    "building_id": 98765,
    "building_type": "Retail Function",
    "param_name": "tD",
    "fixed_value": 3000
  },
  {
    "age_range": "Pre-1980",
    "param_name": "equip_fraction_radiant",
    "min_val": 0.05,
    "max_val": 0.1
    }
  ]
```

- **1st row**: universal override for `lights_fraction_visible = 0.25` (applies to all buildings if not more specifically matched by others).
- **2nd row**: any building with `building_type=="Healthcare Function"`, overrides `lights_wm2` range to `(18.5-20.0)`.
- **3rd row**: must match both `building_id=98765` and `building_type="Retail Function"`, sets `tD` to a single fixed `3000`.
- **4th row**: any building with `age_range="Pre-1980"`, sets `equip_fraction_radiant` to be in `[0.05, 0.1]`.

3.3 Elec_equipment

This one will be one of the objects that is deactivated for now, just as a placeholder. It is not active.

#	building_id in row?	building_type in row?	age_range in row?	Meaning in Plain Words
1	✗	✗	✗	Universal: applies to every building, all building_types, all age_ranges.
2	✓	✗	✗	Applies only to that specific building_id, no restriction on building_type or age_range.
3	✗	✓	✗	Applies to any building_id but only if building_type matches exactly.
4	✗	✗	✓	Applies to any building_id, building_type but only if age_range matches exactly.
5	✓	✓	✗	Must match both building_id and building_type.
6	✓	✗	✓	Must match building_id and age_range.
7	✗	✓	✓	Must match building_type and age_range.
8	✓	✓	✓	Most specific: must match that exact building_id, building_type, and age_range.

As soon as a row fails any present check (`building_id`, `building_type`, or `age_range`), it is skipped.

---

2) Which `param_name` Values Can Be Overridden?

From `assign_equipment_parameters()`, the code looks for these three `param_name` fields:

1. `equip_wm2` → overrides the range for `equip_rng`.
2. `tD` → overrides the range for `tD_rng`.
3. `tN` → overrides the range for `tN_rng`.

Putting it All Together

- **Combinations**: Any subset of `building_id`, `building_type`, `age_range` can be used to filter.
- **Param Names**: `equip_wm2`, `tD`, `tN`.
- **Values**: Code picks from `[min_val, max_val]` by strategy. (Set them equal for a single fixed value.)
- **JSON Structure**: `param_name`, `min_val`, `max_val` are mandatory; the others are optional filters.

That's how your `assign_equipment_parameters()` function will merge user-supplied override rows with the default `equip_lookup`.

3.3 Envelope Parameters (Fenez)

1) All Possible Combinations of Matching Fields

From the code in `fenez_config_manager.py` and `apply_user_fenez_overrides()`, we see that each override row might include:

- `building_function` (e.g. “residential” or “non_residential”)
- `building_type` (e.g. “Apartment” or “Office Function”)
- `age_range` (e.g. “1992 - 2005”)
- `scenario` (e.g. “scenario1”)
- `calibration_stage` (e.g. “pre_calibration” or “post_calibration”)
- `building_id` (optionally, if you want to target a specific building)

Comb. #	building_function?	building_id?	building_type?	age_range?	scenario?	calibration_stage?	Meaning
1	✗	✗	✗	✗	✗	✗	Universal types, age buildings.
2	✓	✗	✗	✗	✗	✗	Only for e.g.`build` ignoring t
3	✓	✗	✓	✗	✗	✗	Must matc
4	✗	✗	✓	✓	✓	✗	Must matc scenario.
5	✓	✗	✓	✓	✓	✓	Must matc + scenario
6	✓	✓	✓	✓	✓	✓	Most spec building_i

---

2) Possible `param_name` Values & How They Are Assigned

(# check)

---

3) JSON Schema for Fenestration Overrides

Below is a possible schema for `fenestration.json` overrides, modeling the logic from `apply_user_fenez_overrides()`:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Fenestration Override Row",
  "type": "object",
  "properties": {
    "building_id": {
      "type": "integer",
      "description": "Optional integer match. If present, must match the building's ID."
    },
    "building_function": {
      "type": "string",
      "description": "Optional string, typically 'residential' or 'non_residential' (case-insensitive)."
    },
    "building_type": {
      "type": "string",
      "description": "Optional sub-type. E.g. 'Apartment', 'Office Function', 'Corner House'."
    },
    "age_range": {
      "type": "string",
      "description": "Optional exact string match, e.g. '1992 - 2005'."
    },
    "scenario": {
      "type": "string",
      "description": "Optional scenario, e.g. 'scenario1'."
    },
    "calibration_stage": {
      "type": "string",
      "description": "Optional stage like 'pre_calibration' or 'post_calibration'."
    },

    "param_name": {
      "type": "string",
      "description": "What you're overriding: 'wwr', 'roof_r_value', 'wall_u_value', etc."
    },
    "fixed_value": {
      "type": "number",
      "description": "If present, sets min_val=max_val=fixed_value for that param.",
      "nullable": true
    },
    "min_val": {
      "type": "number",
      "description": "Used along with max_val if fixed_value is not given.",
      "nullable": true
    },
    "max_val": {
      "type": "number",
      "description": "Used along with min_val if fixed_value is not given.",
      "nullable": true
    }
  },
  "required": ["param_name"],
  "oneOf": [
    { "required": ["fixed_value"] },
    { "required": ["min_val", "max_val"] }
  ],
  "additionalProperties": false
}
```

Example `fenestration.json` Override List

Here is a short example that shows multiple different overrides:

```
[
  {
    "param_name": "wwr",
    "fixed_value": 0.32
    // Universal override for WWR => 0.32 for every building
    // that doesn't have a more specific match
  },
  {
    "building_function": "residential",
    "building_type": "Apartment",
    "age_range": "1985 - 1991",
    "scenario": "scenario1",
    "calibration_stage": "pre_calibration",
    "param_name": "wwr",
    "min_val": 0.25,
    "max_val": 0.30
    // Applies only if the building is residential + type=Apartment +
    // age_range=1985-1991 + scenario=scenario1 + stage=pre_calibration
  },
  {
    "building_id": 555,
    "building_function": "non_residential",
    "building_type": "Office Function",
    "param_name": "wall_u_value",
    "fixed_value": 1.2
    // Only for building_id=555, sets "exterior_wall" => U_value_range=(1.2,1.2).
  },
  {
    "building_function": "residential",
    "param_name": "roof_r_value",
    "min_val": 3.0,
    "max_val": 4.0
    // For any residential building, override "flat_roof" => R_value_range=(3.0,4.0).
  }
]
```

1. **Row 1**: Universal `wwr=0.32` if no more specific row applies.
2. **Row 2**: If the building is an “Apartment” in the year range 1985–1991, etc., override WWR to 0.25–0.30.

3. **Row 3**: If `building_id=555` is “Office Function,” fix the exterior wall U‐value to 1.2.
4. **Row 4**: For any “residential” building, override the roof R‐value range to 3.0–4.0.

3.4 Heating and Cooling (HVAC)

1) All Possible Combinations of Matching Fields

In `find_hvac_overrides()`, each row in `user_config` may optionally include:

- `building_id`
- `building_function`
- `residential_type`
- `non_residential_type`
- `age_range`
- `scenario`
- `calibration_stage`

#	building_id?	building_function?	residential_type?	non_residential_type?	age_range?	scenario?	calibrat
1	❌	❌	❌	❌	❌	❌	❌
2	✔️	❌	❌	❌	❌	❌	❌
3	❌	✔️	❌	❌	❌	❌	❌
4	❌	✔️	✔️	❌	❌	❌	❌
5	❌	❌	❌	❌	✔️	✔️	❌
6	❌	✔️	❌	✔️	❌	❌	❌
7	❌	❌	❌	❌	❌	❌	✔️
8	❌	✔️	❌	❌	✔️	✔️	✔️
…	etc.	etc.	etc.	etc.	etc.	etc.	etc.

If any field is missing, that field is not checked. If it is present, it must match exactly (string match for function/scenario, etc., integer match for `building_id`).

---

2) Possible `param_name` Values

From `assign_hvac_ideal_parameters()`, the recognized HVAC parameters are:

1. `heating_day_setpoint`
2. `heating_night_setpoint`
3. `cooling_day_setpoint`
4. `cooling_night_setpoint`
5. `max_heating_supply_air_temp`
6. `min_cooling_supply_air_temp`

When you provide an override row, you must specify `param_name` as one of these. Then you also supply either:

- `fixed_value`: a single numeric override (both min and max become that same value), or
- `min_val` and `max_val`: define a numeric range, from which the code picks the final number using a strategy (“A” => midpoint, “B” => random, otherwise => `min_val`).

Example:

```
{
  "building_function": "residential",
  "scenario": "scenario1",
  "param_name": "heating_day_setpoint",
  "min_val": 19.0,
  "max_val": 20.0
}
```

This means: for any building whose `building_function` is exactly `"residential"` and whose `scenario` is `"scenario1"`, override the default `heating_day_setpoint` range to (19.0–20.0).

---

3) JSON Schema for HVAC Overrides

A JSON schema for each override row can look like this:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "HVAC Override Row",
  "type": "object",
  "properties": {
    "building_id": {
      "type": "integer",
      "description": "Match exact building_id if present."
    },
    "building_function": {
      "type": "string",
      "description": "Match exact building_function if present. e.g. 'residential', 'non_residential'."
    },
    "residential_type": {
      "type": "string",
      "description": "Match exact if building_function='residential'."
    },
    "non_residential_type": {
      "type": "string",
      "description": "Match exact if building_function='non_residential'."
    },
    "age_range": {
      "type": "string",
      "description": "Match exact age_range if present. e.g. '1992 - 2005'."
    },
    "scenario": {
      "type": "string",
      "description": "Match exact scenario if present. e.g. 'scenario1'."
    },
    "calibration_stage": {
      "type": "string",
      "description": "Match exact calibration_stage if present. e.g. 'pre_calibration'."
    },

    "param_name": {
      "type": "string",
      "enum": [
        "heating_day_setpoint",
        "heating_night_setpoint",
        "cooling_day_setpoint",
        "cooling_night_setpoint",
        "max_heating_supply_air_temp",
        "min_cooling_supply_air_temp"
      ],
      "description": "Which HVAC parameter to override."
    },

    "fixed_value": {
      "type": "number",
      "description": "If given, sets both min and max to this one numeric value.",
      "nullable": true
    },
    "min_val": {
      "type": "number",
      "description": "Used with max_val if you want a range override instead of a fixed_value.",
      "nullable": true
    },
    "max_val": {
      "type": "number",
      "description": "Used with min_val if you want a range override instead of a fixed_value.",
      "nullable": true
    }
  },
  "required": ["param_name"],
  "oneOf": [
    { "required": ["fixed_value"] },
    { "required": ["min_val", "max_val"] }
  ],
  "additionalProperties": false
}
```

Interpretation:

- You can omit any of `building_id`, `building_function`, `residential_type`, etc. If you omit them all, the override is "universal."
- You must include exactly one method of specifying the override:
  - Either `"fixed_value": 21.0`
  - Or `"min_val": 20.0`, `"max_val": 22.0`.
- The code picks the final numeric setpoint from that range, depending on your strategy (midpoint, random, or min).

Sample JSON array of overrides

```
[
  {
    "scenario": "scenario1",
    "param_name": "heating_night_setpoint",
    "fixed_value": 15.0
  },
  {
    "building_function": "residential",
    "residential_type": "Apartment",
    "age_range": "1992 - 2005",
    "param_name": "heating_day_setpoint",
    "min_val": 19.5,
    "max_val": 20.5
  },
  {
    "building_id": 10125,
    "calibration_stage": "pre_calibration",
    "param_name": "max_heating_supply_air_temp",
    "fixed_value": 52.0
  }
]
```

- **1st row**: A universal override for `scenario="scenario1"` only, forcing `heating_night_setpoint = 15.0`.
- **2nd row**: Only applies if `building_function="residential"`, `residential_type="Apartment"`, and `age_range="1992 - 2005"`, adjusting the day heating setpoint range to (19.5–20.5).
- **3rd row**: A building - specific and stage - specific override forcing `max_heating_supply_air_temp=52.0` for `building_id=10125` in `pre_calibration`.

3.5 Ventilation (HVAC)

1) All Possible Combinations of Matching Fields

The function `find_vent_overrides()` in `assign_ventilation_values.py` checks any subset of the following fields in each override row:

- `building_id`
- `building_function`
- `age_range`
- `scenario`
- `calibration_stage`

#	building_id?	building_function?	age_range?	scenario?	calibration_stage?	Meaning
1	✗	✗	✗	✗	✗	Universal override: applies to all buildings, all functions, all ages, all scenarios, all stages.
2	✓	✗	✗	✗	✗	Applies only to a given `building_id`; no restriction on function, age, scenario, or stage.
3	✗	✓	✗	✗	✗	Applies to any building but only if the `building_function` matches (e.g., “residential”).
4	✗	✗	✓	✗	✗	Applies to any building but only if the `age_range` matches (e.g., “1975 - 1991”).
5	✗	✗	✗	✓	✗	Applies to any building but only if the `scenario` matches (e.g., “scenario1”).
6	✗	✗	✗	✗	✓	Applies to any building but only if the `calibration_stage` matches (e.g., “pre_calibration”).
7	✓	✓	✗	✗	✗	Must match both `building_id` and `building_function`.
8	✓	✗	✓	✗	✗	Must match `building_id` and `age_range`.

#	building_id?	building_function?	age_range?	scenario?	calibration_stage?	Meaning
9	✓	✗	✗	✓	✗	Must match `building_id` and `scenario`.
10	✓	✗	✗	✗	✓	Must match `building_id` and `calibration_stage`.
11	✗	✓	✓	✗	✗	Must match both `building_function` and `age_range`.
12	✗	✓	✗	✓	✗	Must match both `building_function` and `scenario`.
13	✗	✓	✗	✗	✓	Must match both `building_function` and `calibration_stage`.
14	✗	✗	✓	✓	✗	Must match both `age_range` and `scenario`.
15	✗	✗	✓	✗	✓	Must match both `age_range` and `calibration_stage`.
16	✗	✗	✗	✓	✓	Must match both `scenario` and `calibration_stage`.
17	✓	✓	✓	✗	✗	Must match `building_id`, `building_function`, and `age_range`.
18	✓	✓	✗	✓	✗	Must match `building_id`, `building_function`, and `scenario`.
19	✓	✓	✗	✗	✓	Must match `building_id`, `building_function`, and `calibration_stage`.
20	✓	✗	✓	✓	✗	Must match `building_id`, `age_range`, and `scenario`.
21	✓	✗	✓	✗	✓	Must match `building_id`, `age_range`, and `calibration_stage`.
22	✓	✗	✗	✓	✓	Must match `building_id`, `scenario`, and `calibration_stage`.

#	building_id?	building_function?	age_range?	scenario?	calibration_stage?	Meaning
23	✗	✓	✓	✓	✗	Must match `building_function`, `age_range`, and `scenario`.
24	✗	✓	✓	✗	✓	Must match `building_function`, `age_range`, and `calibration_stage`.
25	✗	✓	✗	✓	✓	Must match `building_function`, `scenario`, and `calibration_stage`.
26	✗	✗	✓	✓	✓	Must match `age_range`, `scenario`, and `calibration_stage`.
27	✓	✓	✓	✓	✗	Must match `building_id`, `building_function`, `age_range`, and `scenario`.
28	✓	✓	✓	✗	✓	Must match `building_id`, `building_function`, `age_range`, and `calibration_stage`.
29	✓	✓	✗	✓	✓	Must match `building_id`, `building_function`, `scenario`, and `calibration_stage`.
30	✓	✗	✓	✓	✓	Must match `building_id`, `age_range`, `scenario`, and `calibration_stage`.
31	✗	✓	✓	✓	✓	Must match `building_function`, `age_range`, `scenario`, and `calibration_stage`.
32	✓	✓	✓	✓	✓	Most specific: must match all fields (`building_id`, `function`, `age_range`, `scenario`, `stage`).

2) Valid `param_name` Fields and Possible Overrides

Within each override row, you specify `param_name` plus either:

- `fixed_value` (for a single exact value), or
- `min_val` and `max_val` (for a numeric range).

The code in `assign_ventilation_values.py` checks for these `param_name` keys and calls `override_range()` accordingly:

param_name	Type of Override	Meaning
`infiltration_base`	numeric, single/range	Adjusts the base infiltration at 10 Pa, e.g., (1.0, 1.2) or `fixed_value=1.1`.
`year_factor`	numeric, single/range	Multiplier for infiltration based on building age.
`system_type`	string or single value	Forces the final ventilation system choice to “A”, “B”, “C”, or “D”.
`fan_pressure`	numeric, single/range	Pressure rise (Pa) for mechanical fans.
`f_ctrl`	numeric, single/range	“control factor” for required ventilation flow (i.e., occupant control).
`hrv_eff`	numeric, single/range	Sensible heat recovery efficiency (0–1) if the system is “D” (balanced with HRV).
`infiltration_schedule_name`	string	Override the infiltration schedule (e.g., “AlwaysOnSched”, “VentSched_NightOnly”, etc.). Must be given as `fixed_value`. This schedule is predefined.
`ventilation_schedule_name`	string	Override the main ventilation schedule name. Must be given as `fixed_value`. This schedule is predefined.

Examples of `fixed_value` vs. `min_val/max_val`

1. Fixed numeric

   ```
   {
     "param_name": "fan_pressure",
     "fixed_value": 50.0
   }
   ```

   This sets `fan_pressure_range = (50, 50)`, which means the final pick is always `50`.

2. Range

   ```
   {
     "param_name": "infiltration_base",
     "min_val": 1.0,
     "max_val": 1.2
   }
   ```

   This sets `infiltration_base_range = (1.0, 1.2)`. The final pick can be the midpoint or random, etc., depending on the strategy.

3. String schedule

   ```
   {
     "param_name": "ventilation_schedule_name",
     "fixed_value": "WorkHoursSched"
   }
   ```

This directly sets `ventilation_schedule_name` to `"WorkHoursSched"`.

---

3) JSON Schema for Ventilation Overrides

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Ventilation Override Row",
  "type": "object",
  "properties": {
    "building_id": {
      "type": "integer",
      "description": "Optional building ID match."
    },
    "building_function": {
      "type": "string",
      "description": "Optional exact match, e.g. 'residential' or 'non_residential'."
    },
    "age_range": {
      "type": "string",
      "description": "Optional exact match, e.g. '1992 - 2005'."
    },
    "scenario": {
      "type": "string",
      "description": "Optional exact match, e.g. 'scenario1' or 'scenario2'."
    },
    "calibration_stage": {
      "type": "string",
      "description": "Optional exact match, e.g. 'pre_calibration' or 'post_calibration'."
    },
    "param_name": {
      "type": "string",
      "enum": [
        "infiltration_base",
        "year_factor",
        "system_type",
        "fan_pressure",
        "f_ctrl",
        "hrv_eff",
        "infiltration_schedule_name",
        "ventilation_schedule_name"
      ],
      "description": "Which ventilation parameter to override."
    },
    "fixed_value": {
      "description": "If present, sets min_val and max_val to this single value (numeric or string).",
      "type": ["number", "string", "null"]
    },
    "min_val": {
      "type": "number",
      "description": "Numeric lower bound, used if fixed_value not provided."
    },
    "max_val": {
      "type": "number",
      "description": "Numeric upper bound, used if fixed_value not provided."
    }
  },
  "required": ["param_name"],
  "oneOf": [
    { "required": ["fixed_value"] },
    { "required": ["min_val", "max_val"] }
  ],
  "additionalProperties": false
}
```

Example `user_config_vent` Overrides

Below is a sample array of ventilation overrides showing different matching rules:

```
[
  {
    "calibration_stage": "pre_calibration",
    "scenario": "scenario1",
    "param_name": "infiltration_base",
    "min_val": 1.2,
    "max_val": 1.5
  },
  {
    "building_function": "residential",
    "age_range": "1992 - 2005",
    "param_name": "system_type",
    "fixed_value": "D"
  },
  {
    "building_id": 111222,
    "scenario": "scenario1",
    "calibration_stage": "post_calibration",
    "param_name": "ventilation_schedule_name",
    "fixed_value": "WorkHoursSched"
  },
  {
    "param_name": "f_ctrl",
    "min_val": 0.75,
    "max_val": 0.80
  }
]
```

- **1st row:** applies to `pre_calibration` + `scenario1` only, overrides `infiltration_base` to a numeric range (1.2–1.5).
- **2nd row:** requires `building_function="residential"` + `age_range="1992 - 2005"`. It forces `system_type="D"`.
- **3rd row:** requires exactly `building_id=111222`, `scenario="scenario1"`, and `calibration_stage="post_calibration"`. Sets the ventilation schedule name to `"WorkHoursSched"`.
- **4th row:** universal (no building fields). It overrides `f_ctrl` everywhere to a range 0.75–0.80, unless a more specific row also matches.

3.6 Attached Shading

This is another object that is included as a deactivated option and serves only as a placeholder. The relevant code is not finalized yet.

1) All Possible Combinations of Matching Fields

In the shading overrides example, we typically have two fields to match on:

- `building_id`
- `shading_type_key`

Combination #	building_id in row?	shading_type_key in row?	Meaning
1	✗	✗	Universal override: applies to every building, every shading type.
2	✓	✗	Applies only to that specific `building_id` but for all shading types.
3	✗	✓	Applies to any building, but only for the specified `shading_type_key`.
4	✓	✓	Most specific override: must match that building and that shading type exactly.

A typical usage is #4 if you want "`building_id=123` and shading type `my_external_louvers`" to have a custom slat angle. Or you might omit `building_id` to override all louvered blinds with a new default angle.

---

2) Table of Recognized Shading Parameters for Override

From the sample code in `shading_lookup.py`, the typical parameters you can override (and how) are:

Possible Key	Typical Type	Meaning
blind_name	String (fixed)	The name that appears as the Blind object name in E+. You can override this if you want a custom name.
slat_orientation	String (fixed)	“Horizontal” or “Vertical” or similar.
slat_width_range	Numeric range or fixed	For example, (0.025, 0.03) (meters). The code picks a final numeric slat_width.
slat_separation_range	Numeric range or fixed	For example, (0.02, 0.02) => a single fixed 0.02 m.
slat_thickness_range	Numeric range or fixed	For example, (0.001, 0.001).
slat_angle_deg_range	Numeric range or fixed	For example, (30, 60). The code picks a final angle depending on its strategy.
slat_conductivity_range	Numeric range or fixed	Thermal conductivity of the slat material.
slat_beam_solar_transmittance_range	Numeric range or fixed	Beam - normal solar transmittance. Typically, 0 for opaque slats.
slat_beam_solar_reflectance_range	Numeric range or fixed	Reflectance for direct beam solar.
slat_diffuse_solar_transmittance_range	Numeric range or fixed	Diffuse - light transmittance. Usually, 0 for opaque blinds.
slat_diffuse_solar_reflectance_range	Numeric range or fixed	Diffuse reflectance.
slat_beam_visible_transmittance_range	Numeric range or fixed	Beam - normal visible (photopic) transmittance.
slat_beam_visible_reflectance_range	Numeric range or fixed	Beam - normal visible reflectance.
slat_diffuse_visible_transmittance_range	Numeric range or fixed	Diffuse visible transmittance.
slat_diffuse_visible_reflectance_range	Numeric range or fixed	Diffuse visible reflectance.
slat_ir_transmittance_range	Numeric range or fixed	IR transmittance, typically 0.

Possible Key	Typical Type	Meaning
slat_ir_emissivity_range	Numeric range or fixed	IR emissivity for longwave.
blind_to_glass_distance_range	Numeric range or fixed	If external louvers have some distance from the window.
blind_opening_multiplier_top	Numeric range or fixed	Coefficients to artificially scale top/bottom/left/right. Usually (1.0,1.0).
blind_opening_multiplier_bottom	Numeric range or fixed	
blind_opening_multiplier_left	Numeric range or fixed	
blind_opening_multiplier_right	Numeric range or fixed	
slat_angle_min_range	Numeric range or fixed	Lower bound of adjustable angle.
slat_angle_max_range	Numeric range or fixed	Upper bound of adjustable angle.

3) JSON Schema for Shading Overrides

Below is a suggested JSON schema for each override row if you store them in a list. This aligns with the “matching fields + param overrides” approach, letting you combine `building_id` (optional) with `shading_type_key` (optional). Each override row may contain any number of shading parameter overrides.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Shading Override",
  "type": "object",
  "properties": {
    "building_id": {
      "type": "string",
      "description": "Optional. If present, must match the building_id exactly."
    },
    "shading_type_key": {
      "type": "string",
      "description": "Optional. E.g. 'my_external_louvers' or 'my_vertical_fins'."
    },
    "blind_name": {
      "type": "string"
    },
    "slat_orientation": {
      "type": "string",
      "enum": ["Horizontal", "Vertical", "Other?"]
    },
    "slat_angle_deg_range": {
      "type": "array",
      "minItems": 2,
      "maxItems": 2,
      "items": { "type": "number" },
      "description": "Numeric range [min_val, max_val]. If you want a single fixed value, use [45,45]."
    },
    "slat_width_range": {
      "type": "array",
      "minItems": 2,
      "maxItems": 2,
      "items": { "type": "number" }
    }
    // ... etc. for each "xxx_range" you want to allow in JSON
  },
  "additionalProperties": true
}
```

Example `user_config_shading` in JSON

Here is a short example showing multiple rows in an array. Some rows have a `building_id`, while others do not:

```
[
  {
    "shading_type_key": "my_external_louvers",
    "slat_angle_deg_range": [30, 60],
    "slat_width_range": [0.025, 0.03]
  },
  {
    "building_id": "333888",
    "shading_type_key": "my_external_louvers",
    "blind_name": "Custom_Blinds_for_333888",
    "slat_beam_solar_reflectance_range": [0.25, 0.25]
  },
  {
    "slat_angle_deg_range": [10, 20]
  }
]
```

- **1st row:** no `building_id`, but `shading_type_key="my_external_louvers"`. Overwrites the angle range and width range for all buildings that use `my_external_louvers`.
- **2nd row:** for building `333888` and shading type `my_external_louvers`, override the name and the beam reflectance.
- **3rd row:** no `building_id`, no `shading_type_key`, so it is universal: sets a new default slat angle range (10–20) for any shading that does not get a more specific match.

3.7 Ground Temperature

This part is active but will be locked for now, so the user will only see the values. It will serve as a placeholder for future modifications.

1) All Possible Combinations of Matching Fields

Combination #	calibration_stage in row?	month in row?	Meaning in Plain Words
1	❌ (absent)	"January" (example)	Applies to January in all calibration stages.
2	✔ "pre_calibration"	"July"	Applies only to July in the “pre_calibration” stage.
3	❌ (absent)	“ALL_MONTHS” (if you choose)	If you allow a wildcard “ALL_MONTHS,” it applies to every month in all stages. (optional idea)

---

2) Possible `param_name` - Equivalent: The 12 Months

Your code expects a separate numeric range for each month in `groundtemp_lookup`. Hence, the “parameter name” in this domain is simply the month label:

- January

- February
- March
- April
- May
- June
- July
- August
- September
- October
- November
- December

For each monthly override, you can supply either:

- **fixed_value** => sets the month’s tuple to (value, value)
- **min_val + max_val** => sets the month’s tuple to (min_val, max_val)

---

3) JSON Schema for Ground‐Temp Overrides

3.8 Sizing

This part will remain locked. It will serve as a placeholder for changes, and options will be deactivated.

---

3. Clear JSON Schema for Possible Overrides

Here is a comprehensive JSON schema describing the structure for overriding:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "DHW Zone Sizing and Outdoor Air Configuration",
  "description": "Schema to override default building zone sizing and outdoor air specifications",
  "type": "object",
  "properties": {
    "building_function": {
      "type": "string",
      "enum": ["residential", "non_residential"],
      "description": "Specifies the building function used to select sizing parameters."
    },
    "calibration_stage": {
      "type": "string",
      "enum": ["pre_calibration", "post_calibration"],
      "description": "Calibration stage to select sizing parameters."
    },
    "strategy": {
      "type": "string",
      "enum": ["A", "B"],
      "description": "Strategy for parameter selection."
    },
    "random_seed": {
      "type": ["integer", "null"],
      "description": "Optional random seed for reproducibility of sizing parameters."
    },
    "zone_sizing_overrides": {
      "type": "object",
      "description": "Overrides for zone sizing parameters.",
      "properties": {
        "cooling_supply_air_temp": {
          "type": ["number", "null"]
        },
        "heating_supply_air_temp": {
          "type": ["number", "null"]
        },
        "cooling_supply_air_hr": {
          "type": ["number", "null"]
        },
        "heating_supply_air_hr": {
          "type": ["number", "null"]
        },
        "cooling_design_air_flow_method": {
          "type": ["string", "null"],
          "enum": ["DesignDay", "FlowPerArea", "Flow/Person", "none", null]
        },
        "heating_design_air_flow_method": {
          "type": ["string", "null"],
          "enum": ["DesignDay", "FlowPerArea", "Flow/Person", "none", null]
        }
      },
      "additionalProperties": false
    },
    "global_design_specifications": {
      "type": "object",
      "description": "Overrides for global outdoor air and air distribution specifications.",
      "properties": {
        "outdoor_air_flow_per_person": {
          "type": ["number", "null"]
        },
        "outdoor_air_flow_per_zone_floor_area": {
          "type": ["number", "null"]
        },
        "zone_air_distribution_effectiveness_cooling": {
          "type": ["number", "null"]
        },
        "zone_air_distribution_effectiveness_heating_mode": {
          "type": ["number", "null"]
        }
      }
    }
  },
  "required": ["building_function", "calibration_stage"],
  "additionalProperties": false
}
```

Example JSON Config

Here is a realistic JSON configuration example based on the provided logic and schema:

```
{
  "building_function": "non_residential",
  "calibration_stage": "pre_calibration",
  "strategy": "A",
  "random_seed": 42,
  "zone_sizing_overrides": {
    "cooling_supply_air_temp": 13.0,
    "heating_supply_air_temp": 44.0,
    "cooling_supply_air_hr": 0.0095,
    "heating_supply_air_hr": 0.0045,
    "cooling_design_air_flow_method": "FlowPerArea",
    "heating_design_air_flow_method": "FlowPerArea"
  },
  "zone_sizing_overrides": {
    "cooling_supply_air_temp": 13.5,
    "heating_supply_air_temp": 42.0
  },
  "global_design_spec_overrides": {
    "outdoor_air_flow_per_person": 0.00250,
    "outdoor_air_flow_per_zone_floor_area": 0.00035
  }
}









## 4. Climate and Weather File (EPW)

In the data file, we have defined and included some weather files with different climate change
scenarios for various years. For each region or location, we can have different latitude and longitude.
The code automatically assigns a building to the nearest available weather file. We have also included a
placeholder so that future users can provide additional files.

---
```

~~~
#### 1) All Possible Combinations of Matching Fields

The function `find_epw_overrides(building_id, desired_year, user_config_epw)` checks each row of `user_config_epw` for:

- **building_id** (exact match if present)
- **desired_year** (exact match if present)

| # | building_id in row? | desired_year in row? | Meaning in Plain Words                                                                                  |
|---|----------------------|----------------------|---------------------------------------------------------------------------------------------------------|
| 1 | ❌ (absent)          | ❌ (absent)          | Applies to all buildings and all years (universal override).                                            |
| 2 | ✔️ (present)         | ❌ (absent)          | Applies only to that `building_id`, any year.                                                           |
| 3 | ❌ (absent)          | ✔️ (present)         | Applies to any building, but only the specified `desired_year`.                                         |
| 4 | ✔️ (present)         | ✔️ (present)         | Must match that exact `building_id` and `desired_year`.                                                 |

---

#### 2) Table of Override Fields (param_name-equivalents)

Once a row matches, the code in `assign_epw_for_building_with_overrides()` looks for one or more of these keys and applies them in the following ways:

1. **fixed_epw_path**
   - If present, this forces that EPW path (skipping the usual nearest-lat/lon or year logic).
   - Example: `"fixed_epw_path": "C:/myweather/Amsterdam.epw"`

2. **override_year_to**
   - If present, this overrides the building's `desired_climate_year` with a new year before looking up an EPW.
   - Example: `"override_year_to": 2050`

3. **epw_lat** and **epw_lon**
   - If both are present, they override the building's latitude/longitude, then pick the nearest EPW (among those in `epw_lookup`) to that new lat/lon.
   - Example: `"epw_lat": 52.1, "epw_lon": 4.7`

You can supply some or all of these fields in a single row. For example, if a row only has `override_year_to`, we just override the year. If a row has all three (`fixed_epw_path`, `override_year_to`, and lat/lon overrides), then we apply them in this order:

1. `forced_epw = fixed_epw_path` => No further logic is needed, because if `forced_epw` is set, the code stops and doesn't pick from the lookup.
2. If no `fixed_epw_path` is given, we apply the lat/lon override and the new year, then do a normal "closest EPW" search.

| Override Field     | Type    | How It's Used                                                                                 |
|-------------------|---------|-----------------------------------------------------------------------------------------------|
| `fixed_epw_path`   | string  | Forces a specific EPW path. If set, we skip normal lat/year search.                           |
| `override_year_to`| integer | Replaces the building's `desired_climate_year`.                                               |
| `epw_lat`          | float   | Overrides building's latitude (must be used with `epw_lon`).                                  |
| `epw_lon`          | float   | Overrides building's longitude (must be used with `epw_lat`).                                 |

---

#### 3) JSON Schema for EPW Overrides

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "EPW Override Row",
  "type": "object",
  "properties": {
    "building_id": {
      "type": "integer",
      "description": "Optional exact building ID to match."
    },
    "desired_year": {
      "type": "integer",
      "description": "Optional exact desired_climate_year to match."
    },

    "fixed_epw_path": {
      "type": "string",
~~~

```
      "description": "If given, forces this exact EPW path (skips lat/long or year logic)."
    },
    "override_year_to": {
      "type": "integer",
      "description": "If given, overrides the building's desired_climate_year with this new year."
    },
    "epw_lat": {
      "type": "number",
      "description": "If given with epw_lon, override the building's lat/lon used for epw_lookup."
    },
    "epw_lon": {
      "type": "number",
      "description": "If given with epw_lat, override the building's lat/lon used for epw_lookup."
    }
  },
  "additionalProperties": false,
  "anyOf": [
    { "required": ["fixed_epw_path"] },
    { "required": ["override_year_to"] },
    { "required": ["epw_lat", "epw_lon"] }
  ]
}
```

- `building_id` (optional) => If present, the row only applies to that building.
- `desired_year` (optional) => If present, the row only applies to that year.
- **One or more** of [`fixed_epw_path`, `override_year_to`, `epw_lat`, `epw_lon`] must be present for the override to have any effect.

Example JSON Overrides

A typical overrides list (`user_config_epw`) might look like this:

```
[
  {
    "override_year_to": 2050
  },
  {
    "building_id": 101,
    "fixed_epw_path": "C:/EPWs/Rotterdam2030.epw"
  },
  {
    "desired_year": 2020,
    "epw_lat": 52.0,
    "epw_lon": 4.5
  },
  {
    "building_id": 202,
    "desired_year": 2050,
    "epw_lat": 53.0,
    "epw_lon": 5.5
  }
]
```

- **1st row:** Universal override => For any building/year, replace the year with 2050 (unless a more specific row also matches).
- **2nd row:** If `building_id` = 101, force that building to use `"C:/EPWs/Rotterdam2030.epw"` for all years.
- **3rd row:** If a building's `desired_year` is 2020, override its lat/lon to (52.0, 4.5) for EPW selection.
- **4th row:** If `building_id` = 202 and `desired_year` = 2050, override lat/lon to (53.0, 5.5).

5. Output

When users want to run simulations, they have the option to define how the simulation outputs are generated, from which elements to simulate to the scale of resolution. As mentioned at the end, we have default options. However, note that for the desired variables and meters, multiple selections can be made rather than just one. For frequency, it should be one of the allowed values.

#	Desired Variables	Desired Meters	Variable Frequency	Meter Frequency	Include Tables	Include Summary
1	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	✗
3	✓	✓	✓	✗	✓	✓
4	✓	✓	✗	✓	✓	✓
5	✓	✓	✗	✗	✓	✓
6	✓	✗	✓	-	✓	✓
7	✗	✓	-	✓	✓	✓
8	✗	✗	-	-	✓	✓
9	✓	✗	✗	-	✗	✗
10	✗	✓	-	✗	✗	✗
11	✗	✗	-	-	✗	✓

2. Table for Values Allowed for Each Criterion

Criterion	Allowed Values	Example Values
desired_variables	List of available variable names	`"Facility Total Electric Demand Power"`, `"Zone Air Temperature Maximum"`
desired_meters	List of available meter key names	`"Electricity:Facility"`, `"Fans:Electricity"`, `"Electricity:*"`
override_variable_frequency	`"Timestep"`, `"Hourly"`, `"Daily"`, `"Monthly"`, `"Annual"`, `null` (use default)	`"Hourly"`, `null`
override_meter_frequency	`"Timestep"`, `"Hourly"`, `"Daily"`, `"Monthly"`, `"Annual"`, `null` (use default)	`"Daily"`, `null`
include_tables	Boolean (`true`, `false`)	`true`, `false`
include_summary	Boolean (`true`, `false`)	`true`, `false`

3. JSON Schema for Possible Overrides

Here's a clear JSON schema describing the structure of your override configurations:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Output Definitions Override Schema",
  "description": "Schema for overriding DHW output definitions for EnergyPlus",
  "type": "object",
  "properties": {
    "desired_variables": {
      "type": ["array", "null"],
      "description": "List of variable names to include; if null, use all default variables.",
      "items": {
        "type": "string",
        "enum": [
          "Facility Total Electric Demand Power",
          "Facility Total Gas Demand Power",
          "Zone Air Temperature Maximum"
          // add other variables here
        ]
      }
    },
    "desired_meters": {
      "type": ["array", "null"],
      "description": "List of meter key names to include; if null, use all default meters.",
      "items": {
        "type": "string",
        "enum": [
          "Fans:Electricity",
          "Electricity:Facility",
          "Electricity:*"
        ]
      }
    },
    "override_variable_frequency": {
      "type": ["string", "null"],
      "description": "Overrides default variable reporting frequency.",
      "enum": ["Timestep", "Hourly", "Daily", "Monthly", null]
    },
    "override_meter_frequency": {
      "type": ["string", "null"],
      "description": "Overrides default meter reporting frequency.",
      "enum": ["Timestep", "Hourly", "Daily", "Monthly", null]
    },
    "include_tables": {
      "type": "boolean",
      "description": "Determines if monthly or annual tables should be included."
    },
    "include_summary": {
      "type": "boolean",
      "description": "Determines if summary reports should be included."
    }
  }
}
```

Practical Example JSON Based on the Schema

```
{
  "desired_variables": [
    "Facility Total Electric Demand Power",
    "Zone Air Temperature Maximum"
  ],
  "desired_meters": ["Electricity:Facility"],
  "override_variable_frequency": "Hourly",
  "override_meter_frequency": "Hourly",
  "include_tables": true,
  "include_summary": true
}
```

6. Post-process

Here's a structured overview of your JSON-based user configuration for the `postproc/merge_results.py` script, including detailed tables and schema:

1. All Possible Combinations for Overrides

The following table outlines all logical combinations of overrides for your post-processing script:

#	convert_to_daily	convert_to_monthly	daily_aggregator	monthly_aggregator
1	✓	✓	✓	✓
2	✓	✗	✓	-
3	✗	✓	-	✓
4	✗	✗	-	-

- ✓ indicates an explicitly enabled override.
- ✗ indicates an explicitly disabled override.
- `-` means irrelevant (e.g., an aggregator for daily is irrelevant when daily conversion is false).

2. Allowed Values for Each Criterion

Criterion	Allowed Values	Examples
convert_to_daily	Boolean (`true`, `false`)	`true`, `false`
convert_to_monthly	Boolean (`true`, `false`)	`true`, `false`
daily_aggregator	`"mean"`, `"sum"`, `"max"`, `"min"`, `"pick_first_hour"`, `"none"`	`"mean"`, `"sum"`, `"none"`
monthly_aggregator	`"mean"`, `"sum"`, `"max"`, `"min"`, `"pick_first_hour"`	`"mean"`, `"sum"`

(Alternate Enumeration View)

Criterion	Allowed Values (enumeration)	Example
convert_to_daily	Boolean	`true`, `false`
daily_aggregator	`"mean"`, `"sum"`, `"max"`, `"min"`, `"pick_first_hour"`	`"mean"`, `"sum"`
convert_to_monthly	Boolean	`true`, `false`
monthly_aggregator	`"mean"`, `"sum"`, `"max"`, `"min"`, `"pick_first_hour"`	`"mean"`, `"sum"`

3. JSON Schema for Possible Overrides

Below is a clear and structured JSON Schema that defines your configuration:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Post-processing Configuration Schema",
  "type": "object",
  "properties": {
    "post_process": {
      "type": "boolean",
      "description": "Flag to enable or disable post-processing"
    },
    "post_process_config": {
      "type": "object",
      "properties": {
        "base_output_dir": {
          "type": "string",
          "description": "Base directory containing output CSV files."
        },
        "outputs": {
          "type": "array",
          "description": "List of configurations for merged outputs.",
          "items": {
            "type": "object",
            "properties": {
              "convert_to_daily": {
                "type": "boolean",
                "description": "Aggregate hourly data to daily."
              },
              "daily_aggregator": {
                "type": "string",
                "enum": ["mean", "sum", "max", "min", "pick_first_hour", "none"],
                "description": "Method to aggregate hourly data to daily."
              },
              "convert_to_monthly": {
                "type": "boolean",
                "description": "Aggregate daily/hourly data to monthly."
              },
              "monthly_aggregator": {
                "type": ["string", "null"],
                "enum": ["mean", "sum", "max", "min", "pick_first_hour", "none", null],
                "description": "Method to aggregate daily data to monthly."
              },
              "output_csv": {
                "type": "string",
                "description": "Path for the merged output CSV."
              }
            },
            "required": ["convert_to_daily", "convert_to_monthly", "aggregator", "output_csv"],
            "additionalProperties": false
          }
        }
      },
      "required": ["base_output_dir", "outputs"],
      "additionalProperties": false
    }
  },
  "required": ["base_output_dir", "outputs"]
}
```

Practical Example JSON Configuration Based on the Schema

```
"post_process": true,
"post_process_config": {
  "base_output_dir": "Sim_Results",
  "outputs": [
    {
      "convert_to_daily": false,
      "convert_to_monthly": false,
      "aggregator": "none",
      "output_csv": "output/results/merged_as_is.csv"
    },
    {
      "convert_to_daily": true,
      "convert_to_monthly": false,
      "aggregator": "mean",
      "output_csv": "output/results/merged_daily_mean.csv"
    }
  ]
}
```

7. Advanced Processings

7.1 Scenario Development

This section is the advanced part. The paths remain unchanged but should be visible to the user. The first part involves modifying or generating more scenarios for a specific building. In the previous steps, we created IDF files based on the user's selection and configuration. Here, the user can specify the ID of one building, which must match the selected area.

```
"building_id": 4136730
```

`num_scenarios` specifies the number of IDF files that will be generated. We do this because we use the results for surrogate modeling, sensitivity analysis, calibration, etc.

```
"num_scenarios": 5
```

For each new IDF, we assign new values. The strategies can be one of the following:

- **method**:
  - `"random_uniform"` => uniform in `[param_min, param_max]`
  - `"scale_around_base"` => `base_val * random(1 - scale_factor, 1 + scale_factor)`
  - `"offset_half"` => `base_val +/- up to 50% of half the total range`
- **scale_factor**: Used if `method="scale_around_base"`

```
"picking_method": "random_uniform",
"picking_scale_factor": 0.5
```

`run_simulations` must be `true`; otherwise, it will only generate IDF files and not run simulations:

```
"run_simulations": true
```

`num_workers` sets the number of parallel workers:

```
"num_workers": 8
```

We will then have post-processing again. In the JSON, it should always be set to `true` for now, just like in the IDF creation part.

```
"modification": {
  "perform_modification": true,
  "modify_config": {
    "idd_path": "EnergyPlus/Energy+.idd",
    "assigned_csv": {
      "hvac_building": "assigned/assigned_hvac_building.csv",
      "hvac_zones": "assigned/assigned_hvac_zones.csv",
      "dhw": "assigned/assigned_dhw_params.csv",
      "vent_build": "assigned/assigned_vent_building.csv",
      "vent_zones": "assigned/assigned_vent_zones.csv",
      "elec": "assigned/assigned_lighting.csv",
      "fenez": "assigned/structured_fenez_params.csv"
    },
    "scenario_csv": {
      "hvac": "scenarios/scenario_params_hvac.csv",
      "dhw": "scenarios/scenario_params_dhw.csv",
      "vent": "scenarios/scenario_params_vent.csv",
      "elec": "scenarios/scenario_params_elec.csv",
      "fenez": "scenarios/scenario_params_fenez.csv"
    },
    "output_idf_dir": "scenario_idfs",
    "building_id": 4136730,
    "num_scenarios": 5,
    "picking_method": "random_uniform",
    "picking_scale_factor": 0.5,
    "run_simulations": true,
    "simulation_config": {
      "num_workers": 4,
      "output_dir": "Sim_Results/Scenarios"
    },
    "perform_post_process": true,
    "post_process_config": {
      "output_csv_as_is": "results_scenarioes/merged_as_is_scenarios.csv",
      "output_csv_daily_mean": "results_scenarioes/merged_daily_mean_scenarios.csv"
    }
  }
}
```

7.2 Validation

The validation part will have two main steps:

1. Validation of IDF creation results
2. Validation of IDF simulation results

However, in JSON, we see three different configurations. For now, let the validation configuration be in the JSON as is. `validation_config` should always be `false` for now and locked, but we still keep it. It also specifies which columns and files will be compared.

```
"perform_validation": false,
"validation_config": {
```

validation_base

```
"validation_base": {
  "perform_validation": true,
  "config": {
    "real_data_csv": "data/mock_merged_daily_mean.csv",
    "sim_data_csv": "results/merged_daily_mean.csv",
    "bldg_ranges": {
      "0": [
        0,
        1,
        2,
        3
      ]
    },
    "variables_to_compare": [
      "Electricity:Facility [J](Hourly)",
      "Heating:EnergyTransfer [J](Hourly)",
      "Cooling:EnergyTransfer [J](Hourly)"
    ],
    "threshold_cv_rmse": 30.0,
    "skip_plots": true,
    "output_csv": "validation_report_base.csv"
  }
}
```

validation_scenarios

```
"validation_scenarios": {
  "perform_validation": true,
  "config": {
    "real_data_csv": "data/mock_merged_daily_mean.csv",
    "sim_data_csv": "results_scenarioes/merged_daily_mean_scenarios.csv",
    "bldg_ranges": {
      "0": [
        0,
        1,
        2
      ]
    },
    "variables_to_compare": [
      "Electricity:Facility [J](Hourly)",
      "Heating:EnergyTransfer [J](Hourly)",
      "Cooling:EnergyTransfer [J](Hourly)"
    ],
    "threshold_cv_rmse": 30.0,
    "skip_plots": true,
    "output_csv": "validation_report_scenarios.csv"
  }
}
```

Example snippet

```
"perform_validation": false,
"validation_config": {
  "real_data_csv": "data/mock_merged_daily_mean.csv",
  "sim_data_csv": "results/merged_daily_mean.csv",
  "bldg_ranges": {
    "0": [
      0,
      1,
      2
    ]
  },
  "variables_to_compare": [
    "Electricity:Facility [J](Hourly)",
    "Heating:EnergyTransfer [J](Hourly)",
    "Cooling:EnergyTransfer [J](Hourly)"
  ],
  "threshold_cv_rmse": 30.0,
  "skip_plots": true,
  "output_csv": "scenario_validation_report.csv"
}
```

7.3 Sensitivity

Parameters include:

```
:param scenario_folder: path to folder with scenario_params_*.csv
:param method: "correlation", "morris", or "sobol"
:param results_csv: path to results CSV (for correlation)
```

```
:param target_variable: string or list of strings (for correlation)
:param output_csv: results file
:param n_morris_trajectories: int
:param num_levels: Morris design
:param n_sobol_samples: int
```

Example JSON:

```json
"sensitivity": {
  "perform_sensitivity": true,
  "scenario_folder": "scenarios",
  "method": "morris",
  "results_csv": "results_scenarioes/merged_daily_mean_scenarios.csv",
  "target_variable": [
    "Heating:EnergyTransfer [J](Hourly)",
    "Cooling:EnergyTransfer [J](Hourly)",
    "Electricity:Facility [J](Hourly)"
  ],
  "output_csv": "multi_corr_sensitivity.csv",
  "n_morris_trajectories": 10,
  "num_levels": 4
}
```

7.4 Surrogate

Here, the user can select which column of results should be used to develop a surrogate model:

```json
"surrogate": {
  "perform_surrogate": true,
  "scenario_folder": "scenarios",
  "results_csv": "results_scenarioes/merged_daily_mean_scenarios.csv",
  "target_variable": "Heating:EnergyTransfer [J](Hourly)",
  "model_out": "heating_surrogate_model.joblib",
  "cols_out": "heating_surrogate_columns.joblib",
  "test_size": 0.3
}
```

7.5 Calibration

```json
"calibration": {
  "perform_calibration": true,
  "scenario_folder": "scenarios",
  "scenario_files": [
    "scenario_params_dhw.csv",
    "scenario_params_elec.csv"
  ],
  "subset_sensitivity_csv": "multi_corr_sensitivity.csv",
  "top_n_params": 10,
  "method": "ga",
  "use_surrogate": true,
  "real_data_csv": "data/mock_merged_daily_mean.csv",
  "surrogate_model_path": "heating_surrogate_model.joblib",
  "surrogate_columns_path": "heating_surrogate_columns.joblib",
  "calibrate_min_max": true,
  "ga_pop_size": 10,
  "ga_generations": 5,
  "ga_crossover_prob": 0.7,
  "ga_mutation_prob": 0.2,
  "bayes_n_calls": 15,
  "random_n_iter": 20,
  "output_history_csv": "calibration_history.csv",
  "best_params_folder": "calibrated",
  "history_folder": "calibrated"
}
```

End of Configuration

```
  }
}
```

8. Main Config

```
"main_config": {
  "paths": {
    "building_data": "data/df_buildings.csv",
    "fenez_excel": "excel_data/envelop.xlsx",
    "dhw_excel": "excel_data/dhw_overrides.xlsx",
    "epw_excel": "excel_data/epw_overrides.xlsx",
    "lighting_excel": "excel_data/lighting_overrides.xlsx",
    "hvac_excel": "excel_data/hvac_overrides.xlsx",
    "vent_excel": "excel_data/vent_overrides.xlsx"
  }
},
"excel_overrides": {
  "override_fenez_excel": false,
  "override_dhw_excel": false,
  "override_epw_excel": false,
  "override_lighting_excel": false,
  "override_hvac_excel": false,
  "override_vent_excel": false
},
"user_config_overrides": {
  "override_fenez_json": true,
  "override_dhw_json": true,
  "override_epw_json": true,
  "override_lighting_json": true,
  "override_hvac_json": true,
  "override_vent_json": true,
  "override_geometry_json": true,
  "override_shading_json": false
},
"structuring": {
  "perform_structuring": true,
  "dhw": {
    "csv_in": "assigned/assigned_dhw_params.csv",
    "csv_out": "assigned/structured_dhw_params.csv"
  },
  "fenestration": {
    "csv_in": "assigned/assigned_fenez_params.csv",
    "csv_out": "assigned/structured_fenez_params.csv"
  },
  "hvac": {
    "csv_in": "assigned/assigned_hvac_params.csv",
    "build_out": "assigned/assigned_hvac_building.csv",
    "zone_out": "assigned/assigned_hvac_zones.csv"
  },
  "vent": {
    "csv_in": "assigned/assigned_ventilation.csv",
    "build_out": "assigned/assigned_vent_building.csv",
    "zone_out": "assigned/assigned_vent_zones.csv"
  }
}
```

These are files to override later from an Excel file, for example, to permanently change lookup tables based on calibration results or if the user has valid values. For now, in JSON, they will remain as is, with placeholders in the UI.

The **structuring** section should also be left as it is, locked in the UI for the user to see.

---

9. Result Visualizations

Here are the result files. We need to discuss how to visualize them:

```
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-85953b927ae1\results\merged_as_is.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-85953b927ae1\results\merged_daily_mean.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-
85953b927ae1\results_scenarioes\merged_as_is_scenarios.csv
```

```
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-
85953b927ae1\results_scenarioes\merged_daily_mean_scenarios.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-85953b927ae1\calibration_history.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-85953b927ae1\extracted_idf_buildings.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-85953b927ae1\multi_corr_sensitivity.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-85953b927ae1\validation_report_base.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-85953b927ae1\validation_report_scenarios.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-
85953b927ae1\calibrated\calibrated_params_scenario_params_dhw.csv
D:\Documents\E_Plus_2030_py\output\091e2b42-b8d3-4bdb-883c-
85953b927ae1\calibrated\calibrated_params_scenario_params_elec.csv
```

We can plan different visualization strategies (plots, charts, tables) for these outputs based on user needs.