FORGE: Flexible Optimization of Routes for GHG & Energy Model Structure and Data Pipeline

FORGE Developers

1. Overview

Steel and aluminum production are energy-intensive industrial sectors with significant greenhouse gas (GHG) emissions. Accurate modeling of production routes, energy consumption, and emissions is essential for assessing decarbonization pathways and policy impacts. However, most existing models fall in one of two categories: detailed process-based LCA models that are complex and inflexible, or top-down sectoral models that lack process detail, usually behind proprietary data.

FORGE (Flexible Optimization of Routes for GHG & Energy) aims to bridge this gap by providing an open-source, process-based techno-environmental model that is both flexible and user-friendly. It allows users to define and modify production routes, energy sources, and emission factors through simple YAML configuration files, enabling scenario analysis and sensitivity testing. Along with full transparency of inputs, outputs and parameters, FORGE is designed to be extensible to other industrial sectors beyond steel and aluminum. It can also be integrated with broader energy and economic models for comprehensive assessments.

FORGE is a process-based techno-environmental model for industrial production routes, currently focused on steel and aluminum. The model is implemented as a Python package with:

- a generic core engine (`forge.core.*`) that solves material and energy balances and computes emissions,
- sector descriptors and YAML datasets under `datasets/` that encode routes, stages, and parameters,
- an API layer (`forge.steel_core_api_v2`) used by both CLI and a Streamlit UI (`forge.apps.streamlit_app`).

Industrial LCAs and plant models are often opaque, non-parametric, or locked behind proprietary spreadsheets and datasets. This hinders verification, makes reproduction difficult, and complicates boundary comparisons.

FORGE is designed explicitly around transparency. All functions, configuration files, and validation scripts live in a public repository and are driven by human-readable YAML inputs. In the steel case, this addresses three recurring gaps:

- *Route transparency*: explicit unit-operation recipes and route configurations instead of aggregated route-level intensities;
- *Boundary correctness*: consistent handling of process gases, internal electricity credits, and co-products, avoiding double counting;

- *Reproducibility*: single-source YAML parameters, deterministic runs, and regression tests embedded in the codebase.

These design choices make FORGE suitable for academic studies, policy sensitivity analysis, and plant-level "what-if" scenarios, while remaining extensible to other sectors.

This document sketches the main architecture and—most importantly—the data pipeline from YAML input files to emissions outputs. The goal is to serve as a starting point for a full academic paper.

At its core, FORGE implements a material balance solver that walks upstream from final demand to determine production levels per process. Given production levels, energy intensities, and carrier shares, it builds per-process energy balances. This makes the model fully integrated, which allows changes in parameters (ie Blast Furnace energy intensity) to propagate both upstream and downstream. Another feature is that FORGE models process gas recovery in coke making, blast furnace and basic oxygen furncace, in line with the real world practices. A gas-routing module implements process-gas recovery, routing between direct use and electricity, and blended emission factors for gas and internal electricity. Finally, emissions are computed using energy and direct emissions, applying blended emission factors where relevant.

Aluminum implementation does not share the same intricacies of steel, and is shown here as a proof of concept of FORGE's flexibility to model other sectors. The aluminum dataset includes primary and remelt routes, downstream alloying and finishing, and direct process emissions. It does not have process gas recovery or routing, nor alternative data-sets for different scenarios (likely/optimistic/pessimistic).

2. High-level Architecture

2.1. Core engine (`forge.core`)

Key modules:

- `core.models`: defines the `Process` class and constants such as `OUTSIDE_MILL_PROCS`.
- `core.engine`:
  - `calculate_balance_matrix`: material balance solver that walks upstream from final demand to determine production levels per process.
  - `calculate_energy_balance`: builds per-process energy balances from production levels, energy intensities, and carrier shares.
  - `calculate_emissions`: computes energy and direct emissions by process, given energy balances, emission factors, and optional process-specific EFs.
- `core.gas`:
  - `apply_gas_routing_and_credits`: implements process-gas recovery, routing between direct use and electricity, and blended emission factors for gas and internal electricity.
  - `compute_inside_energy_reference_for_share`: reference plant run to compute total gas and electricity for blending.

- `core.io`:
  - `load_data_from_yaml`: generic YAML loader with light normalization.
  - `load_recipes_from_yaml`: loads recipes, evaluating per-process expressions with access to parameters and energy tables.
- `core.routing`:
  - `STAGE_MATS`: default mapping of stage ids to materials (used as a fallback for legacy steel).
  - `build_route_mask`: simple route masks for steel routes (BF–BOF, DRI–EAF, EAF-scrap, External).
  - `enforce_eaf_feed`: clamps EAF recipes to a single feed (scrap, DRI, or pig iron) to avoid mixed-feed routes.
- `core.runner`:
  - `CoreScenario`: dataclass bundling all inputs the engine needs (recipes, energy tables, EFs, route mask, etc.).
  - `run_core_scenario`: orchestrates material balance → energy balance → gas routing → emissions (and optional costs).

2.2. Sector descriptors (`forge.descriptor`)

Each dataset folder, e.g. `datasets/steel/likely` or `datasets/aluminum/baseline`, contains a `sector.yml` that describes:

- *stages* (ids, materials, labels),
- *stage menu* (what appears in the UI's "Product" radio),
- *routes* (BF–BOF, DRI–EAF, EAF-scrap, etc.), including process enables/disables and optional feed modes,
- *process roles* (e.g. gas sources),
- gas configuration (carriers, utility process, reference stage).

The descriptor loader (`forge.descriptor.sector_descriptor`) turns this YAML into a `SectorDescriptor` object used by:

- `scenario_resolver` helpers (route masks, stage material resolution, feed mode),
- the Streamlit UI for stage and route selection,
- the API layer to build a `CoreScenario`.

2.3. Steel API and Streamlit app

- `forge.steel_core_api_v2`:
  - `RouteConfig`: encapsulates route preset, stage key/role, demand and optional pre-selections.
  - `ScenarioInputs`: wraps a scenario dict (YAML overrides) and the route config.
  - `run_scenario`: main entrypoint for steel; responsible for loading YAML, applying overrides, building the production route, calling the core runner, and shaping outputs.
- `forge.apps.streamlit_app`:
  - multi-sector UI (Steel / Aluminum) with sector gate,
  - dataset and scenario selection (sidebar),
  - downstream choices (per-stage ambiguous picks) rendered from the descriptor and recipe graph,
  - model execution via `run_scenario` and result display.

2.4. Software architecture and design rationale

Beyond the module-level split, FORGE follows a three-tier architecture:

- a *computational core* (`forge.core.*`) that implements mass–energy balances, gas routing, and emissions; it is pure Python/numerics with no UI or file I/O;
- an *API/service layer* (`forge.steel_core_api_v2`, `forge.scenarios.*`) that loads YAML data, applies overrides, resolves routes, and builds a `CoreScenario` for execution;
- a *presentation layer* (`forge.apps.streamlit_app`, CLI helpers) for interactive or batch runs.

This separation lets domain experts change scenarios and parameters without touching core algorithms, while keeping the numerical engine small and testable.

Configuration is kept in YAML rather than hard-coded constants. Sector descriptors define stages, routes, and gas roles; dataset folders define recipes, energy tables, and emission factors; scenario YAMLs encode route choices and parameter overrides. The API layer simply merges these structured inputs into a resolved scenario, so changing assumptions is a matter of editing configuration rather than code.

Route resolution is deterministic. Starting from a demanded material at a chosen stage, FORGE walks the recipe graph upstream and enforces a single producer for each material. Ambiguities (e.g. market vs in-mill nitrogen) are resolved either from scenario YAMLs or from explicit UI picks; otherwise the core raises an error rather than guessing. This ensures that repeated runs with the same inputs produce identical flows and emissions.

The main API dataclasses (`RouteConfig`, `ScenarioInputs`, `CoreScenario`, `RunOutputs`) act as data contracts between layers. They make the overall interface explicit (what must be provided, what is returned) and enable additional front-ends (REST services, Monte Carlo drivers, policy simulators) without touching the core math.

3. Main algorithms

At its core, FORGE implements a material balance solver that walks upstream from a final demand to determine production levels per process. Given production levels, energy intensities, and carrier shares, it builds per-process energy balances. Finally, emissions are computed using energy and direct emissions, applying blended emission factors where relevant.

3.1. Material balance

The material balance is computed by `calculate_balance_matrix`, which walks the recipe graph upstream from a user-specified final demand, enforcing a single enabled producer per material. The function takes as input the recipes, the final-demand dictionary, and a per-process on/off mask (`production_routes`), and returns a balance matrix and per-process production levels. The graph walk implements, in a constructive way, the steady-state balance equations formalized below.

3.2. Material balance formulation

Let $\mathcal{R}$ be the set of production recipes (processes), indexed by r, and let $\mathcal{M}$ be the set of materials, indexed by m. Each recipe r consumes inputs $a_{r,m} \geq 0$ and produces outputs $b_{r,m} \geq 0$. The user specifies a final demand vector

$$f_m \geq 0 \qquad (m \in \mathcal{M}).$$

Producers.. For each material m, the set of internal producers is

$$\mathcal{P}(m) = \{r \in \mathcal{R} \mid b_{r,m} > 0\}.$$

Enabled or disabled production routes are indicated by

$$\pi_r \in \{0, 1\},$$

with $\pi_r = 0$ meaning that recipe r is not available.

Credit rules.. Some by-products generate credits for other target materials. Let $\mathcal{C} \subseteq \mathcal{M} \times \mathcal{M}$ be the set of credit pairs (j, t) with associated ratios $\rho_{j \to t} \geq 0$. The corresponding effective output of recipe r for material m is

$$\tilde{b}_{r,m} \;=\; b_{r,m} \;+\; \sum_{j:(j,m)\in\mathcal{C}} \rho_{j\to m}\, b_{r,j}.$$

Decision variables.. Let $x_r \geq 0$ denote the number of production runs of recipe r, and let $e_m \geq 0$ denote external purchases of material m. Disabled routes satisfy $x_r = 0$ whenever $\pi_r = 0$.

Material balances.. For every material $m \in \mathcal{M}$, the steady-state material balance is

$$\sum_{r\in\mathcal{R}} \left(\tilde{b}_{r,m} - a_{r,m}\right) x_r \;+\; e_m \;=\; f_m. \tag{1}$$

Materials without internal producers ($\mathcal{P}(m) = \emptyset$) reduce to

$$-\sum_{r} a_{r,m}\, x_r \;+\; e_m = f_m.$$

Balance matrix.. Once a feasible vector x_r has been determined, we construct a balance matrix with one row per active process, plus rows for external inputs and final demand. For each process r and material m, the net internal flow is

$$B_{r,m} \;=\; \big(b_{r,m} - a_{r,m}\big)x_r.$$

External inputs and final demand are represented by

$$B_{\text{ext},m} = e_m, \qquad B_{\text{fd},m} = -f_m.$$

The resulting matrix

$$\mathbf{B} = \begin{pmatrix} B_{r_1,m_1} & \cdots & B_{r_1,m_k} \\ \vdots & & \vdots \\ B_{r_p,m_1} & \cdots & B_{r_p,m_k} \\ B_{\text{ext},m_1} & \cdots & B_{\text{ext},m_k} \\ B_{\text{fd},m_1} & \cdots & B_{\text{fd},m_k} \end{pmatrix}$$

contains all net flows for all materials. For any material with full internal balancing, the corresponding column of $\mathbf{B}$ satisfies

$$\sum_{\text{rows } i} B_{i,m} = 0,$$

up to numerical tolerance.

3.3. Energy balance formulation

Let $\mathcal{R}$ be the set of processes (recipes), indexed by r, and let $\mathcal{C}$ be the set of energy carriers, indexed by c.

For each process $r \in \mathcal{R}$:

- $x_r \geq 0$ is the production level (number of runs of process r),
- $e_r \geq 0$ is the total energy intensity per run (in MJ per run),
- $s_{r,c} \in [0,1]$ is the share of carrier c in the energy use of process r,

such that

$$\sum_{c\in\mathcal{C}} s_{r,c} = 1 \quad \text{for any process } r \text{ with defined carrier shares.}$$

The energy consumption of process r using carrier c is then

$$E_{r,c} \;=\; x_r\, e_r\, s_{r,c}, \qquad r \in \mathcal{R},\; c \in \mathcal{C}. \tag{2}$$

The total energy consumption by carrier c across all processes is

$$E_c^{\text{tot}} \;=\; \sum_{r\in\mathcal{R}} E_{r,c} \;=\; \sum_{r\in\mathcal{R}} x_r\, e_r\, s_{r,c}, \qquad c \in \mathcal{C}. \tag{3}$$

Energy balance matrix.. We collect these quantities in an energy balance matrix with one row per active process and one additional "TOTAL" row. For each process r and carrier c we set

$$B^E_{r,c} = E_{r,c},$$

and the total row is given by

$$B^E_{\mathrm{TOTAL},c} = E^{\mathrm{tot}}_c = \sum_{r \in \mathcal{R}} B^E_{r,c}.$$

The resulting matrix

$$\mathbf{B}^E = \begin{pmatrix} B^E_{r_1,c_1} & \cdots & B^E_{r_1,c_k} \\ \vdots & & \vdots \\ B^E_{r_p,c_1} & \cdots & B^E_{r_p,c_k} \\ B^E_{\mathrm{TOTAL},c_1} & \cdots & B^E_{\mathrm{TOTAL},c_k} \end{pmatrix}$$

contains, in each column, the disaggregated energy use per carrier and its system-wide total in the last row. "Electricity" is always included as one of the carriers in $\mathcal{C}$, even if its associated entries are zero.

3.4. Emission calculation

We build process-level greenhouse gas emissions by combining energy use, carrier-specific emission factors, and process-specific direct emission factors, with special treatment for electricity and process gas in integrated steel plants.

Let $\mathcal{R}$ be the set of processes and $\mathcal{C}$ the set of energy carriers. As in the previous subsection, $x_r \geq 0$ denotes the production level (number of runs) of process $r \in \mathcal{R}$, and $E_{r,c} \geq 0$ the energy consumption of carrier $c \in \mathcal{C}$ by process r (in MJ).

Emission factors and process sets.. For each carrier $c \in \mathcal{C}$ let

$$\mathrm{EF}^{\mathrm{carrier}}_c \geq 0$$

denote the emission factor (e.g. in kg CO_2e per MJ). For each process $r \in \mathcal{R}$ let

$$\mathrm{EF}^{\mathrm{proc}}_r \geq 0$$

denote its specific direct emission factor per unit output, which is scaled by x_r in the calculation.

We distinguish three process subsets:

- $\mathcal{R}_{\mathrm{purchase}} \subseteq \mathcal{R}$: processes that represent pure market purchases ("from market", "purchase").
- $\mathcal{R}_{\mathrm{outside}} \subseteq \mathcal{R}$: processes located downstream or outside the mill boundary (e.g. finishing).
- $\mathcal{R}_{\mathrm{direct}} \subseteq \mathcal{R}$: onsite processes that are allowed to have direct emissions accounted explicitly.

Electricity mix.. Let $f_{\mathrm{int}} \in [0,1]$ denote the fraction of plant electricity supplied by internal generation. We denote by

$$\mathrm{EF}^{\mathrm{grid}}_{\mathrm{el}} \quad \text{and} \quad \mathrm{EF}^{\mathrm{int}}_{\mathrm{el}}$$

the emission factors for grid and internal electricity, respectively. For processes inside the mill boundary, electricity uses the blended factor

$$\mathrm{EF}_{\mathrm{el}}^{\mathrm{mix}} = f_{\mathrm{int}}\,\mathrm{EF}_{\mathrm{el}}^{\mathrm{int}} + (1 - f_{\mathrm{int}})\,\mathrm{EF}_{\mathrm{el}}^{\mathrm{grid}}.$$

For outside-mill processes ($r \in \mathcal{R}_{\mathrm{outside}}$), electricity is always taken from the grid.

Process-level emissions.. For each process $r \in \mathcal{R}$ we define energy-related emissions E_r^{em} and direct emissions D_r.

(i) Market purchase processes. For processes treated as pure purchases ($r \in \mathcal{R}_{\mathrm{purchase}}$), all emissions are accounted as direct, and there are no energy emissions:

$$E_r^{\mathrm{em}} = 0, \qquad D_r = x_r\,\mathrm{EF}_r^{\mathrm{proc}}. \tag{4}$$

(ii) Onsite processes. For onsite processes ($r \notin \mathcal{R}_{\mathrm{purchase}}$), energy-related emissions are computed from carrier use. The electricity emission factor for process r is

$$\mathrm{EF}_r^{\mathrm{el}} = \begin{cases} \mathrm{EF}_{\mathrm{el}}^{\mathrm{grid}}, & r \in \mathcal{R}_{\mathrm{outside}}, \\ \mathrm{EF}_{\mathrm{el}}^{\mathrm{mix}}, & r \notin \mathcal{R}_{\mathrm{outside}}. \end{cases}$$

Then

$$E_r^{\mathrm{em}} = \sum_{c \in \mathcal{C} \setminus \{\mathrm{el}\}} E_{r,c}\,\mathrm{EF}_c^{\mathrm{carrier}} \;+\; E_{r,\mathrm{el}}\,\mathrm{EF}_r^{\mathrm{el}}, \tag{5}$$

with the exception that, for the coke production process, coal is treated as feedstock rather than fuel and is therefore excluded from the energy term:

$$\text{if } r = \text{“Coke Production”, then the term with } c = \text{Coal is omitted in (5).} \tag{6}$$

Direct emissions for onsite processes are only included for a whitelisted subset $\mathcal{R}_{\mathrm{direct}}$:

$$D_r = \begin{cases} x_r\,\mathrm{EF}_r^{\mathrm{proc}}, & r \in \mathcal{R}_{\mathrm{direct}}, \\ 0, & r \notin \mathcal{R}_{\mathrm{direct}}. \end{cases} \tag{7}$$

Total emissions.. The total emissions for process r are

$$T_r = E_r^{\mathrm{em}} + D_r, \qquad r \in \mathcal{R}. \tag{8}$$

In tabular form, we collect for each active process r the triplet $(E_r^{\mathrm{em}}, D_r, T_r)$, which corresponds to the columns “Energy Emissions”, “Direct Emissions”, and “TOTAL CO_2e” in the emissions matrix returned by the model.

4. Data Configuration

FORGE is driven by human-readable YAML files.

4.1. Recipes

The main process configuration is done via `recipes.yml`, which defines per-process input and output coefficients. Each process has a recipe dict with `inputs` and `outputs` sub-dicts mapping material names to quantities per run. Quantities can be numeric constants or expressions that reference parameters or energy intensities. For example, the Blast Furnace is configured as follows:

```
- process: Blast Furnace
  inputs:
    Sinter: "1.03 * (1 / iron_content_pig_iron) * (feo3_lump / (blend.sinter * feo3_sinter + blend.p
    Pellet: "1.03 * (1 / iron_content_pig_iron) * (feo3_lump / (blend.sinter * feo3_sinter + blend.p
    Iron Ore: "1.03 * (1 / iron_content_pig_iron) * (feo3_lump / (blend.sinter * feo3_sinter + blend
    Nitrogen: 0.0750
    Oxygen: 0.0800
    Limestone: 0.0400
    Coke: >
      energy_int['Blast Furnace']
      * energy_shares['Blast Furnace']['Coke']
      / energy_content['Coke']
    Coal: >
      energy_int['Blast Furnace']
      * energy_shares['Blast Furnace']['Coal']
      / energy_content['Coal']
  outputs:
    Pig Iron: 1.0
    BF Process Gas: "process_gases['BF Process Gas']['recovery_fraction'] * energy_int['Blast Furnac
```

As the most complex process in the entire production chain, the Blast Furnace recipe warrants a detailed explanation. The inputs include iron sources (sinter, pellet, lump ore) calculated based on their iron content and blend ratios, as well as auxiliary materials like nitrogen, oxygen, limestone, coke, and coal. The coke and coal inputs are dynamically calculated based on the energy intensity of the Blast Furnace and the respective energy shares defined in the energy matrix. The outputs include pig iron and recovered blast furnace process gas, with the latter calculated based on a recovery fraction parameter and the energy intensity. Note that the use of expressions allows for flexibility in adjusting the recipe based on different parameters and energy intensities. Also, the energy to mass conversion for both coal and coke are needed to ensure upstream processes (Coke Making, Coal Purchase) are correctly sized.

Here, `scrap_fraction` is a parameter defined in `parameters.yml`. When loading recipes, the API evaluates these expressions in a namespace that includes all parameters and energy intensities, allowing dynamic adjustment of recipes based on scenario inputs.

The user must make sure that recipe chains are integrated, meaning that outputs of one process are inputs to another, forming a connected graph from raw materials to final products. If the chain breaks anywhere, upstream or downstream processes will not run, thus not appear on Production Runs, which can thus be used as a debugger. All possible processes can have recipes on the same file, as a separate implementation was done to specifically solve ambiguities. The core calculations are will not run if this is not done, but will also not disambiguate, by default, to avoid model guessing.

All recipes must be configured as unitary processes (1 unit of process output per run – NOT PER FINAL PRODUCT RUN). So the Nitrogen recipe is scaled to one unit of nitrogen, not unit of steel.

The model engine handles all scaling internally. Recipes are usually defined in constant, numerical units; however, some main inputs and outputs can be defined as expressions that reference parameters or energy intensities. This allows dynamic adjustment of recipes based on scenario inputs. Scenario overrides from default will be explained later.

4.2. Energy Intensities

As with recipes, energy intensities are defined per process in `energy_int.yml` as MJ per run. The engine scales accordingly. For example:

```
Basic Oxygen Furnace: 2.0
Electric Arc Furnace: 10.0
```

4.3. Energy Matrix

The energy matrix (`energy_matrix.yml`) defines the share of each energy carrier used by each process. Shares are fractions that sum to 1.0 per process, and a specific test was added for this purpose. For example:

```
Basic Oxygen Furnace:
  Natural Gas: 0.7
  Electricity: 0.3
Electric Arc Furnace:
  Electricity: 1.0
```

4.4. Process gases

Process gas recovery can be done in the Blast Furance, Basic Oxygen Furnace and Coke Making processes. The configuration for these gases is done in `process_gases.yml`, which defines per-volume energy content and also energy density in kg per normal cubic meterm as well as the recovery fractions (in energy terms), and the producing process. This configuration ensures mass values for recipes and energy values for the gas recovery utility. For example, the Blast Furnace process gas is defined as:

```
  BF Process Gas:
    source_process: "Blast Furnace"
    energy_gj_per_ndam3: 3.232
    density_kg_per_nm3: 1.250
    recovery_fraction: 0.30
```

4.5. Auxiliary energy and emissions files

Other YAML files necessary for model execution include the Lower Heating Values (`energy_content.yml`), Emission Factors (`emission_factors.yml`), Electricity Emission Factors (`electricity_intensity.yml`), Direct Process Emissions (`process_emissions.yml`), and Parameters (`parameters.yml`). An auxiliary file, `ghg_protocol.yml` defines combustion only emission factors, as opposed to the main file, with total (life-cycle) emission factors. This is used in gas routing to avoid double counting of process gas carbon.

4.6. Sector Descriptor

Due to the flexible nature of FORGE, each material needs a specific configuration file, `sector.yml`, that defines the stages, routes, process roles and gas configuration. The stages are the possible products that can be modeled. For example, for steel there are the cast steel, with locked choices, crude steel, with upstream processes that can be selected, and also Finished Steel, with all possible combinations. The stages also need to be defined, so the UI can be built uppon it. Whenever multiple producers exists, the user must create a stage. The file also defines what possible routes exists for that specific product. So, for steel, there is the Blast Furnace-Basic Oxygen Furnace (BF-BOF), the BF-BOF with charcoal, the Direct Reduced Iron-Electric Arc Furance (DRI-EAF) and the Scrap-EAF. This file indicates which processes can be producers of recovery gas.

5. Data Pipeline

This section focuses on how data flows from YAML to emissions. The pipeline is essentially the same for steel and aluminum; differences are encoded in the sector descriptors and dataset content.

5.1. Input files per dataset

For each dataset folder under `datasets/<sector>/<variant>/`, FORGE expects a consistent set of core YAML files:

- `sector.yml`: descriptor (stages, routes, process roles, gas config).
- `recipes.yml`: process-level input/output coefficients.
- `energy_int.yml`: energy intensity per process (MJ per run).
- `energy_matrix.yml`: carrier shares per process (fractions).
- `energy_content.yml`: lower heating values (MJ per unit of fuel).
- `emission_factors.yml`: fuel emission factors (gCO_2e/MJ).
- `process_emissions.yml`: direct emissions per process ($kgCO_2e/t$ output).
- `parameters.yml`: scalar parameters used in recipe expressions and gas routing (e.g. yields, fractions).
- `mkt_config.yml`: market vs endogenous production flags.
- optional: `process_gases.yml` (gas meta), energy and material price tables, electricity intensity by country.

Scenario YAMLs (e.g. `scenarios/BF_BOF_coal.yml` or `scenario_aluminum.yml`) provide per-run overrides:

- route overrides (enable/disable specific processes),
- modifications to energy intensities, shares, or EFs,

- parameter overrides and recipe tweaks,
- gas-routing fractions and optional price/EF tweaks.

5.2. From scenario to `CoreScenario`

At a high level, the API does:

1. Load the sector descriptor for the dataset.
2. Determine the route preset and stage key from the scenario name or scenario dict.
3. Load base YAML tables (recipes, energy, EFs, parameters, markets).
4. Apply scenario overrides (fuel substitutions, energy tables, parameters, recipes) and reload recipes to re-evaluate expressions.
5. Build a route mask using the descriptor and route preset.
6. Construct the production route from ambiguous picks (or defaults) using `_build_routes_from_picks`.
7. Build a `CoreScenario` via `forge.scenarios.builder` (`build_core_scenario`), which:
   - locks route preset and EAF feed mode,
   - carries energy tables, EFs, process EFs, gas config, and any process-emission whitelists,
   - handles fallback materials and optional cost inputs.

5.3. Core runner and gas routing

Given a `CoreScenario`, `run_core_scenario` performs:

1. **Material balance**:
   - Build final demand dictionary for the chosen stage material (e.g. `Finished Products`, `Primary Aluminum`).
   - Call `calculate_balance_matrix` with recipes, demand, and route mask. This returns:
     - a balance matrix (rows = processes + external + final demand; columns = materials),
     - per-process production levels (# runs).
2. **Energy balance**:
   - Expand energy tables for all active processes (`expand_energy_tables_for_active`).
   - Call `calculate_energy_balance` to obtain carrier-by-process energy consumption plus a TOTAL row.
3. **Gas routing and credits**:
   - Build a gas-routing scenario (gas config, route preset, demand, stage reference, gas fractions).
   - Call `apply_gas_routing_and_credits`:
     - Identify process-gas sources and compute total recovered gas.
     - Split recovered gas between direct use and electricity (utility plant) according to `direct_use_fraction`.
     - Compute process-gas EF from fuel blends (excluding electricity and the carrier itself), and blended gas EFs from internal vs grid contributions.

- Adjust the energy balance by:
  - adding an internal electricity export row (Utility Plant),
  - redirecting a fraction of gas consumption to the process-gas carrier for direct use.
- Return updated energy balance, updated energy EFs, and a meta dict with diagnostics.

4. **Emissions**:
   - Use a robust wrapper to call `calculate_emissions` with whatever combination of arguments it accepts (maintains backwards compatibility).
   - `calculate_emissions` separates market vs onsite processes, applies blended electricity and fuel EFs, and adds direct process emissions when whitelisted.
   - A total row/column is computed when necessary; total emissions are returned in kg CO_2e.

5.3.1. Gas recovery and internal electricity production

In the steel datasets, several processes (Coke Production, Blast Furnace, and Basic Oxygen Furnace) are marked as gas sources in the sector descriptor and mapped to a process-gas carrier (e.g. `Process Gas (internal)`). Additional metadata in `process_gases.yml` specifies per-unit energy content, recovery fractions, and the producing process. The API attaches this metadata to the parameter namespace and gas configuration before building a `CoreScenario`.

The gas module then computes, for a given scenario:

- the total recovered process gas G_{proc} in MJ from all gas sources,
- a split between direct fuel use and electricity generation, controlled by the scenario's `direct_use_fraction` and optional `electricity_fraction`,
- an effective process-gas emission factor EF_{proc} built from the upstream energy-carrier mix of gas-producing units, excluding electricity and the gas carrier itself.

Let f_{dir} be the share routed to direct use and f_{el} the share routed to electricity (with $f_{\text{dir}} + f_{\text{el}} \leq 1$). Define

$$G_{\text{dir}} = f_{\text{dir}}\, G_{\text{proc}}, \qquad G_{\text{el}} = f_{\text{el}}\, G_{\text{proc}}.$$

The utility process has a fixed electrical efficiency η_{el} read from its recipe (MJ of electricity per MJ of gas). The potential internal electricity from process gases in the present run is

$$E_{\text{int}} = \eta_{\text{el}}\, G_{\text{el}}.$$

To avoid circularity in the blended electricity emission factor, FORGE also performs a reference run via `compute_inside_energy_reference_for_share`, using a fixed route and stage defined in the descriptor (for steel, an IP3 "inside-mill" boundary). That reference run yields an in-mill electricity demand $E_{\text{elec}}^{\text{ref}}$. The fraction of in-mill electricity credited to internal gas is then

$$f_{\text{int}} = \min\left(1,\ \frac{E_{\text{int}}}{E_{\text{elec}}^{\text{ref}}}\right),$$

and the internal electricity emission factor is a simple proxy

$$\mathrm{EF}_{\mathrm{int}} = \frac{\mathrm{EF}_{\mathrm{proc}}}{\eta_{\mathrm{el}}}.$$

The plant-level electricity emission factor used in emissions is the blend

$$\mathrm{EF}_{\mathrm{elec,plant}} = f_{\mathrm{int}}\,\mathrm{EF}_{\mathrm{int}} + (1 - f_{\mathrm{int}})\,\mathrm{EF}_{\mathrm{grid}},$$

where $\mathrm{EF}_{\mathrm{grid}}$ is the grid electricity factor (possibly overridden by a country code).

Direct use of process gas is treated as a substitution in the energy balance. The model computes a fraction $f_{\mathrm{gas,int}}$ of total plant gas demand that can be supplied by recovered process gas and:

- reduces natural-gas consumption by $f_{\mathrm{gas,int}}$ at each process, and
- adds the same MJ to the process-gas carrier column.

The resulting blended gas emission factor is

$$\mathrm{EF}_{\mathrm{gas,blend}} = f_{\mathrm{gas,int}}\,\mathrm{EF}_{\mathrm{proc}} + (1 - f_{\mathrm{gas,int}})\,\mathrm{EF}_{\mathrm{nat,grid}}.$$

Both $\mathrm{EF}_{\mathrm{elec,plant}}$ and $\mathrm{EF}_{\mathrm{gas,blend}}$ are fed into `calculate_emissions` so that process gases are credited once, via the electricity block and gas substitution, without double counting.

The implementation is intentionally attributional. It does not add an explicit combustion step for process-gas carbon, does not model auxiliaries or fugitive emissions for gas handling, and keeps most upstream burdens at the originating units. As a result, the internal electricity factor should be read as a consistent proxy derived from upstream carriers, suitable for comparative scenarios and sensitivity analysis.

6. Automation and User-Defined Scenarios

Reproducible runs and user experiments are driven by simple CLI and Makefile wrappers; no UI interaction is required.

Makefile targets.. Common runs are exposed as `make` targets that delegate to `scripts/run_profiles.py`: `make finished` (paper finished portfolio), `make paper` (paper portfolio), `make aluminum` (baseline aluminum batch), `make mc-as-cast` and `make mc-finished` (Monte Carlo sweeps), plus `make list` to show available profiles. Each target maps to a profile in `configs/run_profiles.yml`, which records the command and environment variables needed for that run.

Adding a profile.. Profiles are YAML entries with a description, an `env` map, and a `cmd` array. Example:

```
my_profile:
  desc: "Custom batch"
  env: { PYTHONPATH: src }
  cmd:
    - "python3"
    - "-m"
    - "forge.cli.steel_batch_cli"
    - "run"
    - "--spec"
    - "configs/my_batch.yml"
```

Execute with `make run PROFILE=my_profile`.

Single-scenario run..

1. Choose a dataset (e.g. `datasets/steel/likely`) and copy an existing scenario YAML under `scenarios/`.
2. Edit `route_overrides` (enable/disable processes) and, if needed, adjust `energy_int`, `energy_matrix`, `emission_factors`, `parameters`, or `recipe_overrides`.
3. Run via:

```
PYTHONPATH=src python3 -m forge.cli.steel_batch_cli run \
  --data-dir datasets/steel/likely \
  --scenario datasets/steel/likely/scenarios/BF_BOF_coal.yml \
  --route BF-BOF --stage-key Finished --country BRA \
  --output results/custom_run.json
```

Batch spec.. For multiple runs, define a spec with optional `defaults` and a `runs` list:

```
defaults:
  data_dir: datasets/steel/likely
  route: {route_preset: BF-BOF, stage_key: Finished, demand_qty: 1000}
runs:
  - name: coal
    scenario_file: datasets/steel/likely/scenarios/BF_BOF_coal.yml
  - name: charcoal
    scenario_file: datasets/steel/likely/scenarios/BF_BOF_charcoal.yml
    overrides: { emission_factors.Charcoal: 5.0 }
```

Run with `python3 -m forge.cli.steel_batch_cli run --spec configs/my_batch.yml --output results/my_batc`
Each run can set country codes, `picks_by_material`, or `pre_select_soft`.

Monte Carlo.. The Monte Carlo utility (`forge.scenarios.monte_carlo_tri`) samples energy intensities, carrier shares, and emission factors between min/mode/max datasets and can blend multiple runs from a portfolio. A typical call is:

```
python3 -m forge.scenarios.monte_carlo_tri \
  --min datasets/steel/optimistic_low \
  --mode datasets/steel/likely \
  --max datasets/steel/pessimistic_high \
  --route BF-BOF \
  --portfolio configs/as_cast_portfolio.yml \
  --countries BRA \
  --n 500 \
  --out results/mc_as_cast
```

Outputs include `mc_summary.csv` and optional histogram/ECDF PNGs.

6.1. Aluminum specifics

Aluminum uses the same core but different descriptors and datasets:

- **Routes and stages** are defined in `datasets/aluminum/baseline/sector.yml`, with stages for primary, remelt, and finished aluminum.
- **Downstream alloying and finishing** are described via recipes for:
  - *Remelt blending* and alloy series (1, 5, 6, 7),

  - metallurgical aluminum aggregation,
  - rolling, extrusion, ingot casting, and raw products,
  - manufactured products and final coatings (no coating, powder coating, liquid painting).

- **Direct process emissions** for aluminum are configured via `process_emissions.yml` and a scenario-level whitelist (`allow_direct_onsite`) that flags which processes count as onsite direct emitters.

7. Uncertainty, sensitivity, and validation

Uncertainty in FORGE enters primarily through energy intensities, carrier shares, emission factors, and selected process parameters. For steel, the repository provides three dataset variants (`optimistic_low`, `likely`, `pessimistic_high`) with consistent YAML schemas; aluminum currently has a single baseline dataset.

In the Streamlit interface, users can:

- switch between dataset variants in the sidebar (steel only);
- choose country-specific grid factors from `electricity_intensity.yml`;
- adjust key parameters in the "Sensitivity" and "Static Mods" tabs (e.g. Blast Furnace energy intensity schedules, DRI/EAF mixes, scrap shares, downstream choices).

These controls exercise the same API used by scripts, so any combination of UI settings can be reproduced with a `RouteConfig` and scenario dict.

For Monte Carlo analysis, `forge.scenarios.monte_carlo_tri` samples uncertain YAML tables between the min/likely/max datasets using triangular distributions, renormalizes energy matrices, and runs many independent scenarios through `run_scenario`. It writes per-sample totals and histograms/ECDFs of emission factors, enabling uncertainty bands around route comparisons or downstream portfolios.

Validation in the steel case is currently focused on:

- mass, energy, and elemental closure checks at the process and plant levels;
- unit tests for recipes and yields, including internal consistency of blend parameters;
- regression-style tests that compare route-level cradle-to-gate intensities against published ranges (e.g. Worldsteel 2023);
- sensitivity and Monte Carlo sweeps to ensure that the model responds smoothly to parameter perturbations rather than relying on tuned point estimates.

Aluminum validation is more limited and currently serves mainly as a proof of concept for extending the framework to other sectors.

8. Usage

FORGE can be used via two main interfaces, an interactive Streamlit app and a Python API for scripting and batch runs.

8.1. Streamlit app

The Streamlit app is launched via:

```
streamlit run src/forge/apps/streamlit_app.py
```

The app's landing screen asks for the industrial sector (Steel or Aluminum). Once a sector is selected, the sidebar lets the user choose:

- a dataset variant (e.g. `datasets/steel/likely`);
- a scenario file (e.g. `BF_BOF_coal.yml`);
- a product stage (Validation / Crude steel / Finished);
- a country code for grid electricity emission factors.

The main panel provides tabs for the core model, 1-D sensitivity sweeps, and static modifications (e.g. Blast Furnace energy-intensity schedules, DRI/EAF mixes, scrap shares). All runs ultimately call the same `run_scenario` API described earlier.

8.2. CLI and scripting

FORGE's core can be exercised without the UI using the small engine CLI:

```
PYTHONPATH=src python3 -m forge.cli.engine_cli \
  --data datasets/steel/likely \
  --route BF-BOF \
  --stage Finished \
  --country BRA \
  --demand 1000 \
  --out results/engine_demo
```

This runs a single scenario through `run_scenario`, printing a summary (total CO_2e and top emitting processes) and writing CSV outputs for the energy balance and emissions, plus a small JSON manifest with the dataset path, arguments, and Git commit.

For larger sweeps and portfolios, `forge.cli.steel_batch_cli` implements a batch runner around the same API:

```
PYTHONPATH=src python3 -m forge.cli.steel_batch_cli run \
  --spec configs/finished_steel_portfolio.yml \
  --data-dir datasets/steel/likely \
  --output results/finished_portfolio_bf.json
```

The batch spec file describes one or more runs (route, stage, picks, scenario overrides), and the CLI fan-outs country codes or portfolio components as needed. Outputs are aggregated JSON/CSV suitable for post-processing (e.g. figures and tables in this paper).

Monte Carlo uncertainty analysis is available via `forge.scenarios.monte_carlo_tri`:

```
PYTHONPATH=src python3 -m forge.scenarios.monte_carlo_tri \
  --min datasets/steel/optimistic_low \
  --mode datasets/steel/likely \
  --max datasets/steel/pessimistic_high \
  --route BF-BOF \
  --n 500 \
  --out results/mc_bf_bof
```

This samples energy and emission tables between the three steel datasets, runs each draw through `run_scenario`, and writes CSV summaries plus basic histograms/ECDF plots of emission factors.

9. Next Steps for the Full Paper

This skeleton focuses on:

- code structure and responsibilities,
- file-level data pipeline,
- gas-routing scheme and its integration with the core.

For a complete academic paper, suggested additions:

- mathematical formulation of the balance solver (graph walk, uniqueness of producers, handling of external purchases),
- validation section (comparison vs external benchmarks for steel and aluminum),
- uncertainty ranges and scenario variants (likely/optimistic/pessimistic),
- discussion of limitations and future work (e.g. costs, other sectors).