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

FORGE Developers

November 26, 2025

1 Overview

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`

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.

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 preselections.
  - `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.

3 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.

3.1 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.

3.2 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.

3.3 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.

3.4 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.

4 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),
- explicit equations for gas routing and EF blending,
- 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).