
Master Degree course in ICT for Smart Societies

Master Degree Thesis

# Development of a graphical user interface for modeling and co-simulation of urban energy scenarios

**Supervisors**
Prof. Edoardo PATTI
Prof. Lorenzo BOTTACCIOLI
PhD. Pietro Rando Mazzarino

**Candidate**
Davood Shaterzadeh

Academic Year 2025-2026

# Acknowledgements

I would like to express my sincere gratitude to my supervisors, **Prof. Edoardo Patti** and **Prof. Lorenzo Bottaccioli**, for their guidance and support throughout the development of this thesis.

A special thanks goes to **Dr. Pietro Rando Mazzarino** for his constant availability, technical insights, and invaluable mentorship during the design and implementation of this platform. His feedback was essential in shaping the architectural decisions of this work.

I also wish to thank **Juri Sanni** for his collaboration and advice regarding the user experience design process, which helped refine the frontend interactions.

Finally, I am deeply grateful to my family and friends for their encouragement, patience, and unwavering support during my academic journey.

**Abstract**

The transition toward decentralized urban energy systems requires simulation environments capable of representing interactions across electrical, thermal, and building domains. However, configuring urban co-simulation workflows remains difficult, since users must define heterogeneous models, specify their data dependencies, and produce executable configurations, tasks that are often handled manually and therefore remain largely reserved for domain experts. This thesis addresses that challenge through the development of tools that support the visual authoring and automated configuration of urban energy co-simulation scenarios.

The thesis presents the Coesi UrbanSim Frontend, a web-based environment for the visual authoring of urban energy co-simulation scenarios, together with YAML Builder, a companion microservice that compiles user-defined scenario graphs into standardized executable YAML configurations. The frontend adopts a node-based interaction paradigm in which users compose workflows by connecting physical entities, data sources, and simulation models through typed links, while schema-driven interfaces support model configuration without requiring frontend code changes for each new model. In parallel, YAML Builder performs graph interpretation, validation, type coercion, and configuration generation, translating visual scenarios into artifacts suitable for distributed simulation. The approach is demonstrated through a case study based on a neighbourhood dataset from San Salvario, Turin. The results suggest that combining visual scenario composition with automated graph-to-YAML compilation improves transparency and reduces the effort required to configure complex urban energy co-simulation workflows.

# Contents

# List of Tables

# List of Figures

# List of Acronyms

| | |
|---|---|
| **ABM** | Agent-Based Model |
| **API** | Application Programming Interface |
| **DB** | Database |
| **DER** | Distributed Energy Resource |
| **DES** | District Energy System |
| **DH** | District Heating |
| **GIS** | Geographic Information System |
| **GUI** | Graphical User Interface |
| **HCI** | Human–Computer Interaction |
| **HDF5** | Hierarchical Data Format v5 |
| **HTTP** | Hypertext Transfer Protocol |
| **HVAC** | Heating, Ventilation, and Air Conditioning |
| **ICT** | Information and Communication Technology |
| **ID** | Identifier |
| **JSON** | JavaScript Object Notation |
| **KPI** | Key Performance Indicator |
| **ML** | Machine Learning |
| **NFR** | Non-Functional Requirement |
| **PV** | Photovoltaic |
| **REST** | Representational State Transfer |
| **UBEM** | Urban Building Energy Modelling |

| | |
|---|---|
| **UC** | Use Case |
| **UI** | User Interface |
| **URL** | Uniform Resource Locator |
| **UX** | User Experience |
| **UCD** | User-Centered Design |
| **WGS84** | World Geodetic System 1984 |
| **YAML** | YAML Ain't Markup Language |

# Chapter 1

# Introduction

## 1.1 Motivation and Problem Statement

Urban energy systems are increasingly characterized by the coexistence of multiple domains (e.g., electricity networks, thermal systems, buildings, and control/communication layers) whose interactions determine the overall system behavior. In such contexts, analyzing a single subsystem in isolation may lead to incomplete or misleading conclusions, because the effects of strategies (e.g., flexibility and demand response actions) depend on the response of the surrounding environment and on cross-domain constraints. Co-simulation has therefore emerged as a key approach to evaluate strategies from a broader and more realistic perspective by coupling heterogeneous models and simulators within a single experiment. This need is well highlighted in smart grid studies, where a "horizontal perspective" on the multitude of subsystems is required to properly interpret outcomes and enable meaningful benchmarking across scenarios. [16]

Despite the progress in co-simulation environments, the setup of complex energy co-simulations remains challenging. Multi-energy systems involve heterogeneous software ecosystems and non-trivial coupling and interoperability issues; as the number of simulated entities grows (e.g., from a few buildings to a district), manual configuration and human intervention increase and become a major bottleneck. [24]

A representative example of this complexity is the definition of how multiple simulators are parameterized and interconnected, including the explicit description of data-flow topology, initialization conditions, and output selection. Existing approaches often rely on configuration files (e.g., YAML templates) that encode scenario settings, simulator instances, and connections. While such templates can simplify scenario composition through reusable patterns, they still require expert users to reason about the connection topology and correctly specify parameters and instance definitions at scale. [20]

In parallel, urban-scale simulations require integrating heterogeneous geospatial and contextual data (e.g., building footprints and attributes, census-based information, weather data, and terrain models). Prior work has shown that microservices-based designs can improve modularity and scalability when integrating data sources and simulation components; however, the end-to-end workflow still requires careful configuration and coordination across multiple modules. [22]

A number of platforms and frameworks have been proposed for urban energy modeling and for energy community studies. However, several existing tools rely on highly customized and application-specific frameworks, often requiring high-level expertise in both energy and coding, which can restrict extensibility and adaptability. Recent literature highlights that co-simulation can integrate multiple specialized tools and that plug-and-play approaches can address these limitations by improving flexibility and supporting more realistic cross-domain interactions. [3, 15]

This thesis addresses the above gap from the perspective of *interaction and configuration*: how to make the composition of complex co-simulation scenarios more transparent, reproducible, and accessible, while maintaining the flexibility and scalability required at district scale. The core problem is that, for large and heterogeneous scenarios, manual configuration approaches (even when structured through templates) do not sufficiently support users in expressing model dependencies, binding models to many physical entities, validating connections, and generating a correct executable configuration.

## 1.2 Objectives and Contributions

The main objective of this work is to support urban-energy co-simulation users in defining, executing, and analyzing scenarios through a workflow that reduces configuration burden and makes data dependencies explicit.

To achieve this objective, this thesis presents the development of:

- **A web-based graphical frontend (Coesi UrbanSim Frontend)** that enables scenario composition through a node-based visual programming paradigm. Users can create projects and scenarios, select physical entities from a geospatial context, configure model parameters via UI forms, and connect typed input/output ports to define data flow.

- **A configuration-generation microservice (yaml_builder)** that compiles the high-level visual graph produced by the frontend into a low-level executable configuration (YAML). The service performs graph compilation, instance expansion for entity batches, connection resolution (including time-shifted links to handle feedback), and validation/type coercion.

The key contributions can be summarized as follows:

1. **Visual composition of co-simulation scenarios:** a node-link interface to represent heterogeneous simulation components and their data dependencies as an explicit directed topology.

2. **Geospatial entity binding:** mechanisms to bind selected real-world entities (e.g., sets of buildings) to simulation nodes, enabling scalable scenario definitions.

3. **Graph-to-YAML compilation pipeline:** a backend "compiler-like" service that converts the visual graph into a standardized executable configuration, supporting validation and scalability.

4. **Workflow transparency through configuration preview:** a human-readable preview/approval step of the generated configuration before execution, improving traceability and reducing late-stage failures.

This work is positioned within a broader co-simulation ecosystem where configuration files and automated composition have already been identified as key enablers (e.g., scenario builders relying on YAML schemas). [20]

## 1.3 Methodology Overview

The thesis follows a design-and-implementation methodology driven by the requirements of urban-energy co-simulation workflows:

- **Requirement identification:** functional and non-functional requirements were collected by analyzing the needs of urban-scale co-simulation (heterogeneous tools, scalable entity counts, reproducibility) and by comparing with established approaches that highlight configurability and modularity. [16, 24]
- **Frontend implementation:** the GUI was implemented as a web application providing (i) geospatial selection and inspection, and (ii) node-based scenario composition with typed connections and parameter editors.
- **Compilation service implementation:** the yaml_builder microservice was implemented to transform the frontend scenario representation into executable YAML, handling model instantiation, batch expansion, connection rules, and strict validation.
- **Demonstration via case study:** the workflow is demonstrated through a case study based on a neighbourhood dataset from San Salvario, Turin, leveraging district-scale inputs and producing outputs that can be inspected through the frontend.

Where relevant, the design is informed by architectural patterns that emphasize modularity and scalability when integrating heterogeneous data and simulation components (e.g., microservices approaches for co-simulation platforms). [22]

## 1.4 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2** introduces the background concepts required to understand urban-energy co-simulation, including model coupling, scenario configuration, and data integration challenges.
- **Chapter 3** reviews related work on urban energy modeling platforms, co-simulation frameworks, and approaches that address setup complexity and interoperability.

- **Chapter 4** defines the requirements and design rationale, including the user workflow and the principles guiding the interface and configuration process.
- **Chapter 5** provides a system overview and architecture, describing the platform components and the end-to-end scenario lifecycle from UI to execution and analysis.
- **Chapter 6** details the design and implementation of the Coesi UrbanSim Frontend.
- **Chapter 7** details the design and implementation of the yaml_builder compilation microservice.
- **Chapter 8** presents the case study and demonstration scenarios, including the workflow execution and result inspection.
- **Chapter 9** concludes the thesis and outlines limitations and future work.

# Chapter 2

# Background

This chapter introduces the foundational concepts required to understand the contributions of this thesis. It defines the terminology used in the Coesi UrbanSim Frontend workflow and provides background on (i) district-scale urban energy modeling, (ii) co-simulation as a coupling paradigm, (iii) the configuration and orchestration challenges that arise when scaling to many entities and heterogeneous models, and (iv) the role of geospatial entities and data layers in urban simulation workflows. Figure 2.1 provides a high-level overview of the conceptual workflow considered in this thesis, from heterogeneous inputs to scenario definition, orchestration, and results.


Figure 2.1. Conceptual workflow of a district-scale co-simulation study.

## 2.1 Urban Energy Modeling and District-Scale Simulation

Urban energy modeling aims at representing and analyzing energy-related processes within urban environments, often at the scale of neighborhoods or districts. At this scale, studies typically involve many heterogeneous assets (e.g., buildings with different uses and properties) and multiple energy technologies and carriers. The motivation for district-scale simulation is commonly scenario-based: analysts compare alternative configurations (e.g., baseline vs. increased PV adoption) to evaluate system-level impacts and key performance indicators. [3]

In this thesis, simulations are organized around a hierarchical structure:

- **Project**: a top-level container associated with a specific geographic location.
- **Scenario**: a concrete configuration within a project, characterized by temporal settings (e.g., start date, duration, step size) and a logical composition of connected simulation components.

At district scale, scenarios must integrate heterogeneous inputs, typically including building-related data (geometries and attributes), weather time-series data (e.g., outdoor temperature and solar irradiance), and—depending on the use case—additional contextual layers such as demographics or infrastructure representations. Prior platforms for urban energy analysis show that integrating such heterogeneous sources is a central requirement for realistic district simulations. [22] Urban-scale energy analysis tools are also frequently discussed under the broader umbrella of *digital twins* for the built environment. However, the term is used with substantial variability: a large-scale review of built-environment literature reports a sheer volume of definitions and that a terminological consensus is still out of reach, which can create ambiguities in conceptualization and implementation. Moreover, the same study observes that many built-environment digital twins are closer to *long-term decision support* systems than to strictly real-time, fully bi-directional cyber-physical systems; components such as simulation, AI/ML, and real-time capabilities are not consistently mature or present across implementations. These observations motivate the need to clearly specify the assumptions, components, and workflow boundaries of urban simulation platforms. [1]

## 2.2 Co-Simulation Concepts

Co-simulation is a methodology in which multiple simulators (often developed independently and representing different domains) exchange data during a coordinated execution over a common simulation clock (start time, step size, and duration). This approach enables a more comprehensive analysis of complex systems, where subsystem interactions affect overall performance. In smart-grid and multi-energy contexts, co-simulation supports the evaluation of strategies (e.g., flexibility or demand response) in the broader perspective of the entire system. [16]

A useful abstraction for co-simulation is a **directed dependency graph**:

- **Nodes** represent simulation components (e.g., a PV model, a building demand model, a grid solver) or data sources (e.g., weather time series).
- **Directed edges** represent data flow dependencies: an output produced by a source component is consumed as an input by a target component.

In the platform implementation, time-series inputs (e.g., weather) are provided through dedicated data-source models instantiated as nodes in the scenario graph.

This directed topology makes dependencies explicit and provides a clear basis for orchestration: at each synchronization point, simulators exchange the variables required by the declared connections. [20]

### Time and synchronization

Most co-simulation workflows proceed in discrete time steps. At each time step $t$, components compute outputs and exchange values according to the connection graph. A central practical issue is the presence of feedback loops, where two components depend on each other. If both dependencies are enforced at the same instant (i.e., $t \to t$ in both

directions), the system may form an algebraic loop. A common pragmatic solution is to allow **time-shifted coupling**, where some dependencies use values from the previous step (i.e., $t \rightarrow t + 1$) to break instantaneous cycles. This thesis adopts the distinction between:

- **Same Time** connections: synchronous exchange at the current step ($t$).
- **Next Time** connections: delayed exchange where the source output at step $t$ is used by the target at step $t + 1$.

The formal handling of these connection types is discussed later in the configuration-generation pipeline (Chapter 7).

## 2.3 Configuration and Orchestration Challenges

While co-simulation enables modular integration, it also introduces significant configuration and orchestration complexity. A runnable scenario must encode, at minimum:

- **Global scenario settings**: time horizon, start date, step size.
- **Simulator/model instances**: which components are instantiated and their parameters.
- **Connections**: the directed data-flow topology between component ports.
- **Outputs**: which variables are logged, stored, and visualized.

Many co-simulation platforms represent these elements through structured configuration files, such as YAML-based scenario descriptions that specify simulator instances, connections, and output selections. [20]

### Scaling and setup complexity

Scenario configuration tends to scale with the number of instantiated components and connections. In district studies, a single logical model may need to be replicated across hundreds or thousands of entities (e.g., one instance per building), which amplifies the amount of parameter specification and coupling definitions. As noted in the literature, this growth increases the need for manual intervention and makes configuration a practical bottleneck in large co-simulation deployments. [24]

### Data heterogeneity and consistency

Urban-scale scenarios additionally require consistent handling of heterogeneous datasets and representations (e.g., different formats and attribute conventions across sources). Even when simulation components are modular, maintaining consistent identifiers, units, and metadata across the pipeline remains a non-trivial requirement for reproducible and scalable studies. [15]

## 2.4 Geospatial Entities and Data Layers

District-scale simulation is inherently spatial: scenarios are anchored to a geographic region, and the modeled objects correspond to physical assets located in space. In this thesis, the term **entity** refers to a physical object that a model can operate on, with a unique identifier and a set of attributes (e.g., building height, floor count, footprint geometry). In practice, urban simulation workflows commonly integrate multiple geospatial layers, such as building footprints and associated attributes, and may include additional contextual layers (e.g., terrain models, census-based information, or infrastructure layers). [22]

A key mechanism for managing district scale is the concept of **entity grouping** (or batching): users often select a subset of entities (e.g., a neighborhood's buildings) and apply a given model to all entities in that set. This requires a robust mapping from the geospatial selection (IDs and attributes) to the logical simulation graph (nodes and edges). Data standardization and consistent representation of entities support this binding and reduce friction in the end-to-end workflow. [15]

### Summary

In summary, Chapter 2 introduced the core concepts that motivate and explain the design choices of this thesis: co-simulation as directed coupling of heterogeneous components, the configuration/orchestration burden that grows with scale, and the role of geospatial entities and data layers in district simulations. These foundations enable the detailed presentation of the platform architecture (Chapter 5) and the implementation of the frontend (Chapter 6) and configuration-generation service (Chapter 7).

# Chapter 3

# State of the Art

This chapter positions the thesis within recent research on (i) urban energy modelling platforms, (ii) co-simulation frameworks and workflow infrastructures, (iii) node-based and visual programming interfaces, and (iv) UX/HCI practices for scientific and engineering software. The discussion highlights limitations that motivate the contributions of this work, namely a node-based scenario editor coupled to a backend compilation service that generates executable co-simulation configurations.

## 3.1 Urban Energy Modeling Platforms

Urban Building Energy Modelling (UBEM) encompasses methods and tools that estimate energy demand and system performance for groups of buildings (neighbourhoods, districts, cities), typically to support scenario-based planning and decarbonization studies [26]. Contemporary UBEM literature emphasizes that the field has evolved from isolated building-level simulation toward end-to-end workflows that must (i) operate on heterogeneous building stocks, (ii) use incomplete and uncertain urban data, and (iii) provide outputs that are interpretable and actionable for planning, policy, and infrastructure decisions [15, 26]. These pressures are amplified when UBEM is used not only for annual energy balances, but also for technology adoption, retrofit prioritization, heating/cooling planning, flexibility assessment, and multi-energy interactions across electrical and thermal domains [19, 26].

### 3.1.1 UBEM scope and modelling paradigms

UBEM workflows span multiple modelling paradigms. At one end, detailed physics-based or high-fidelity simulation models can represent building envelope dynamics and HVAC system behaviour, enabling fine-grained assessment but demanding richer inputs and higher calibration effort. At the other end, reduced-order or archetype-based approaches trade physical detail for scalability, often relying on typological building libraries, statistical assumptions, and simplified control representations to simulate large building stocks under constrained data availability [18, 26]. In practice, many urban studies adopt hybrid pipelines (e.g., physics-inspired models with data-driven parameter inference), reflecting

the need to balance computational cost, transparency, and robustness to uncertain inputs. A recurring conclusion is that model choice is inseparable from data readiness: the achievable fidelity is often limited by the availability of geometry, building attributes, occupancy/use patterns, and energy-system information at urban scale [26].

Beyond technical modelling, UBEM is increasingly framed as an enabling layer for sustainable urban development. The recent state-of-the-art review on UBEM and Sustainable Development Goals (SDGs) stresses that UBEM can support a broad range of sustainability targets (e.g., resource efficiency, policy evaluation, and equity-aware interventions), but that translating modelling outputs into practice requires careful interpretation of trade-offs and uncertainties [26]. For example, interventions that reduce energy use may interact with comfort, overheating risk, or distributional impacts; similarly, socio-technical data can improve targeting of vulnerable groups but may raise privacy or governance concerns. These observations motivate UBEM workflows that provide transparent assumptions, uncertainty awareness, and reproducible scenario definition.

### 3.1.2 End-to-end UBEM workflows and the data-to-model pipeline

Most UBEM platforms operationalize a common pipeline: (i) urban geometry acquisition (footprints or 3D city models), (ii) semantic enrichment with building attributes (construction period, use type, height/floor count, envelope archetypes), (iii) synthesis of operational inputs (weather, schedules, internal gains, setpoints), (iv) simulation/execution, and (v) post-processing for aggregated indicators and maps. While conceptually straightforward, the pipeline is a major practical bottleneck because urban datasets are typically sparse, inconsistent, and collected for non-simulation purposes. As a result, "data cleaning" and enrichment become a substantial part of UBEM effort, and the resulting modelling choices can strongly influence reproducibility and validity.

This issue is made explicit by Rehmann et al., who position UBEM as a critical tool for local heating and cooling planning and highlight that reproducibility and reliability remain limited due to data scarcity and workflow complexity [18]. Their Berlin case study proposes a structured approach to (i) acquire heterogeneous geometry sources, (ii) enrich buildings via archetypes and district information, and (iii) evaluate the pipeline through sensitivity and validation steps. Importantly, the study treats the data pipeline itself as an object of evaluation: instead of assuming that input preparation is a neutral preprocessing step, it is shown to be a dominant driver of model outcomes and a source of uncertainty that must be explicitly managed [18]. For urban-energy planning contexts (where decisions must often be made under incomplete knowledge), this reinforces the need for workflows that document assumptions, track provenance, and support iterative refinement as new data becomes available.

Figure 3.1 summarizes this end-to-end workflow and highlights the data-to-model preparation stage.

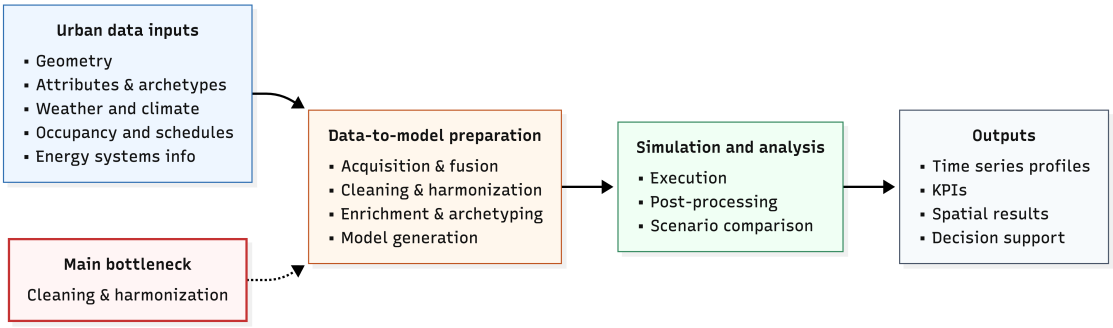


Figure 3.1. End-to-end UBEM workflow from urban data inputs to simulation outputs.

### 3.1.3 Tool diversity and platform-dependent results

A second recurring theme is that UBEM results and usability depend strongly on platform choice. Different tools require different input schemas, enforce different internal assumptions, and provide different degrees of transparency regarding intermediate artefacts (e.g., how archetypes are assigned, how schedules are constructed, or how HVAC systems are represented). Comparative studies therefore serve two roles: they help users select tools aligned with available data and required outputs, and they reveal where "tool interchangeability" breaks down.

Nardelli et al. provide a detailed comparative analysis of two widely used UBEM tools—IES-iCD (commercial) and umi (originating from academic development)—by covering the full workflow from model creation and input assignment to simulation, visualization, and export. A key conclusion is that UBEM tools still lack unified standards across input/output definitions, nomenclature, and calculation approaches, and that substantial differences can appear in results and workflow complexity depending on modeller expertise and platform defaults [10]. This observation is particularly relevant for scenario-driven urban studies, where analysts frequently need to compare alternatives across many buildings: if each tool encodes different hidden assumptions or requires different manual interventions, the barrier to reproducible scenario comparison increases. Consequently, the literature suggests that tool comparison is not only a benchmarking exercise, but also an argument for workflow standardization and for interfaces that reduce "configuration ambiguity."

### 3.1.4 Standardization and urban data representations

Because UBEM depends on integrating geometry with energy-relevant attributes, standardization efforts in urban data representation have become central to scaling and interoperability. City models and built-environment standards aim to reduce friction in the data-to-model step by providing consistent identifiers, geometries, and attribute

schemas that can be shared across tools and workflows. In this direction, CityJSON-based pipelines have been proposed to bridge sparse urban datasets and simulation inputs. For example, Rando Mazzarino et al. present an automated tool that converts sparse datasets into CityJSON, supporting the creation of urban energy models from heterogeneous sources [15]. Such approaches matter not only for data preparation, but also for linking geospatial "entities" to simulation components: consistent entity identifiers and attribute schemas support scalable instantiation (e.g., "apply this building model to all buildings in a selected district") and reduce coupling errors when multiple models consume or produce building-level variables.

In district-scale settings, standardization becomes a prerequisite for multi-model studies: UBEM outputs (demands, profiles, KPIs) are often used as inputs to other analyses (e.g., grid impact, district heating studies, energy community optimization). Without shared semantics, units, and naming, integrating these workflows is brittle. This also motivates the separation between *high-level scenario composition* (what models are connected and to which entities) and *low-level executable configuration* (how each instance is parameterized and wired), since the latter is precisely where schema mismatches and ambiguous conventions typically surface. [6]

### 3.1.5 Scaling from building stocks to interconnected urban energy systems

Finally, UBEM is increasingly embedded in broader urban energy system modelling, where buildings interact with distribution networks, district heating and cooling, transport electrification, and storage/control layers. A recent critical review on modelling interconnected energy systems across urban scales highlights the need for integrated multi-scale modelling frameworks and notes that many studies still focus on isolated scales—especially the building level—without capturing district-level interdependencies and end-user behavioural effects. The same review emphasizes a shift toward end-user-centric modelling and more explicit representation of how human behaviour (appliance use, comfort preferences, EV charging) influences demand patterns and flexibility at district scale [19]. These requirements often exceed what a single monolithic UBEM tool can reasonably represent, and instead motivate modular workflows where specialized models are coupled in a controlled experiment definition.

From a workflow perspective, this literature indicates that "urban energy modelling platforms" must support more than running a single simulation engine: they must provide mechanisms for (i) consistent entity-centric data management, (ii) scalable scenario definition across many instances, and (iii) interoperable exchange of variables across heterogeneous submodels. These requirements align with the motivation of this thesis: reducing the scenario-definition burden and making dependencies explicit through a node-based representation, while generating strict executable configurations for co-simulation.

Overall, the UBEM literature shows strong methodological breadth, but persistent workflow limitations. Data readiness, tool-dependent assumptions, limited standardization, and multi-scale coupling needs remain key barriers to reproducible and scalable scenario analysis. This motivates platform concepts that treat scenario composition and

configuration generation as first-class concerns, rather than purely as manual preprocessing steps.

## 3.2 Co-Simulation Frameworks and Workflow Tools

Co-simulation frameworks address the integration of heterogeneous simulators by coordinating initialization and execution, managing data exchange, and regulating/synchronizing time progression via a master algorithm (co-simulation orchestrator) that advances a common simulation time while handling different simulator time-step durations [3,20,24]. In energy-system applications, this enables analysts to couple models that are developed independently (e.g., building loads, PV generation, thermal networks, storage, control logic) while preserving the domain-specific strengths of each tool [20, 24]. In the renewable-energy-community context, recent work presents co-simulation infrastructures that emphasize modularity and plug-in integration of specialized components under a common experiment definition, allowing scenario-based comparison of alternative community designs and operating assumptions [3]. Such infrastructures are particularly relevant at district scale, where a single study may involve many entities (e.g., buildings) and multiple energy carriers, and where system-level performance depends on cross-domain interactions rather than isolated subsystem behaviour.

**From domain models to reusable co-simulation platforms.** A recurring theme in recent energy co-simulation research is the move from case-specific scripts to platform-like architectures that support reuse, declarative scenario definition, and comparative studies [21]. For instance, modular platforms have been proposed to compare demand-side flexibility solutions in district heating (DH) under variable operating conditions, explicitly positioning the co-simulation layer as the integration point for multiple interacting subsystem models. In this setting, the platform perspective matters because the research goal is not to run a single configuration once, but to benchmark alternative flexibility strategies under consistent assumptions and comparable performance indicators, enabling systematic scenario exploration [14]. Similar motivations appear in renewable-energy-community analysis, where infrastructures are built around explicit scenario design and KPI/economic evaluation across multiple community configurations [3].

Beyond electric-thermal coupling, district energy systems (DES) introduce further complexity when advanced control or optimization methods must be integrated with detailed physical simulators. Hybrid co-simulation approaches illustrate this need: for example, TRNSYS-based DES models can be coupled with Python components to implement optimization and data-driven controllers that are cumbersome to develop directly inside the monolithic simulator environment [5]. These studies highlight an important practical point: co-simulation is often adopted not only to couple different *physical* domains, but also to connect simulation engines with external optimization, AI/ML, or supervisory control modules [5, 14].

**Scalability and operationalization: from experiments to distributed execution.** As co-simulation studies scale from a handful of components to large integrated scenarios, operational concerns become first-order requirements. Recent contributions explicitly focus on deployment and streamlined operation of distributed co-simulation platforms, targeting reproducibility and scalable execution by adopting microservice-based architectures and containerized deployment practices. In these approaches, packaging simulators and supporting services as containers can improve portability across environments and enable horizontal scaling, while an orchestrated deployment model helps manage the lifecycle of multiple interconnected services (e.g., configuration services, data services, simulation runtimes). However, the same line of work also points out that—despite improvements in runtime scalability—users still face friction at the *scenario-definition* stage, motivating tool support (including graphical tools) to simplify scenario specification and reduce manual effort [21].

**Configuration as a bottleneck: scenario descriptions, templates, and workflow tooling.** Most co-simulation platforms ultimately require a machine-readable scenario description that encodes (i) global settings (time horizon, step size), (ii) simulator/model instances and parameters, (iii) the data-flow topology (connections), and (iv) output selections. In practice, these descriptions are often expressed through structured configuration formats (commonly YAML/JSON) or platform-specific schemas. While templates and reusable patterns can reduce duplication, configuration complexity tends to grow with the number of instantiated components and connections, especially when a logical model must be replicated over many entities (e.g., one instance per building). This makes scenario configuration a practical bottleneck in large deployments and increases the likelihood of errors in port mapping, parameter specification, and output definitions.

Workflow-oriented research outside the energy domain reinforces why this is difficult: experimental simulation campaigns often require repeatedly assembling, modifying, and executing pipelines while preserving reproducibility. Workflow frameworks therefore emphasize explicit pipeline representations, dependency management, and support for iteration/loops and systematic experimentation [23]. Although the target domains may differ, the underlying issue is shared with energy co-simulation: users benefit when the experiment definition is represented in a structured, inspectable form that supports reuse, validation, and automation.

Figure 3.2 places this configuration burden in context by summarizing the typical layers of co-simulation platforms and the role of structured scenario descriptions in driving orchestration and execution.

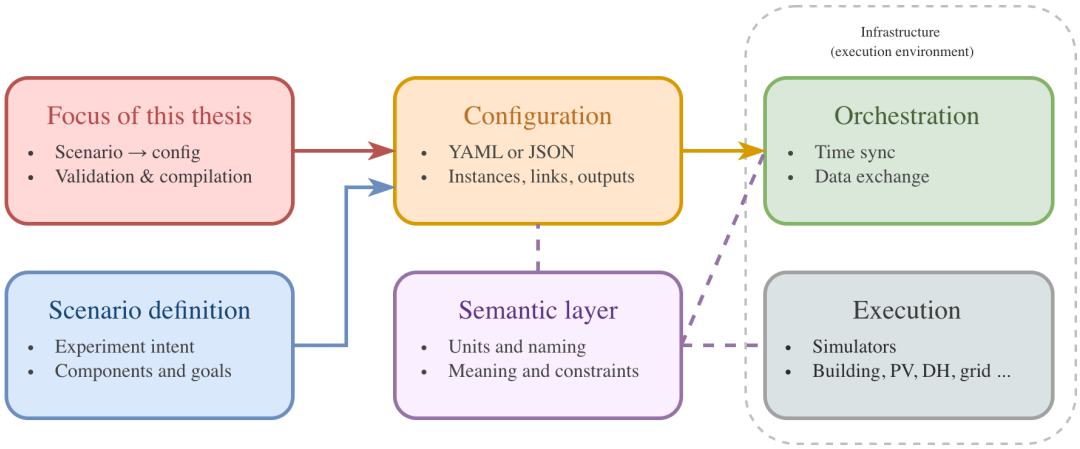


Figure 3.2. Layered view of a co-simulation platform workflow, from scenario definition to orchestrated simulator execution, with cross-cutting semantic constraints.

**Semantic interoperability: beyond "making tools talk."** In addition to runtime scalability, *semantic* scalability is a concern: different simulators may use different naming conventions, units, sign conventions, reference frames, or implicit assumptions for exchanged variables. Work on semantic interoperability formalizes use cases and requirements showing that consistent semantics and metadata are necessary to avoid brittle or ambiguous couplings, particularly when models are reused across contexts or combined in new ways [6]. This literature clarifies that interface standards alone are not sufficient if they do not capture the meaning and constraints of exchanged quantities. As a result, semantic annotations, metadata schemas, and mapping/validation mechanisms become important enablers for robust model coupling [6].

**Assisted setup and configuration generation.** Complementary approaches aim to reduce setup complexity by assisting users in producing correct scenario descriptions. Ontology-based methodologies, for example, provide structured representations of models, ports, constraints, and compatibility relations, and can support users by guiding configuration choices and reducing the risk of invalid couplings [24]. In parallel, deployment-focused platform work motivates higher-level tooling—potentially including visual editors— to simplify scenario definition while preserving the advantages of structured executable configuration files [21]. These directions align with the compiler-like role adopted in this thesis: separating high-level scenario composition (user-facing graph of components and dependencies) from low-level executable configuration generation, while adding validation and type/compatibility checks to reduce build-fail cycles.

## 3.3 Node-Based and Visual Programming Interfaces

Visual and node-based programming paradigms have been widely explored to simplify the creation of complex pipelines by making dependencies explicit and reducing error-prone manual configuration. In general, these approaches represent a system as a directed graph where nodes encapsulate functions or components and edges encode the flow of data, control, or events. This representation is particularly suited to integration-heavy tasks, because it externalizes structure (what depends on what) and enables both user-facing reasoning (inspection, debugging) and machine-facing processing (validation, transformation, code/configuration generation).

A prominent strand of work comes from the digital-twin literature, where web-based visual programming environments have been proposed to support both design-time composition and operational workflows. Van Emmerik and Waardenburg propose an integrated web-based environment that combines 3D modelling, data visualization, monitoring, and control within a single interface [25]. A key feature is a Scratch-like block workspace implemented with Blockly, providing standard control structures (conditionals, loops, variables) alongside domain-specific blocks for geometric construction and operations (e.g., primitives, transformations, and Boolean operations). Beyond expressiveness, an important usability advantage is that block composition prevents syntactic invalidity: the editor constrains how blocks can be combined into a valid script, reducing a common source of user error in text-based scripting [25]. The same work also emphasizes operational capabilities: visual rules can be defined for monitoring and control using Blockly and JavaScript, supporting a consistent end-user environment across modelling and automation tasks [25].

Related concepts appear in other engineering domains, illustrating both the breadth and the maturity of visual programming as an interaction paradigm. In smart manufacturing and robotics, Niermann proposes a software framework that combines a digital twin with a graphical visual programming language to enable intuitive process creation involving multiple robotic systems [12]. The framework uses a node-based representation with explicit connection logic and supports user-centric process design, motivated by the need to make programming and orchestration accessible to non-expert users in industrial environments [12]. This cross-domain evidence is valuable for urban energy simulation: it suggests that node-based composition can act as a "common language" for expressing interactions among heterogeneous components, even when underlying execution involves complex distributed systems.

Beyond explicitly visual environments, graph-based workflow representations are also central in scientific computing and simulation tooling. The `mmodel` framework, for example, adopts a graph-theory representation of experimental simulation components to improve modularity and reduce duplication in simulation code; importantly, it enables systematic workflow modifications (e.g., adding loops for parameter exploration) without rewriting the original procedural implementation [23]. This demonstrates a broader principle relevant to this thesis: graph representations are not only helpful for UI-level composition, but also enable *compiler-like* transformations that generate or rewrite executable artefacts.

Nevertheless, for urban energy co-simulation, the challenge is not only *visual composition* but also *scalable instantiation* and *executable configuration generation*. In district-scale settings, a visually simple graph may expand to hundreds or thousands of model instances when bound to geospatial entities (e.g., "apply this building model to all buildings in a neighborhood"), and the resulting configuration must remain correct, reproducible, and compatible with strict co-simulation schemas. Moreover, co-simulation graphs frequently carry domain semantics (units, temporal meaning, sign conventions) that must be validated to avoid silent coupling errors. This thesis builds on visual composition concepts while focusing specifically on the graph-to-executable compilation step required by distributed co-simulation engines: the node editor provides a high-level, user-facing graph, while a backend service expands, validates, and compiles that graph into standardized runnable configurations.

Figure 3.3 summarizes this gap by showing how a user-facing scenario graph must be expanded, validated, and compiled into a strict executable co-simulation configuration before it can be run on a distributed runtime.

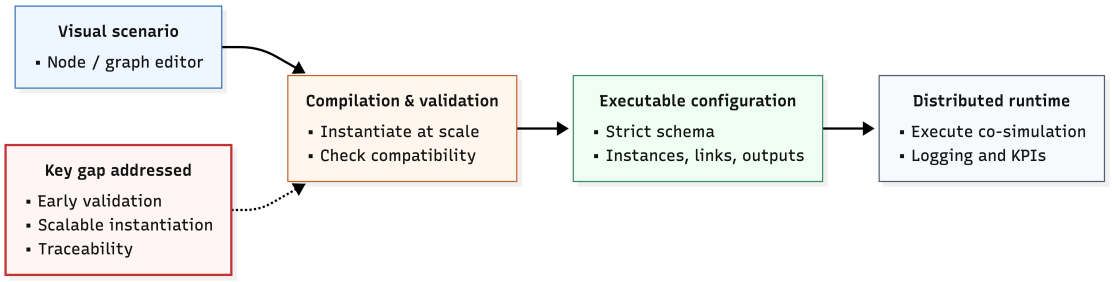


Figure 3.3. From visual scenario composition to executable co-simulation.

## 3.4 UX/HCI for Scientific and Engineering Software

Usability and user-centered design (UCD) have long been recognized as important for scientific software, where adoption and correct usage depend not only on algorithms but also on interface quality and workflow fit. Classic work in IEEE Software argues that scientific software development benefits from explicit UCD practices, especially because tools are often built by small teams under domain-driven constraints, and users are commonly domain experts rather than software specialists [9]. From this perspective, usability work is not merely "polish": it can reduce user errors, increase learnability, and improve the effectiveness and trustworthiness of results by aligning interfaces with real user tasks and mental models [9].

More recent engineering-software work provides concrete examples of UCD-style evaluation in complex technical systems. In industrial digital-twin and visual-programming environments, usability is commonly operationalized through structured user studies (e.g., questionnaires, task-based evaluation), acknowledging that the success of advanced

automation depends on whether operators can correctly configure and supervise processes [12]. These findings generalize naturally to co-simulation: configuration errors, misinterpreted variables, or hidden assumptions can invalidate results, and the cost of failure increases with scenario size (e.g., long runtimes, distributed deployments, and complex debugging).

In the context of co-simulation, these findings support the need for interfaces that (i) make system dependencies visible, (ii) reduce configuration burden through guided interaction and reusable abstractions, and (iii) provide early validation to prevent costly build-fail cycles. These principles inform the design rationale of the Coesi UrbanSim Frontend and the supporting configuration-generation service presented in the next chapters.

## Summary

The reviewed literature shows strong progress in co-simulation infrastructures, distributed execution, and visual workflow paradigms. However, recurring gaps remain in the usability and scalability of scenario setup: users must still couple heterogeneous models, bind them to many entities, and produce strict executable configurations, while also managing deployment and semantic correctness. This thesis addresses these gaps through a node-based scenario editor and a backend compilation pipeline that generates standardized, validated co-simulation configurations for scalable execution.

# Chapter 4

# Requirements and Design Rationale

This chapter defines the requirements that guided the development of the *Coesi UrbanSim Frontend* and motivates the main design choices. In the context introduced in Chapters 1–3, the core difficulty is not only executing a co-simulation, but enabling users to *compose, configure, and validate* heterogeneous models and their couplings in a way that is transparent and repeatable. Accordingly, the frontend is designed around a node-based scenario editor and supporting views (e.g., entity selection and simulation settings) that reduce configuration errors and make experiment definitions inspectable and reusable. The requirements and UI decisions are informed by established usability principles (e.g., consistency, visibility of system status, and error prevention) and by recommendations from research-software and scientific-workflow literature that emphasize usability, reproducibility, and interoperability as prerequisites for adoption and credible results. [7, 9, 11, 13]

## 4.1 Target Users and Use Cases

### 4.1.1 Stakeholder identification

In research-software projects, requirements are shaped by heterogeneous stakeholders with different levels of domain knowledge and technical expertise. To structure early requirements elicitation, a lightweight stakeholder mapping was performed following common stakeholder-analysis practice: identify relevant stakeholder groups, position them by interest and influence, and define appropriate engagement strategies (e.g., inform, consult, collaborate). [4, 17]

In this project, stakeholder identification was guided by the intended interaction workflow supported by the frontend (model selection from a catalog, node-based composition and connection semantics, entity batching from geospatial data, configuration export, and guided execution). Usability considerations are particularly relevant in scientific software where users are often domain experts but not software experts, and where configuration errors can invalidate results. [9, 13]

Stakeholders are defined here as roles that either directly use the editor or influence its requirements through model provisioning, execution services, and operational constraints; the categories do not imply separate user interfaces, but different goals and responsibilities interacting with the same platform.

Based on these considerations, stakeholders were grouped into four main categories:

- **Simulation analysts and energy researchers (primary users).** Users who define scenarios, select and group district entities (e.g., buildings), compose co-simulation workflows in the node editor, configure parameters and outputs, run experiments, and compare results.
- **Model developers (secondary users).** Users who contribute simulation components and expose ports, parameters, and metadata so the frontend can generate configuration views and validate model couplings (e.g., type/unit compatibility) before execution.
- **Decision-support users (secondary/tertiary users).** Users such as municipal energy officers or consultants who need to explore prepared scenarios and interpret outputs, prioritizing transparency of assumptions and clarity of results over low-level configuration.
- **Platform maintainers (internal stakeholders).** Users responsible for deployment, API availability, model registries, execution services, and operational reliability (e.g., monitoring run failures and maintaining compatibility across model versions).

These stakeholder groups inform the use cases in the next subsection: the primary flow targets analysts composing, validating, and executing scenarios in the workspace editor, while secondary considerations address model onboarding constraints and operational requirements such as reliability, error visibility, and reproducibility through inspectable exported configurations.

### 4.1.2 Representative use cases

The following representative use cases were derived from the stakeholder groups above and from the interaction workflow implemented in the current workspace editor. They are described at a level that can be traced to concrete UI features and backend interactions in Chapters 5–7 (model catalog discovery, graph editing, validation services, configuration generation, and execution monitoring).

- **UC1 — Create and manage entity batches for a scenario.** Select buildings or other district entities through the entity workflow (attribute filters, random split, or map selection), group them into named batches, and review summary statistics and identifiers to confirm the selected dataset before linking entities to models.
- **UC2 — Instantiate models and compose a co-simulation graph.** Browse a catalog of available models, instantiate components via drag-and-drop onto the

canvas, and connect outputs to inputs using a node-link representation of the scenario.

- **UC3 — Define coupling semantics and validate model connections.** Create connections between ports and specify temporal dependency semantics (e.g., *Same Time* vs. *Next Time*) for each edge. Connections are validated (e.g., data type and unit compatibility), and invalid couplings are blocked before execution.

- **UC4 — Configure model parameters and simulation settings.** Configure node parameters and computation settings via the parameter editor, and specify global simulation settings such as time horizon and output selection. Where supported by the underlying model metadata, the UI provides defaults and basic constraints to reduce configuration errors.

- **UC5 — Generate, inspect, and execute an experiment configuration.** Compile the visual scenario into a machine-readable configuration (YAML), preview and download it for inspection and reproducibility, submit the run to the backend, and monitor progress via a step-based execution view that surfaces failures and completion status.

- **UC6 — Browse scenario/run history and view available outputs.** After execution, access an inspection view to list scenarios and runs, and open available output artifacts for completed executions.

Figure 4.1 summarizes the end-to-end interaction workflow captured by the representative use cases, from entity selection and scenario composition to configuration generation, execution monitoring, and results inspection.

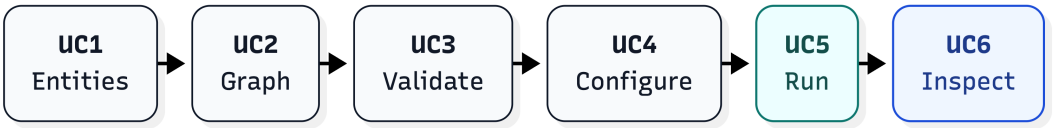


Figure 4.1. Representative end-to-end workflow supported by the frontend (UC1–UC6).

## 4.2 Functional Requirements

Functional requirements (FR) were derived by translating the use cases into system capabilities. Requirements are organized by feature area and expressed as *shall* statements. Priority is indicated using a simple MoSCoW classification (M: must, S: should, C: could).

Table 4.1. Functional requirements for the Coesi UrbanSim Frontend.

| ID | Requirement | Pr. |
|---|---|---|
| FR-1 | The system shall allow users to create, load, and delete scenarios within a project workspace. | M |
| FR-2 | The system shall provide a map-based view to select and inspect spatial entities (e.g., buildings) and bind them to scenario elements. | M |
| FR-3 | The system shall provide a node-based editor to compose a scenario as a directed graph of components connected via typed ports. | M |
| FR-4 | The system shall enforce connection rules (e.g., compatible data types, directionality, timing semantics) and prevent invalid couplings. | M |
| FR-5 | The system shall expose node configuration through forms generated from model metadata (schema-driven UI), including defaults, constraints, and help text where available. | M |
| FR-6 | The system shall validate scenarios incrementally (as users edit) and provide actionable error messages for missing parameters, invalid connections, and inconsistent units/ranges. | M |
| FR-7 | The system shall generate an executable experiment definition (YAML) from the scenario graph and associated parameters. | M |
| FR-8 | The system shall allow users to preview and download the generated YAML before execution, enabling reproducibility and review. | S |
| FR-9 | The system shall submit an execution request to the backend and display execution status (queued/running/failed/completed) with logs or error details when available. | M |
| FR-10 | The system should support scenario templates (starter graphs) and parameter presets to reduce effort for common study types. | S |
| FR-11 | The system could support batch runs (parameter sweeps) and structured comparison of KPIs across runs. | C |

## 4.3 Non-Functional Requirements

Non-functional requirements (NFR) capture quality attributes that are particularly important for research software: usability, reproducibility, robustness, and maintainability. They also reflect that users may not be software experts and that errors in experiment definition can invalidate results.

- **Usability and learnability.** The interface shall support efficient learning and reduce cognitive load for multi-step tasks such as scenario creation and graph composition. Design choices shall be guided by established usability heuristics (e.g., visibility of system status, consistency, error prevention, recognition over recall). [11]

- **Reproducibility and transparency.** The system shall enable users to export and reconstruct a complete experiment configuration (including model identifiers/versions and parameter values) so results can be repeated, inspected, and reviewed. [2, 7]

- **Performance and responsiveness.** Common UI interactions (entity selection,

graph editing, incremental validation) shall remain responsive for realistic district-scale scenarios. Long-running operations (e.g., configuration compilation and execution submission) shall run asynchronously and provide progress feedback.

- **Robustness and error handling.** The system shall fail safely: invalid scenario definitions shall be blocked before execution, and backend failures shall be reported with actionable information (e.g., error messages, logs when available) and recovery options (e.g., retry or return to edit).

- **Extensibility and maintainability.** New models shall be integrable without rewriting the UI by relying on shared metadata/schemas and consistent component interfaces.

- **Portability.** The frontend shall run in standard modern browsers and support typical research workflows (desktop-first, large screens), while remaining usable on smaller screens where feasible.

## 4.4 UI/UX Design Process and Principles

This section describes how the frontend UI/UX was designed and refined, and which principles guided key decisions in the workspace editor. The goal is to ensure that scenario configuration and execution monitoring are understandable, error-resistant, and reproducible, despite the heterogeneity of models and the complexity of co-simulation workflows.

### 4.4.1 User-centred design process

UI/UX design in this thesis followed a user-centred, iterative workflow consistent with ISO 9241-210, which frames interactive-system design as a cycle of (i) understanding the context of use, (ii) specifying user requirements, (iii) producing design solutions, and (iv) evaluating designs against requirements. [8] In a research-software setting, access to end users is often limited and the user community is heterogeneous; therefore, formative feedback was primarily gathered through supervisor/maintainer review and targeted walkthroughs, an approach commonly recommended for improving usability and adoption in scientific software. [2, 9]

Concretely, the process in this work comprised three iterations: (1) an initial iteration in which requirements were identified and the representative use cases (UC1–UC6) were defined; (2) a prototype-oriented iteration focusing on the workspace editor (entity workflow, node graph editing, and schema-driven parameter configuration); and (3) an implementation-focused iteration in which validation feedback, configuration export (YAML), and execution monitoring were refined to reduce setup errors and improve transparency. Throughout these iterations, design changes were recorded as issues (usability defects, missing feedback, unclear terminology) and prioritized based on frequency and impact on the primary task flow (scenario definition and execution).

### 4.4.2 Design principles applied to the frontend

Design decisions were guided by usability heuristics and by the specific risks of this domain. [11] The principles below summarize the rationale and how they are operationalized in the frontend:

- **Visibility of system status.** The UI exposes scenario validity and validation state, and makes execution status observable (queued/running/failed/completed), with progress indicators for long-running operations. [11]
- **Error prevention through constraints and typed composition.** Invalid couplings are prevented by typed ports and connection rules (direction, data type, unit compatibility, and timing semantics), and parameter editors enforce constraints and defaults where metadata is available. [11]
- **Actionable feedback and recovery.** When validation fails, the UI provides localized, actionable messages (what is missing, where the problem is, and what to change), and supports recovery by returning users to the relevant node/edge/parameter without losing work.
- **Match to the domain language and units.** Labels and descriptions follow energy-domain terminology (e.g., PV, storage, time step), and numerical inputs display expected units, ranges, and meanings near the control to reduce ambiguity during configuration.
- **Recognition over recall.** The model catalog is searchable and the node inspector keeps relevant parameters and metadata visible in context, minimizing hidden state and reducing the need for users to remember configuration details across views. [11]
- **Progressive disclosure.** The interface emphasizes a minimal "happy path" for common tasks, while advanced configuration options are grouped and expandable, limiting cognitive load for first-time users.
- **Consistency across workflow steps.** Layout, interaction patterns, and terminology are kept consistent across entity selection, graph composition, configuration, and execution monitoring to reduce mode switching and training overhead. [11]

### 4.4.3 Evaluation approach

Given the limited timeline and the research-software context, evaluation in this thesis was formative and lightweight. Feedback was collected through supervisor/maintainer review sessions and practical walkthroughs of the core workflow (UC1–UC5), and issues were recorded and prioritized to guide iterative improvements.

In practice, evaluation relied on: (i) informal heuristic checks guided by Nielsen's usability heuristics (e.g., visibility of system status and error prevention); (ii) short task-based walkthroughs with lab members (e.g., composing a small scenario graph, fixing an invalid connection, exporting YAML, and submitting a run); and (iii) issue logging to track UX defects and prioritize fixes that block execution or can lead to invalid experiment definitions. [11]

# Chapter 5

# System Overview and Architecture

This chapter provides a system-level view of the Coesi UrbanSim Frontend workflow and clarifies how the two components developed in this thesis—the **Coesi UrbanSim Frontend** and the `yaml_builder` microservice—fit into the broader co-simulation platform. The remaining backend services (model registry, semantic validation, scenario storage, execution runtime, and results serving) are treated as *external platform dependencies* accessed through stable REST APIs. Their internal implementation is outside the scope of this thesis; they are described only to define interfaces, responsibilities, and end-to-end data flow.

## 5.1 Platform Overview

### Deployment model and boundary

The platform is deployed via **Docker Compose** and can be run in two modes: (i) a *development* profile where core services run in containers while the frontend runs natively (Vite on the host), and (ii) a *full* profile where the frontend is also served as a containerized static build (Nginx). In both cases, all services communicate over an internal Docker bridge network (`coesi_net`), while an **Nginx API Gateway** provides a single external entry point (port 80) and routes requests to internal service DNS names.

From the thesis perspective, the system boundary is:

- **Implemented in this thesis:** Coesi UrbanSim Frontend (web UI) and YAML builder (Scenario JSON → YAML compilation).
- **External dependencies (called via API):** Model Manager + GraphDB (model metadata), Validation Engine (connection checks), Scenario Manager (file persistence), Simulation Engine (execution + results endpoints), API Gateway (routing).

### Component roles

At a high level, the platform separates *interaction and experiment definition* (frontend) from *compilation* (YAML builder) and *execution services* (backend). The resulting architecture supports modular evolution: the UI can evolve independently from the execution engine, while the YAML schema and compilation logic are isolated in a dedicated service.

**API Gateway (routing and composition).** The API Gateway acts as a reverse proxy and provides a single external entry point for the platform. Incoming requests are routed to the corresponding internal services based on URL prefixes, allowing the frontend to rely on stable public paths while internal container ports and service addresses remain hidden. The gateway exposes the following public prefixes:

```
/api/v1/models
/api/v1/validate
/api/v1/scenarios
/api/v1/simulations
/yaml-builder
/urbansim
```

**Knowledge graph and model registry.** Model metadata (ports, parameters, semantic annotations) is stored in a triple store (GraphDB) and exposed via the Model Manager API. The frontend consumes this metadata to populate a searchable model catalog and to generate schema-driven parameter editors.

**Scenario storage and execution.** Scenario definitions are persisted as YAML files through the Scenario Manager. The Simulation Engine executes scenarios (triggered by scenario name) and produces results as HDF5 artifacts on the server side. For visualization, results are exposed through API endpoints that return *JSON slices* of HDF5 datasets (rather than reading HDF5 in the browser).

**High-level component architecture.** Figure 5.1 summarizes the platform at deployment level. The Coesi UrbanSim Frontend and the `yaml_builder` service (developed in this thesis) are highlighted, while the remaining services are shown as external platform dependencies accessed through REST APIs via the gateway. Persistent volumes are included to emphasize where durable state is stored (knowledge graph data, scenario files, and simulation outputs).

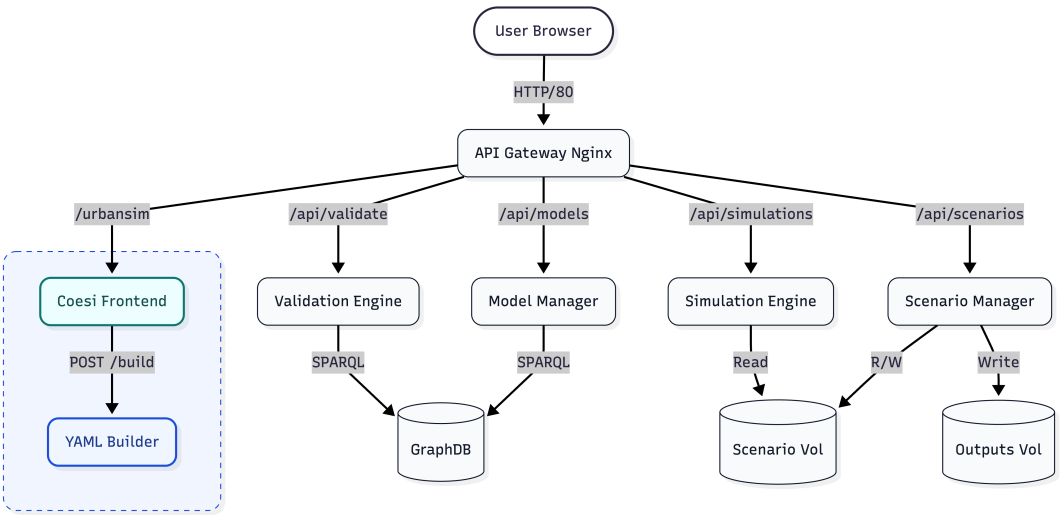


Figure 5.1. High-level platform architecture and system boundary.

## 5.2 Frontend Responsibilities and Boundaries

The Coesi UrbanSim Frontend is responsible for *interactive scenario authoring* and for orchestrating the user-facing workflow from graph composition to execution monitoring. Its responsibilities are intentionally limited to avoid duplicating backend logic and to ensure that the browser remains a thin client with a clear contract.

### Responsibilities (what runs in the browser)

**Graph editing and scenario state.** The frontend provides a node-based editor where users add model nodes, connect typed ports, and configure parameters. The graph state (nodes, edges, selection, and editing events) is managed client-side to enable immediate feedback (dragging, snapping, connecting, etc.).

**Pre-submission preparation and normalization.** Before submission, the frontend exports a *Scenario JSON* representation of the current workspace. This export step includes:

- flattening composite blocks into atomic nodes (when present),
- normalizing edge representations (including timing semantics such as “Same Time” vs. “Next Time” connections),
- collecting scenario settings (time horizon and date) and output selections,
- packaging entity-to-model assignments derived from the geospatial workflow.

**Client-side orchestration of the lifecycle.** The frontend coordinates a multi-step workflow (validate → export → compile → review → persist → execute → monitor → view results), issuing API calls and updating the UI based on responses. This design emphasizes transparency: users can inspect the generated YAML before committing it to execution.

### Boundaries (what the frontend delegates)

**YAML generation (via `yaml_builder`).** The frontend does not embed YAML schema logic directly in the client. Instead, it exports a normalized Scenario JSON representation and invokes the dedicated `yaml_builder` service—developed in this thesis—to compile Scenario JSON into the executable YAML configuration (Chapter 7).

**Semantic validation and model knowledge.** Connection validation against ontology/type rules is delegated to the external Validation Engine and Model Manager services. The frontend treats these as black-box validators and renders actionable feedback based on their responses.

**Scenario persistence and execution.** The frontend does not directly write scenario files or outputs. YAML persistence is handled by the Scenario Manager, and execution is handled by the Simulation Engine. The frontend only triggers runs, polls status, and requests result slices for visualization.

### Explicit non-goals

To keep responsibilities clean and to avoid duplicating platform logic, the frontend does *not*:

- execute simulations or run solvers,
- access GraphDB directly,
- parse binary HDF5 locally (it requests JSON slices),
- provide full authentication/authorization guarantees (authentication is treated as development/scaffolded in the current deployment profile).

**Responsibility boundary.** Figure 5.2 summarizes the division of responsibilities in the proposed workflow. The Coesi UrbanSim Frontend implements the interactive authoring environment and orchestrates the user-facing process, including graph editing, entity binding, scenario export, and results inspection. Tasks that require shared platform knowledge or durable state are delegated to external services via REST APIs , such as semantic connection validation, scenario persistence, execution, and access to model metadata. Configuration compilation is handled by the dedicated `yaml_builder` service, which translates the exported Scenario JSON into the YAML format consumed by the runtime.

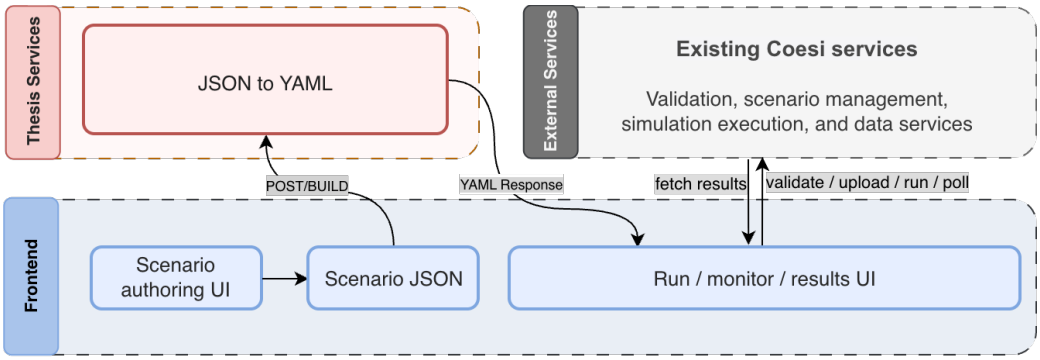


Figure 5.2. Responsibility boundary of the Coesi UrbanSim Frontend.

## 5.3 Data Flow and Scenario Lifecycle

This section describes the end-to-end lifecycle of a scenario, from interactive editing to results inspection. The primary artifacts exchanged across components are: (i) **Scenario JSON** (frontend export contract), (ii) **YAML** (engine-executable configuration), and (iii) **results datasets** (HDF5 stored server-side, exposed as JSON slices).

### Step 1: Authoring (UI state)

Users compose a scenario by: (1) selecting entities (e.g., buildings) and defining batches, (2) instantiating models from the catalog, (3) connecting ports to define directed data-flow, (4) setting global scenario parameters (start, stop, step), (5) selecting outputs for logging and comparison.

### Step 2: Export (Scenario JSON)

When the user triggers build/preview, the frontend exports the workspace into Scenario JSON. This contract contains:

- `nodes`: the flattened node list with model selections and parameters,
- `edges`: the connection list including timing semantics,
- `assignments`: entity/model bindings (batch expansion inputs),
- `scenarioSettings`: global simulation configuration.

### Step 3: Compile (Scenario JSON → YAML)

The frontend sends Scenario JSON to `yaml_builder` (`POST /build`) and receives a YAML string. YAML is then shown to the user for review (preview/approval), enabling:

- traceability of what will be executed,

- early detection of missing parameters or unexpected expansions,
- reproducibility through a portable, human-readable artifact.

### Step 4: Persist (save YAML)

After approval, the frontend uploads the YAML to the Scenario Manager service, which persists it as a file. The scenario name (file name without extension) becomes the identifier used for execution.

### Step 5: Execute and monitor

The frontend requests execution from the Simulation Engine by submitting the scenario name. The Simulation Engine starts an asynchronous run and returns a simulation identifier. The frontend then:

- polls `/status`,
- renders progress and failure states,
- enables result exploration once the run reaches a completed state.

### Step 6: Results access and visualization

Results are produced as HDF5 files server-side. For visualization, the frontend first requests the list of available result files, then requests specific datasets (series/attributes) as JSON slices. This approach keeps the browser lightweight and avoids shipping large binary files to the client.

**Artifacts exchanged across services.** To make the lifecycle concrete, Table 5.1 summarizes the main artifacts exchanged between the frontend, the `yaml_builder` service, and the external platform services. In particular, Scenario JSON acts as the frontend's normalized export contract, YAML is the executable configuration persisted for reproducibility, and simulation outputs are accessed as plot-ready JSON slices derived from server-side result files.

Table 5.1. Key artifacts exchanged in the Coesi UrbanSim Frontend workflow.

| **Artifact** | **Produced by** | **Consumed by** | **Purpose** |
|---|---|---|---|
| Scenario JSON | Frontend | `yaml_builder` | Normalized representation of the authored graph (nodes, edges, assignments, settings). |
| YAML configuration | `yaml_builder` | Scenario Manager / Simulation Engine | Executable scenario definition for the runtime; reviewed by the user before persistence. |
| Scenario YAML file | Scenario Manager | Simulation Engine | Durable scenario storage (portable, inspectable). |
| Results datasets | Simulation Engine | Frontend (via APIs) | Output artifacts (HDF5 server-side) written by the runtime; accessed for visualization. |
| JSON result slices | Simulation Engine | Frontend | Plot-ready slices of results (no HDF5 parsing required in the browser). |

**End-to-end workflow sequence.** Figure 5.3 complements Table 5.1 by showing the corresponding message flow: validation requests, compilation from Scenario JSON to YAML via the `yaml_builder`, scenario persistence, execution triggering, status monitoring, and result retrieval for visualization.

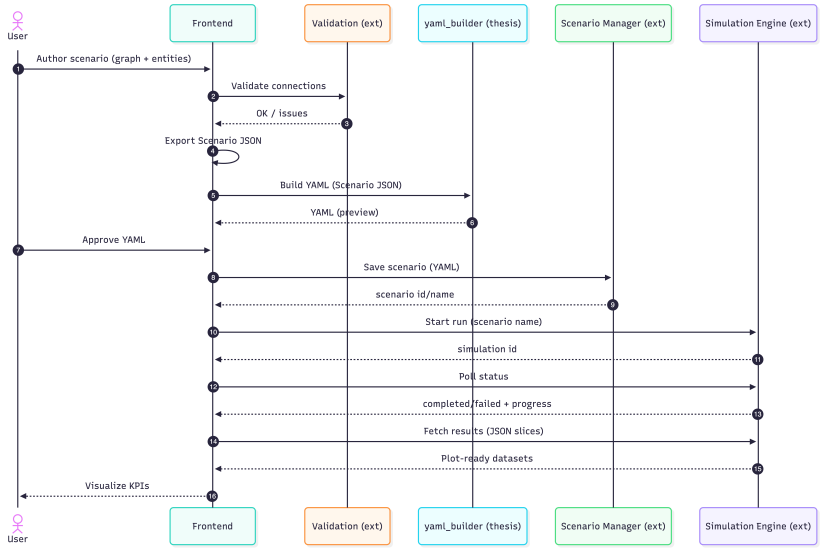


Figure 5.3. End-to-end scenario lifecycle in the Coesi UrbanSim Frontend workflow.

## 5.4 Design Decisions

This section motivates the main architectural choices that impact the thesis contributions, with emphasis on separation of concerns between interactive authoring (frontend) and configuration compilation (`yaml_builder`).

### Microservice separation for compilation (`yaml_builder`)

The graph-to-YAML transformation is treated as a *compiler-like* step with its own evolution pace (schema changes, expansion rules, compatibility constraints). Implementing it as a dedicated microservice isolates this complexity from the UI codebase, enables reuse from other clients, and supports independent testing of the compilation pipeline.

### Schema-driven UI and model registry integration

The frontend avoids hardcoding model-specific forms. Instead, it consumes model metadata from the external model registry and generates parameter editors dynamically. This design supports extensibility: new models can be onboarded into the registry without requiring frontend recompilation, while keeping the UI aligned with the authoritative model contract.

### Client-orchestrated workflow for transparency

Rather than hiding the lifecycle behind a monolithic "one-click run" endpoint, the frontend exposes intermediate steps. This choice aligns with the thesis goal of reducing configuration errors and increasing traceability of experiment definitions.

### File-based scenario persistence (YAML as the exchange artifact)

Scenarios are persisted as YAML files to keep experiment definitions portable and inspectable. File-based storage enables straightforward sharing, version control, and manual auditing, which is particularly valuable in research settings.

### Result access via JSON slices instead of client-side HDF5 parsing

Serving result slices as JSON preserves a thin-client frontend and simplifies deployment (standard HTTP, no special browser filesystem handling). It also allows the backend to enforce consistent dataset naming and slicing rules, while the frontend focuses on visualization UX.

### Limitations implied by the current architecture

- YAML is currently a one-way artifact in the UI workflow (JSON $\rightarrow$ YAML); importing YAML back into an editable graph is not implemented.
- Authentication/authorization is not a thesis contribution, and the reference Docker Compose setup uses development-mode token handling.

# Chapter 6

# Frontend Design and Implementation

## 6.1 Technology Stack and Code Organization

This section describes the implementation choices for the Coesi UrbanSim Frontend and how the codebase is organized to support interactive scenario authoring, incremental validation, and orchestration of external platform services. In line with the system boundary defined in Chapter 5, the frontend remains a thin client.

Figure 6.1 summarizes the internal layering of the frontend. The following subsections describe the main technology choices and how they map onto this modular organization.

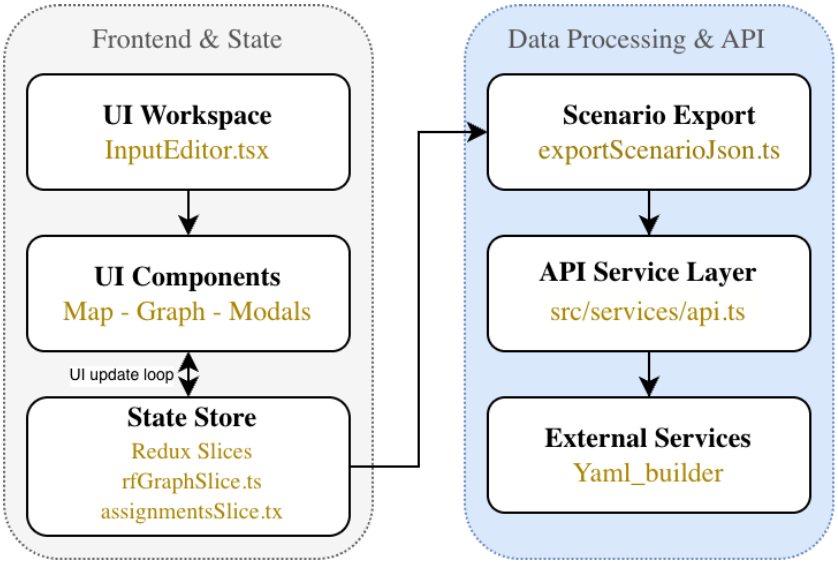


Figure 6.1. Frontend internal module map: main UI layers, state management, export step, and API boundary toward external platform services.

### 6.1.1 Core technology stack

The frontend is implemented as a single-page web application built with React and TypeScript. Development and bundling are handled via Vite, enabling fast iteration through hot-module reloading and producing an optimized static build for deployment.

Two interaction-heavy subsystems drive the user experience: (i) a node-based graph editor for composing scenarios, and (ii) a geospatial viewer for selecting and inspecting district entities. These subsystems are integrated into a single workspace so that entity selection, model configuration, and coupling definition remain visible and consistent during authoring.

Table 6.1 lists the main third-party libraries that enable the two interaction-heavy subsystems (graph editing and geospatial selection) and the overall application infrastructure.

Table 6.1. Key frontend libraries and their roles in the Coesi UrbanSim Frontend editor.

| **Library / tool** | **Version** | **Role in the frontend** |
|---|---|---|
| React | 18.3.0 | Component-based UI framework for the workspace, editors, and modals. |
| TypeScript | 5.7.2 | Static typing for UI state, scenario structures, and API contracts. |
| Vite | 6.3.1 | Build tooling and dev server for the frontend project. |
| @xyflow/react (React Flow) | 12.6.1 | Node-based graph editor: nodes, edges, drag/drop, and custom node types. |
| mapbox-gl | 3.12.0 | Interactive basemap rendering and map interactions (pan/zoom/style). |
| @deck.gl/* | 9.0.33 | High-performance rendering of building layers (e.g., GeoJSON) and selection overlays. |
| @reduxjs/toolkit | 2.7.0 | Primary application state management (graph state, assignments, settings). |
| zustand | 5.0.8 | Lightweight local state where needed (present in dependencies; complements Redux in specific components). |

## 6.2 Geospatial Editor and Entity Binding

District-scale co-simulation requires binding simulation models to many physical entities (e.g., hundreds of buildings). In the Coesi UrbanSim Frontend workflow, the frontend supports this step through a geospatial editor that enables users to (i) select buildings interactively, (ii) store selections as reusable *batches*, and (iii) bind these batches to the scenario graph so they can be expanded downstream during configuration compilation.

This aligns with the responsibility boundary described in Chapter 5, where the frontend provides interactive authoring and exports a normalized Scenario JSON contract including `assignments` (entity/model bindings).

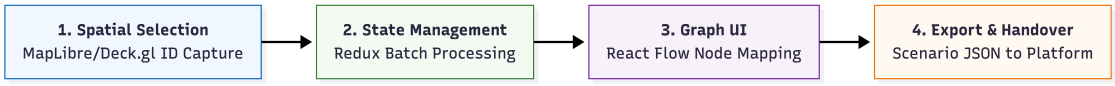


Figure 6.2. From spatial selection to Scenario JSON

Figure 6.2 summarizes the implemented pipeline: the map interaction produces a set of selected building IDs; the selection is stored in the global state as named batches; batches are represented in the graph UI as dedicated nodes; and the complete workspace (graph + assignments + settings) is exported as Scenario JSON for downstream compilation and execution.

### 6.2.1 Data source and rendering stack

The current prototype geospatial editor visualizes building geometry from a static GeoJSON demo dataset (a `FeatureCollection`) imported in the map component. A live `/urbansim` integration is planned behind the same gateway prefix, but is not required for the interaction and evaluation goals of this prototype. Each feature includes a unique building identifier in `feature.properties.id` and carry additional attributes (e.g., height, construction year, usage, surface, EPC rating), which are used for inspection and lightweight summaries in the UI.

Rendering is implemented with Deck.gl using a `GeoJsonLayer` over a MapLibre base map. The layer is `pickable` and supports both hover and selection highlighting via accessors that change fill/line colors depending on the hovered feature and the current selected set.

### 6.2.2 Spatial selection mechanisms

Selection supports interactive picking and polygon-based selection. For polygon selection, the UI uses Mapbox Draw (`@mapbox/mapbox-gl-draw`) in `draw_polygon` mode, while the spatial filtering is performed client-side using a custom point-in-polygon implementation based on ray casting. Concretely, for each building polygon the frontend computes a centroid (mean of polygon coordinates) and then checks whether that centroid lies inside the drawn polygon.

This choice is computationally lightweight and easy to reason about, but it implies a well-defined limitation: selection is centroid-based (a building that intersects the drawn polygon but whose centroid lies outside will not be selected). This is acceptable for the current prototype and can be replaced by full polygon intersection if higher geometric fidelity becomes necessary.

### 6.2.3 Batch creation and state representation

Selected building IDs are normalized into named batches (assignments) through the assignment panel. The UI prevents empty batches by disabling the creation action when no IDs are selected. When created from the map selection, a batch is assigned a default name pattern (e.g., `batch_map_{N}`) and is stored in the Redux `assignmentsSlice`.

The internal batch structure records a stable identifier, name, supported entity type (`building`), and the list of selected IDs. The slice also maintains convenience metadata such as `count` and `createdAt` to support UI listing and inspection.

Beyond manual selection, the assignment view supports rule-based creation via dataset splitting: a percentage- and seed-driven function (`samplePercentWeighted`) can divide the full building set into groups, enabling quick scenario partitioning for comparative runs (e.g., group A vs. group B).

Table 6.2. Batch creation mechanisms supported by the assignment management panel.

| **Mechanism** | **User inputs** | **Resulting batch** |
|---|---|---|
| Manual batch from map selection | Current selected IDs (interactive pick or polygon selection) and optional name | Creates one `Assignment` with `entityType=building` and explicit `ids[]` list; creation is disabled if selection is empty. |
| Rule-based dataset split | Percentage and random seed (optional naming convention) | Generates batches by sampling/splitting the building set using `samplePercentWeighted`; enables quick partitioning for comparative runs (e.g., A/B groups). |

The batch creation interface is implemented as a modal dialog that unifies spatial selection and batch definition. The user assigns a batch name, inspects the current selection as an explicit list of building identifiers, and can remove individual buildings from the batch before finalizing creation. A contextual details panel provides quick feedback on the currently hovered or selected building (e.g., height, surface, usage, and construction period), supporting rapid sanity checks during district-scale selection.

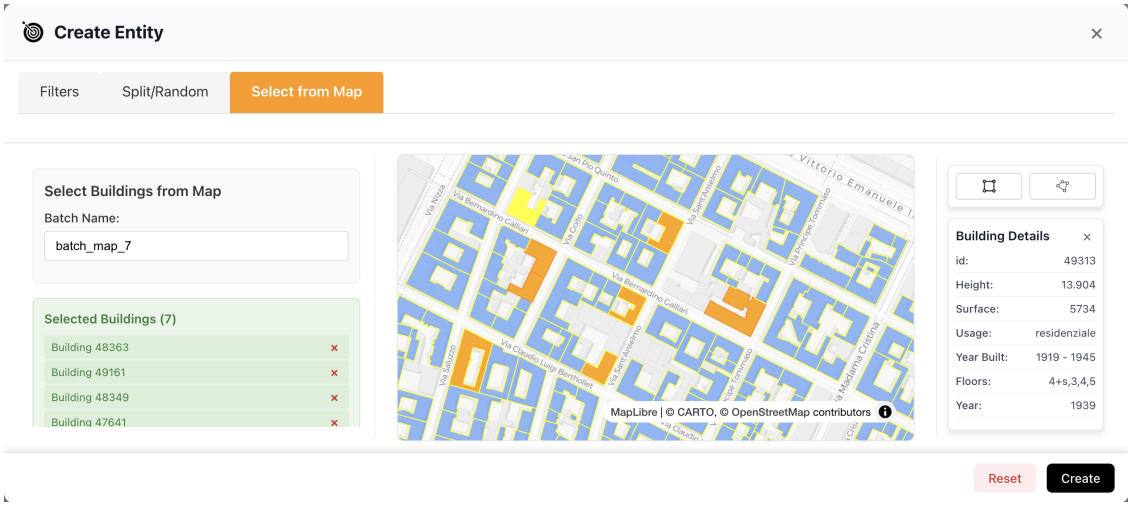


Figure 6.3. Create Entity dialog (Select from Map): interactive building selection and batch definition with naming, selected-ID list, and per-building attribute inspection.

### 6.2.4 Binding batches to the scenario graph

Batches become first-class objects in the node editor. They are represented as specialized React Flow nodes: nodes carrying `data.assignmentBatchId` are rendered with distinct styling and dedicated handles, enabling connections between a batch and downstream model nodes. This representation allows a model to be conceptually bound to "all buildings in batch X" without duplicating nodes for each building in the editor, keeping the authored graph compact and readable at district scale.

The UI also provides a lightweight "review" function for batch nodes: a modal can present basic batch analytics (e.g., mean height/year/usage) derived from the GeoJSON attributes, supporting quick sanity checks of the selection.

### 6.2.5 Export contract and current scope

At export time, assignments are included in the Scenario JSON contract as the execution-relevant binding artifact. The export step filters assignments to retain only `entityType=building` and emits a minimal payload (`id`, `name`, `entityType`, `ids`), intentionally excluding UI-only metadata such as timestamps. This keeps the contract stable and focused, while downstream services (notably `yaml_builder`) perform the necessary expansion to per-building instances.

In its current thesis scope, entity binding is intentionally limited to buildings (`entityType:building`). Extending the same mechanism to additional entity types (e.g., network assets or DER components) is a natural future step, but is out of scope for the present prototype.

## 6.3 Node-Based Scenario Editor

The Coesi UrbanSim Frontend provides a node-based editor where a scenario is authored as a directed graph of components connected through explicit ports. This representation follows the co-simulation abstraction introduced in Chapter 2 (directed dependency graphs), making data-flow dependencies visible and supporting incremental preparation of the Scenario JSON export contract (nodes, edges, assignments, and scenario settings). The editor is implemented using React Flow (`@xyflow/react`) and is integrated into the main workspace page (`src/pages/InputEditor.tsx`), with graph state managed centrally through Redux (notably `src/slices/rfGraphSlice.ts`) to enable consistent updates across the canvas, side panels, and export pipeline.

### 6.3.1 Node types and authoring interactions

The editor exposes a small set of custom node types, each corresponding to a different authoring purpose:

- **Model instance nodes (`parameditor`).** These nodes represent instantiations of catalog models (e.g., demand models, data sources, controllers). They are created through a palette and drag-and-drop interaction and act as the primary building blocks of the scenario graph.

- **Batch/entity nodes (`batchEntity`).** These nodes represent entity batches produced by the geospatial workflow (Section 6.2). Internally, they reference a batch via `data.assignmentBatchId` and are rendered with distinct styling and dedicated handles to emphasize their role as *scaling drivers* rather than simulation components.

- **Composite blocks (`CompositeBlock`).** Composite blocks provide a lightweight grouping mechanism for reuse and readability during authoring. At export time, composite structures are flattened so that the Scenario JSON contains an atomic node list suitable for compilation (as described in the frontend export responsibilities in Chapter 5).

Table 6.3 summarizes the node types exposed by the editor and clarifies their role in the exported Scenario JSON contract. In particular, batch nodes encode the entity-binding intent from Section 6.2, while model nodes carry the metadata-driven parameterization and port interfaces used for coupling.

Table 6.3. Main node types in the scenario editor and their roles in the Scenario JSON contract.

| **Node type** | **Purpose in authoring** | **Key fields stored** |
|---|---|---|
| Model instance (`parameditor`) | Instantiate a registry model and configure its parameters and I/O ports for coupling. | Model id/version; parameters; declared ports. |
| Batch/entity (`batchEntity`) | Represent a reusable building batch created in the geospatial editor; enables scaling/broadcasting without duplicating nodes. | `data.assignmentBatchId`; node label/metadata. |
| Composite block (`CompositeBlock`) | Group parts of the graph for readability during authoring; flattened at export. | Subgraph membership; layout metadata. |

In addition to creation and placement, the editor supports node selection and editing via contextual modals. Batch nodes expose a "review" interaction that shows simple analytics over the selected building set (e.g., mean height/year/usage), providing a fast sanity check before binding a batch to models.

### 6.3.2 Schema-driven ports and parameter editing

A key design goal is to avoid hardcoding model-specific configuration forms. Instead, the frontend consumes model metadata from the external Model Manager and generates editing views dynamically. The metadata includes (at minimum) model parameters and the declared interface in terms of `input_variables` and `output_variables`. The editor uses these declarations in two ways: (i) to populate the parameter editor modal with the appropriate fields and defaults, and (ii) to render connection handles for the node's inputs/outputs so that edges are created against explicit port names rather than implicit conventions.

The parameter editor modal is implemented as a dynamic form: lists of parameters are mapped to input controls, and changes update the node's stored configuration in the global state. This approach keeps the UI extensible: onboarding a new model through the registry can immediately make it selectable and configurable in the editor without requiring frontend-specific form code.

### 6.3.3 Computation settings and parallel execution configuration

Beyond model parameters and port bindings, the Frontend exposes *computation settings* at node level to control how replicated model instances are distributed across runtime processes. This is particularly relevant when a bindable model node is instantiated for many entities (or when multiple instances of the same model appear in the scenario), since the resulting execution workload can be partitioned to improve throughput and reduce wall-clock time.

**Parallelization strategy UI.** For each configurable model node, the *Computation setting* tab provides a *Number of Parallel Processes* field that the user can adjust within backend-provided limits (minimum/maximum). The UI also reports a concise load summary (e.g., *Total Models Detected* and an approximate *Models per Process*) to make the effect of the chosen value transparent. When the number of processes matches the number of detected model instances, the configuration corresponds to maximum parallelism (approximately one model per process).

**Propagation to execution artifacts.** The selected process count is persisted as part of the scenario definition and exported in the Scenario JSON as `numberOfParallelProcesses` for the corresponding node. During compilation, the `yaml_builder` uses this value when expanding bindable nodes into concrete instances and when grouping those instances into process-level containers in the emitted YAML configuration (see Chapter 7 for the compiler-side algorithm and its mapping to the YAML schema).

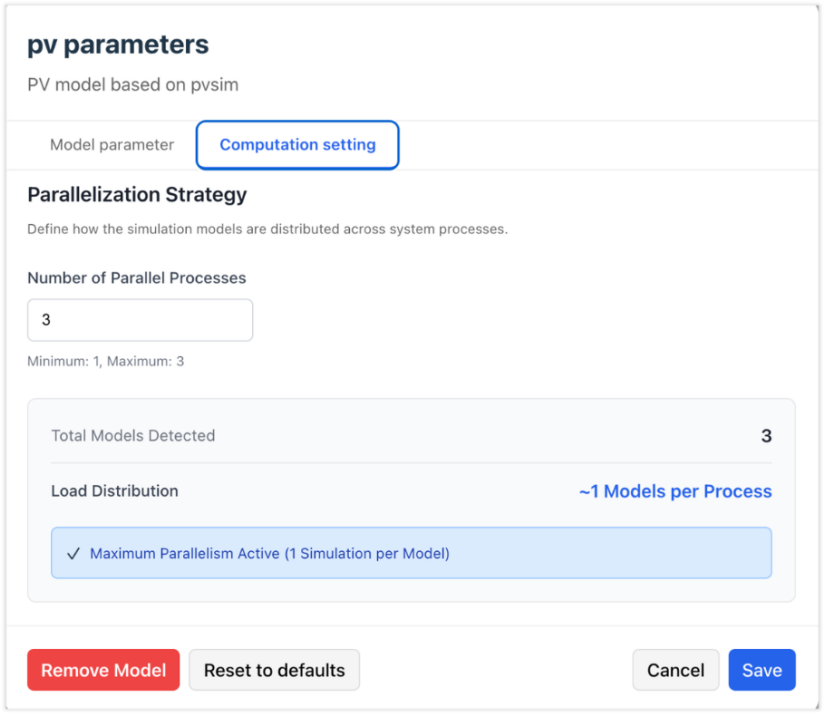


Figure 6.4. Computation settings in the model configuration dialog

### 6.3.4 Connections and temporal coupling semantics

Edges in the graph represent directed data-flow from a source output port to a target input port. Beyond simple connectivity, each edge stores a *temporal coupling semantics* consistent with the co-simulation time model used in this thesis: *Same Time* edges represent synchronous exchange at the current step, while *Next Time* edges represent

delayed exchange where the source output at step $t$ is consumed at step $t+1$ to break instantaneous feedback loops.

In the UI, the semantics are set per-connection through a dedicated modal. The selected option is stored as `edge.data.connectionType` with values `"Same Time"` or `"Next Time"` (defaulting to `"Same Time"`). During export, this is translated into the normalized Scenario JSON representation, and later compiled by `yaml_builder` into the runtime-specific coupling representation (see Chapter 7 for the formal handling of time-shifted links and loop breaking).

### 6.3.5 Connection rules and validation feedback

While the graph editor allows users to draw connections interactively, semantic correctness is enforced through platform validation services rather than through duplicated rule logic in the browser. Figure 6.5 summarizes the implemented workflow: after a user attempts to connect an output port to an input port, the frontend submits the candidate connection to the external Validation Engine via `POST/api/v1/validate/port-connections`. The service returns a boolean `valid` flag and, when invalid, a list of human-readable reasons that explain the incompatibility. This mechanism supports the functional requirements for typed graph composition and coupling validation (see Table 4.1, FR-3–FR-6) while keeping the frontend aligned with the platform's authoritative semantic rules.

In the current implementation, validation feedback is surfaced primarily through toast notifications and connection-level modal messages (global UX feedback), rather than through fine-grained inline annotations directly on ports. This choice keeps interaction lightweight and consistent with the "thin client" boundary, but it also motivates a clear UX refinement direction: richer, localized highlighting of the specific ports, types, or units involved in a rejected connection.

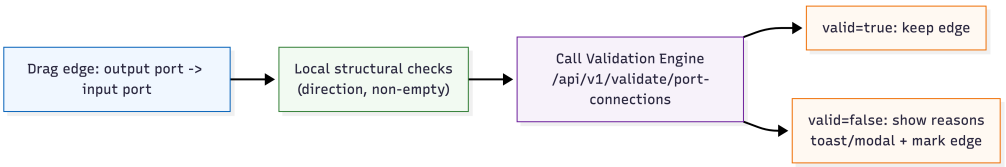


Figure 6.5. Connection validation flow in the frontend: interactive edge creation followed by server-side semantic checks and user feedback.

### 6.3.6 Batch binding and broadcasting intent in the graph

The editor encodes *broadcasting/scaling intent* by allowing batch nodes to be connected to model nodes. Conceptually, this expresses that a model instance should be replicated or parameterized over the full set of entities contained in the batch. The export step preserves both (i) the batch definitions in the `assignments` array and (ii) the graph edges that link batches to downstream models. Downstream, `yaml_builder` uses this information to expand the logical graph into an executable configuration that can run

per-entity processes (formalized in Chapter 7, including broadcasting and parallelism derivation).

This design keeps the authored graph compact at district scale (one batch node instead of hundreds of per-building nodes) while still providing enough structure for deterministic compilation and reproducible execution.

## 6.4 Scenario Management and Persistence

Scenario management in the Coesi UrbanSim Frontend must balance two competing needs: (i) keeping interactive authoring fast and forgiving (graph edits, map selection, parameter tweaks), and (ii) persisting an *inspectable* and *reproducible* experiment definition that can be executed by external services. In line with the platform boundary described in Chapter 5, the frontend treats the editable scenario primarily as *in-memory UI state*, while durable persistence is delegated to the Scenario Manager as a file-based artifact (YAML), which is the runtime-executable configuration produced by `yaml_builder`.

To clarify what is persisted at each stage, Figure 6.6 summarizes the persistence layers used by the frontend. Interactive authoring is stored primarily as in-memory Redux state. Durable persistence is achieved by saving the compiled YAML configuration to the Scenario Manager, which serves as the executable artifact for subsequent runs.

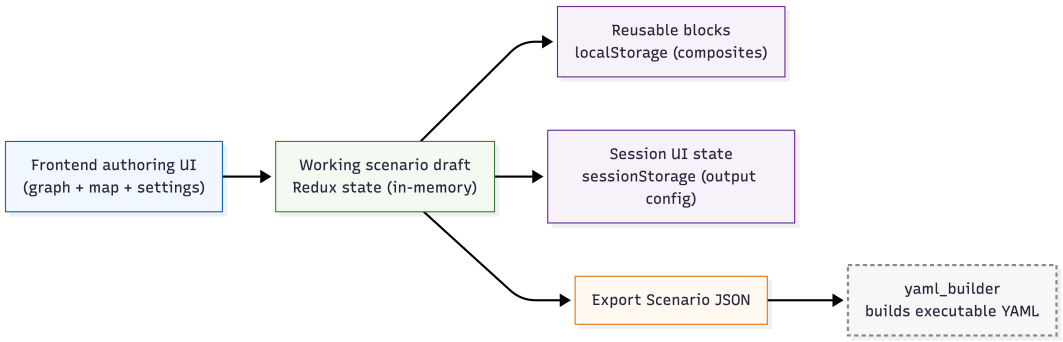


Figure 6.6. Persistence layers in the Coesi UrbanSim Frontend workflow: in-memory authoring state, local/session storage aids, and durable YAML persistence via the Scenario Manager.

### 6.4.1 Working scenario as browser-side state

During authoring, the scenario exists as a set of Redux-managed slices that capture the graph (nodes and edges), entity batches (`assignments`), and global scenario settings. This structure supports incremental edits across multiple panels (graph canvas, inspectors, modals) without introducing backend round-trips for every small change. The resulting "working scenario" is therefore best understood as an *editable draft*: it is optimized for interaction and can be exported at any time into the normalized Scenario JSON contract used by the compilation pipeline (Chapter 5).

### 6.4.2 Local persistence for authoring aids

While the full scenario draft is not persisted as a first-class project object in the current prototype, the frontend does persist two *authoring aids* locally to improve usability across browser sessions:

- **Composite blocks.** User-defined composites under a dedicated key (e.g., `coesi.composites`) so they remain available. This enables lightweight reuse of frequently used subgraphs.
- **Output configuration (sessionStorage).** Output-selection state is stored with a session-scoped key (e.g., `output-config:{path}`), allowing the user to navigate within the UI without losing output choices, while still avoiding long-lived persistence of potentially inconsistent partial drafts.

These local mechanisms are intentionally narrow: they preserve reusable building blocks and UI preferences, but do not attempt to provide full "save/load scenario" semantics for the entire workspace graph.

Table 6.4. Persistence layers in the frontend workflow and the scope of stored state.

| **Layer** | **Mechanism** | **What is persisted (scope)** |
|---|---|---|
| Working scenario draft | In-memory Redux state | Graph (nodes/edges), assignments, settings during interactive editing; optimized for responsiveness. |
| Reusable authoring blocks | `localStorage` | Composite definitions (user-created composite blocks) to enable reuse across sessions. |
| Session-only UI preferences | `sessionStorage` | Output-selection configuration scoped to the current browser session. |
| Durable scenario artifact | Scenario Manager (server-side) | YAML configuration file (reviewed by the user), used as the executable input for the Simulation Engine. |

### 6.4.3 Durable persistence as YAML

The durable, shareable representation of a scenario in this workflow is the compiled YAML configuration. After the user triggers the build step, the frontend exports Scenario JSON and invokes `yaml_builder` to produce the executable YAML. The YAML is then shown to the user for inspection (preview/review); only after explicit approval does the frontend upload the YAML to the Scenario Manager (`POST /api/v1/scenarios`). The Scenario Manager persists this YAML as a file, and the *scenario name* (file name without extension) becomes the identifier used later to trigger execution.

This choice directly supports reproducibility goals: the persisted artifact is human-readable, portable, and suitable for sharing and version control, while the UI remains a thin orchestrator rather than a stateful backend for draft scenarios.

### 6.4.4 Current limitations and planned extensions

Two scope-limiting consequences follow from the current persistence design.

First, persistence is *one-way* in the UI workflow: the frontend supports Scenario JSON → YAML, but importing an existing YAML back into an editable node graph is not implemented. Second, project-style scenario management features such as version history/delta tracking, scenario duplication, or undo/redo of workspace edits are not implemented in the current prototype. These are natural next steps if the platform evolves toward long-lived collaborative scenario repositories rather than file-based exchange artifacts.

## 6.5 Execution UX and Monitoring

This section describes how the frontend supports the final stages of the primary user workflow (UC5): transforming an authored scenario (graph + assignments + settings) into an executable configuration, making that configuration inspectable before execution, submitting a run, and monitoring progress until completion. The interaction design intentionally exposes intermediate artifacts (especially the generated YAML) to support review, debugging, and reproducibility, rather than hiding the lifecycle behind a single opaque "Run" action (FR-7–FR-9 in Table 4.1).

### 6.5.1 YAML generation and review (approval step)

As detailed in Section 5.3, execution begins by exporting the current workspace into the Scenario JSON contract and compiling it into an engine-ready YAML configuration via `yaml_builder`. In the frontend, this step is exposed as *Build YAML*: the system opens a dedicated review modal containing the generated YAML with code-style formatting (Fig. 6.7). The modal supports lightweight inspection actions (scroll/search and copy) and an optional local download for archiving or version control. Execution remains blocked until the user explicitly approves the artifact, implementing the "preview-before-execute" requirement (FR-8) and providing a natural checkpoint when troubleshooting validation or coupling issues.


Figure 6.7. YAML review modal shown after compilation: the generated configuration can be inspected (and optionally copied/downloaded) before it is uploaded and executed.

Only after approval does the frontend proceed to persistence and run submission (Section 6.5.2).

### 6.5.2 Run submission and scenario persistence

After user approval, the frontend persists the YAML to the platform scenario storage service. In the current architecture, the YAML file is treated as the canonical persisted scenario artifact (file-based persistence), which keeps scenarios portable and inspectable across environments. Once stored, the frontend submits an execution request to the simulations service, referencing the stored scenario. The run identifier returned by the backend becomes the stable handle used for monitoring status and retrieving result artifacts (FR-9).

For clarity, Table 6.5 summarizes the main UX steps and the corresponding API interactions exposed by the platform boundary.

Table 6.5. Execution UX steps and corresponding platform interactions.

| **UX step** | **Service interaction** | **User-visible feedback** |
|---|---|---|
| Generate YAML | `POST {YAML_BUILDER}/build` (Scenario JSON → YAML) | YAML preview modal opens; compilation errors shown as blocking feedback. |
| Approve & persist | `POST /api/v1/scenarios` (store YAML) | Confirmation toast and/or modal state update indicating the scenario was saved. |
| Submit run | `POST /api/v1/simulations` (execute scenario) | Run created; execution panel opens with initial status (e.g., queued/running). |
| Monitor status | `GET/api/v1/simulations/<id>/status` (polling) | Step-based progress updates; failure states surface reasons/logs when available. |

### 6.5.3 Progress monitoring and status polling

Once a run is submitted, the frontend switches to an execution-monitoring view implemented as a step-based preview panel. Rather than showing a single "loading" indicator, the panel communicates discrete stages (e.g., configuration accepted, queued, running, completed/failed), aligning with the usability principle of *visibility of system status* (Section 4.4). Status is refreshed by periodically polling the simulations service for the current run state, and the UI updates the stepper accordingly.

When failures occur, the monitoring view surfaces a concise failure state and the available diagnostic information returned by the backend (e.g., error reason strings and logs when provided). In the current prototype, feedback is primarily delivered through global UI mechanisms (modal messages) rather than fine-grained inline annotations, keeping the client thin while still supporting rapid recovery.

### 6.5.4 Freeze context and reproducibility of a running execution

A practical risk in interactive editors is confusing the *currently edited* graph with the *configuration actually being executed*. The execution UX mitigates this by treating each run as a *frozen context* anchored to a specific uploaded YAML artifact and run identifier. Monitoring and result inspection always refer to that immutable run handle, ensuring that progress and outputs remain traceable to the exact configuration that was approved in the YAML review step. Any subsequent edits in the workspace require regenerating and re-approving a new YAML artifact to create a new run, preserving a clear audit trail.

### 6.5.5 Results entrypoint and lightweight visualization

After completion, the execution view exposes an entrypoint to inspect available outputs. In line with the thin-client boundary, the frontend does not attempt to parse large binary

artifacts directly in the browser. Instead, result visualization is driven by backend-served slices (e.g., JSON responses derived from stored datasets), which the frontend renders in a lightweight chart/table view. This keeps deployment simple (standard HTTP) and ensures consistent dataset naming and slicing rules are enforced server-side.

To support rapid iteration during scenario design, the Frontend includes a lightweight results viewer that allows users to inspect key output time series directly in the browser after a run completes. Results are grouped by simulator/model output namespaces and by concrete instance, and users can overlay multiple variables to perform quick sanity checks (e.g., daily cycles, ranges, and correlation between drivers and responses) before exporting data or computing KPIs.

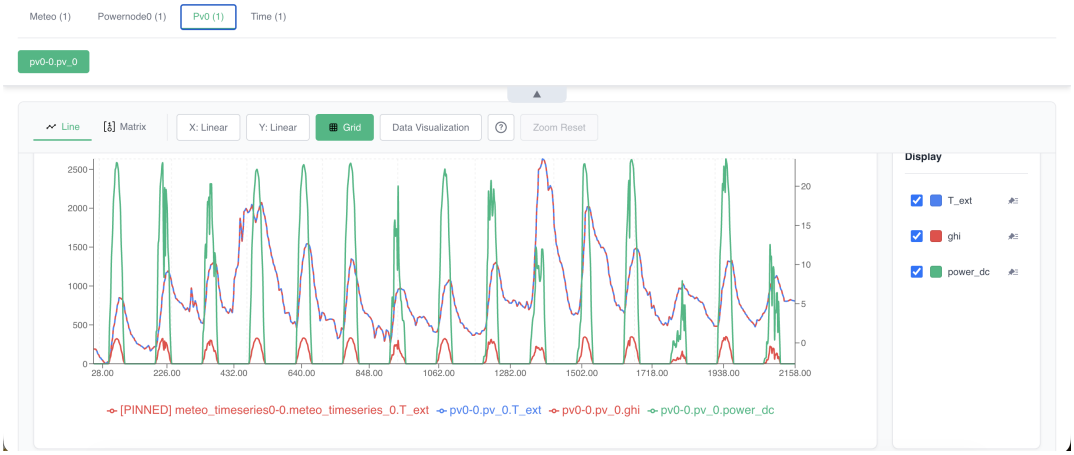


Figure 6.8. Results inspection entrypoint after completion: example visualization based on backend-served data slices, rendered as charts/tables in the frontend.

### 6.5.6 Model management and onboarding UI

Although the model registry and catalogue are provided by external platform services, the Frontend implements a dedicated *Model Management* workspace that exposes model onboarding and catalogue operations through the same REST layer used elsewhere in the application. Concretely, this functionality is implemented as a unified drawer component (`src/components/Projects/AddModelJsonDrawer.tsx`) that provides three interaction modes: *Data input*, *Model builder*, and *Model catalogue*. Internally, user inputs are stored in a single `modelData` state object that maps directly to the platform model-definition schema; a dedicated utility (`buildModelJson`) assembles this state into a schema-compliant JSON artifact prior to submission.

**Backend integration and authentication.** The Frontend integrates with the external *Model Manager* microservice via the `/api/v1/models` endpoints. In particular, the catalogue is retrieved with `GET /api/v1/models?details=true`, a specific model can be inspected using `GET /api/v1/models/{name}`, new models are registered using `POST`

`/api/v1/models`, and removal is supported via `DELETE /api/v1/models/{name}`. Requests are issued through a shared `ApiService`, which transparently attaches a `Bearer` token (from browser local storage) to the `Authorization` header.

**JSON-driven onboarding (Data input).** The *Data input* tab allows users to provide a model definition directly, either by uploading a `.json` file or by pasting JSON into an in-browser editor. File uploads enforce a 2 MB size limit and validate the `.json` extension before parsing. The editor provides real-time feedback: JSON syntax and schema compliance are checked on a debounced interval (300 ms) using a local validation utility (`validateModelJson`), enabling early detection of missing required fields and malformed variable declarations. This workflow supports reproducibility and portability, as definitions can be stored and re-registered across environments subject to backend validation.

**Guided onboarding (Model Builder).** The *Model Builder* tab provides a structured form for creating the same JSON artifact without requiring users to manually author JSON. The form captures key metadata (e.g., `name`, `description`, `tags`, `solver`, `model_execution_cmd`, and simulator identifiers) and then compiles it into a valid model definition via `buildModelJson`. Field-level guards reduce submission errors: the *Add model* action is disabled until mandatory fields (e.g., `name` and `model_execution_cmd`) are present and required arrays/attributes (notably `input_variables` and `output_variables` with `start_value`) satisfy the schema constraints. On submission, the frontend issues a `POST /api/v1/models` request with the complete model definition. If the user enables the *Share with community* option, the model is persisted via the backend; otherwise, the definition is stored locally in the browser (local storage) for personal use.

**Catalogue integration and model lifecycle operations.** The *Model catalogue* tab lists available models retrieved from the backend catalogue (`GET/api/v1/models?details=true`) and supports detail inspection (via `GET` on `/api/v1/models/\{name\}`). Importantly, the thesis does not implement the registry service itself; the contribution here is the UI integration that exposes catalogue browsing, onboarding, and lifecycle operations (register/view) in a coherent user-facing workflow.

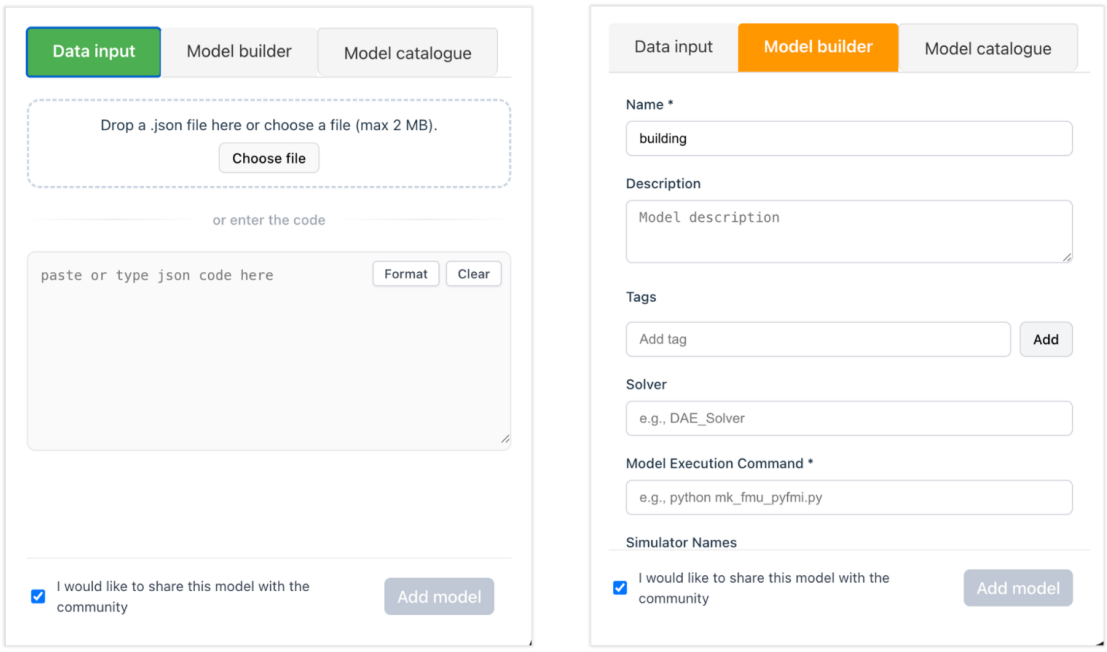


Figure 6.9. The Frontend's Model Management drawer generates schema-compliant JSON through either direct file/editor input or a guided form.

**Error handling and conflict resolution.** Validation is intentionally *dual-layered*. Client-side validation provides immediate feedback during editing, while backend validation remains the source of truth for conflicts and persistence rules (e.g., duplicate model names). When the backend returns an error (e.g., `400 Bad Request`), the UI surfaces the message prominently in the drawer. If a local model conflicts by name, the UI triggers a replacement/overwrite confirmation flow to resolve the conflict.

## 6.6 Error Handling and Resilience

Given the distributed nature of the Coesi UrbanSim Frontend workflow, failures can occur at multiple layers: UI rendering (graph canvas), client-side orchestration (export/build/-submit/poll), and external platform services (validation, scenario storage, simulations, results). The frontend therefore adopts a layered strategy for resilience: prevent avoidable errors early (through validation), isolate UI crashes so the workspace does not blank out, and surface actionable feedback so users can recover without losing work. These choices directly support the robustness and error-handling goals stated in the non-functional requirements (Section 4.3) and the incremental validation requirement (FR-6 in Table 4.1).

### 6.6.1 UI-level fault isolation (error boundaries)

Interactive graph editors are sensitive to runtime exceptions (e.g., unexpected node/edge payloads, custom node rendering failures). To prevent a single rendering exception from crashing the entire application, the node editor wraps the React Flow canvas in a dedicated React error boundary (`ReactFlowErrorBoundary`). The boundary implements the standard React mechanisms (`getDerivedStateFromError` and `componentDidCatch`) to capture exceptions originating inside the graph subtree and replace only that region with a fallback UI, while keeping the surrounding workspace (navigation and side panels) mounted.

The fallback explicitly surfaces a short error summary, provides a collapsible details section for diagnosis, and includes a *Try Again* action that resets the editor region without requiring a full page reload. In addition, the boundary performs targeted logging for known immutability-related failure modes (e.g., frozen-object warnings) to support debugging without disrupting the user workflow.

### 6.6.2 API and orchestration error handling

Most failure modes in the workflow originate from external service calls (compile, persist, execute, poll, retrieve results). To keep handling consistent, the frontend routes these interactions through a centralized API service layer (`src/services/api.ts`), where requests are issued and responses are normalized into a small set of user-facing outcomes: (i) success (continue workflow), (ii) validation-style failure (actionable reasons), (iii) operational failure (network/HTTP/server error).

For blocking steps in the execution pipeline (e.g., YAML compilation and run submission), errors are treated as *hard stops*: the UI keeps the user in the current step, reports the failure, and allows retry after the underlying issue is fixed (e.g., correct an invalid connection, fill missing parameters, or restore API availability). For non-blocking steps (e.g., background polling), failures are surfaced without destroying the monitoring context, so temporary backend outages do not reset the monitoring UI.

### 6.6.3 User-facing feedback (toasts and modal messages)

The primary feedback channel is a global toast mechanism that supports severity levels (success, info, warning, error) and a short duration suitable for frequent workflow events (e.g., connection validation results, save confirmations, transient API errors). For user actions that require explicit acknowledgment (e.g., YAML review, connection semantics editing), the UI also uses modal dialogs to present context-rich messages and enforce confirmation before proceeding.

In the current prototype, feedback is intentionally *global* (toasts/modals) rather than field-level inline annotations. This choice keeps the frontend thin and reduces duplication of backend validation logic, but it also identifies a clear refinement opportunity: attach validation messages directly to the affected port/edge/parameter controls to minimize visual search and improve recoverability.

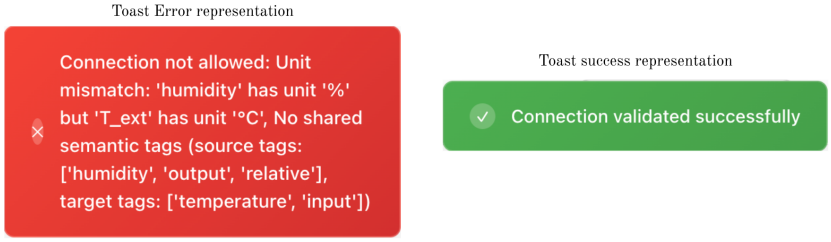


Figure 6.10. Example of global error feedback via toast notifications during models port validation.

Table 6.6. Typical error sources in the workflow and corresponding UI recovery mechanisms.

| **Error source** | **How it is surfaced** | **Recovery action** |
|---|---|---|
| Invalid port connection (semantic mismatch) | Blocking feedback with human-readable reasons returned by validation; shown in global UI messages. | User edits the edge (or removes it) and retries connection. |
| Compilation failure (Scenario JSON → YAML) | Blocking error during "Generate/Preview YAML"; user remains in the build/review step. | Fix scenario definition (parameters/graph), then regenerate YAML. |
| Scenario persistence failure (upload YAML) | Error toast/modal; scenario is not stored and execution cannot proceed. | Retry after restoring platform availability or correcting request payload. |
| Execution submission / status polling failure | Monitoring view remains visible; transient failures reported through toasts. | Retry submission, or continue polling once service recovers. |
| UI rendering exception (graph canvas) | Caught by `ReactFlowErrorBoundary`; fallback UI shown. | Reload the editor or adjust problematic graph state; continue authoring. |

### 6.6.4 Cancellation and polling hygiene

Long-running interactions (especially status polling during execution monitoring) require careful cleanup to avoid duplicate requests, race conditions, or stale UI updates. The frontend uses request cancellation mechanisms (e.g., `AbortController`) so that in-flight requests can be safely aborted when the user navigates away, restarts an action, or when a polling loop is stopped. This prevents memory leaks and reduces misleading states caused by late-arriving responses from older requests.

### 6.6.5 Current limitations and future hardening

Two resilience limitations are explicit in the current prototype. First, HTTP requests are generally issued as a single attempt: the codebase does not implement automatic retry with exponential backoff for transient errors. Second, validation feedback is not yet localized at the field level (e.g., highlighting the exact port or parameter responsible for a failure).

Future work can harden robustness without changing the platform boundary by adding (i) controlled retries for idempotent requests (polling, metadata fetches) with backoff and jitter; (ii) richer error taxonomy (network vs. server vs. semantic validation) to tailor messaging; and (iii) inline annotations for parameters and ports to reduce recovery time during complex scenario composition.

# Chapter 7

# YAML Builder Design and Implementation

This chapter describes the design and implementation of the `yaml_builder` microservice, which compiles the frontend scenario graph into an executable YAML configuration for the Coesi runtime. In the end-to-end workflow introduced earlier, the frontend acts as an *authoring tool* for a high-level, human-composable representation (nodes, typed ports, edges, and entity bindings), while the `yaml_builder` performs a *compiler-like* transformation into a strict schema that can be consumed by the simulation engine.

## 7.1 Service Purpose and Interfaces

The `yaml_builder` is implemented as a stateless Python microservice (FastAPI) and deployed as a Docker container. Its responsibility is narrowly scoped: given a scenario definition (JSON) produced by the frontend, it returns YAML renderings of the corresponding executable configuration.

**Why a dedicated compilation service.** Treating graph-to-YAML transformation as a separate service isolates schema evolution and expansion logic from the UI codebase, enables reuse by other clients, and supports focused testing of compilation rules (e.g., instance expansion, connection resolution, and coercion). This separation is aligned with the architectural boundary introduced in Chapter 5, where configuration generation is positioned as an independent platform function rather than a UI concern.

**Deployment and endpoint contract.** The service runs in a containerized environment and exposes an HTTP interface. The main endpoint is:

- `POST /build`

Figure 7.1 summarizes the interaction between the frontend and the `yaml_builder`. The key point is that YAML generation is a stateless compilation step: the frontend

submits Scenario JSON to `POST /build` and receives a YAML artifact that is previewed and only then confirmed for persistence/execution.

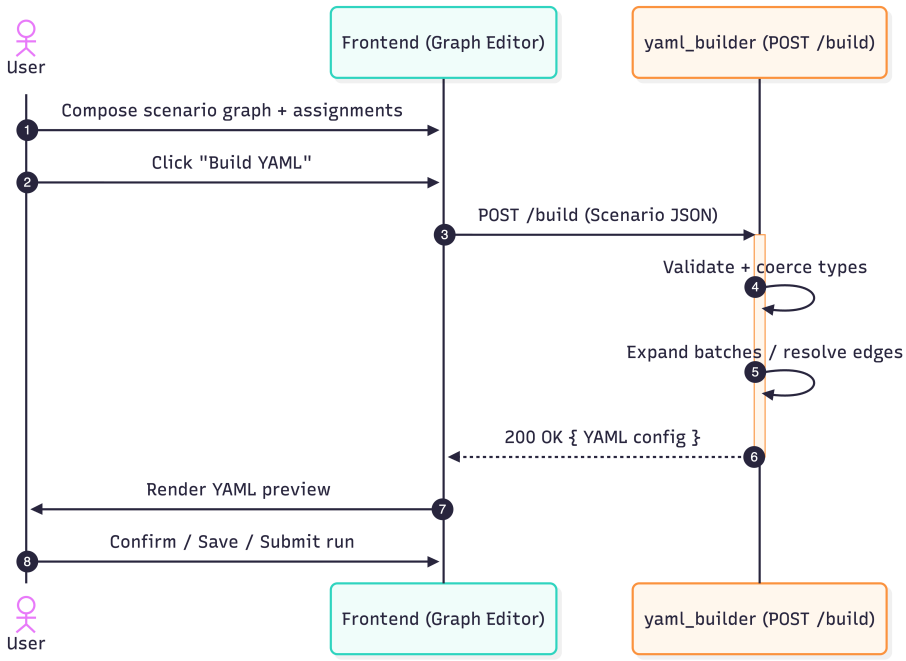


Figure 7.1. Build-and-preview interaction: the frontend submits Scenario JSON to `POST /build` and receives YAML (V2 as primary output) for user preview before confirmation.

This sequence clarifies why compilation is separated from authoring: the editor stays focused on graph construction, while the service guarantees a deterministic translation to the engine-facing schema.

The request body is a JSON object representing the scenario. The response is a JSON payload containing the generated YAML as strings. Errors are returned as `400 Bad Request` with diagnostic details.

**Example (request/response shape).** Listing 7.1 illustrates a minimal valid request structure, while Listing 7.2 shows the high-level organization of the corresponding YAML.

```
{
  "project": "DemoProject",
  "scenarioSettings": {
    "scenarioName": "TestScenario",
    "days": 1,
    "startDate": "2024-01-01",
    "stepSize": 600
  },
  "nodes": [
    {
```

```
      "id": "node_1",
      "data": {
        "modelDefId": "pv_system",
        "name": "MyPV",
        "numberOfParallelProcesses": 1,
        "model_parameters": [
          { "name": "capacity", "value": 5.0, "data_type": "float" }
        ],
        "input_variables": [],
        "output_variables": [
          { "name": "active_power" }
        ]
      }
    }
  ],
  "edges": [],
  "assignments": []
}
```

Listing 7.1. Example Scenario JSON request submitted to `POST /build`.

Listing 7.2. High-level structure of YAML V2 produced by the service.

```
SCEN_CONFIG:
  DAYS: 1
  SCENARIO_NAME: "TestScenario"
  START_DATE: "2024-01-01"
SIM_CONFIG:
  pv_system0:
    RUN_PROCESS:
      cmd: "python pv_system.py %(addr)s"
    MODELS:
      pv_system:
        ATTRS: ["active_power"]
        PARAMS:
          capacity: 5.0
CONNECTIONS: {}
SCEN_OUTPUTS:
  DB:
    step_size: 600
    attrs: []
```

## 7.2 Internal Architecture and Data Resolution Pipeline

The `yaml_builder` microservice operates on a strict separation of concerns. The Scenario JSON received from the frontend acts purely as a lightweight topology map, containing only the structural definition of the graph (nodes and edges) and abstract entity identifiers. To generate a simulation-ready configuration, the `yaml_builder` must dynamically resolve the missing heavy metadata. Internally, the service divides this workload between

two primary components: a **Data Fetcher** layer that retrieves external metadata, and a **YAML Compiler (Mapper)** that merges this data with the frontend topology.

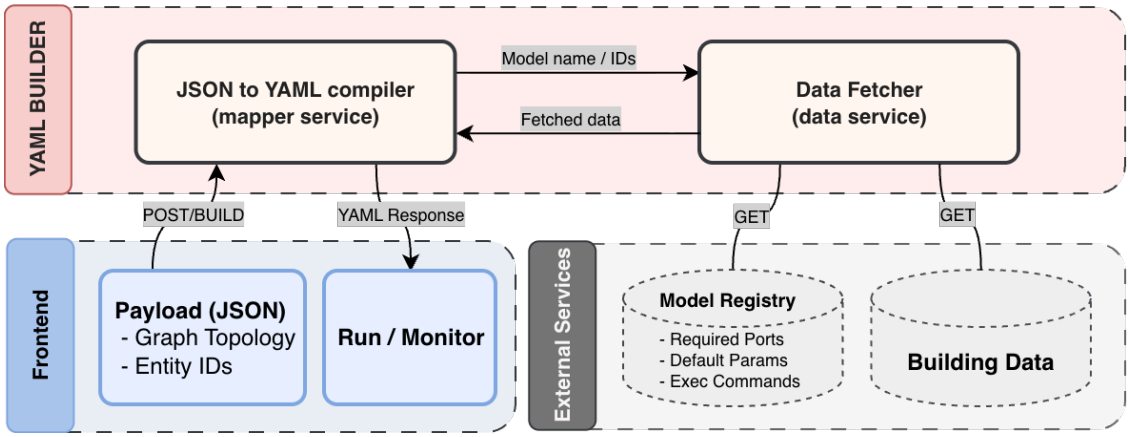


Figure 7.2. Internal architecture of the `yaml_builder`.

### 7.2.1 Payload Parsing and the Data Fetcher Layer

The compilation process begins at the `POST /build` endpoint, where the service parses the incoming JSON payload. The extracted topology fields—specifically `nodes`, `edges`, `assignments`, and `entityIds`—act as triggers for the Data Fetcher layer.

The Data Fetcher interfaces with two external platform services:

- **Model Registry (`app/catalog_client.py`):** The service issues an HTTP GET request to retrieve the full model catalog. From this response, the compiler extracts execution-critical metadata for each referenced node, including port definitions (`input_variables`, `output_variables`), default parameter values, and the `model_execution_cmd` required to launch the simulator.
- **Entity Data Service (`app/data_service.py`):** For each building identifier present in the frontend payload, the service issues an HTTP GET request to the external API. This retrieves specific building attributes (e.g., `net_leased_area`, `height`, `building_type`), which are normalized internally later.

### 7.2.2 The YAML Compiler (Mapper)

Once the external data is fetched, the compilation logic (`app/mapper.py`) executes a five-phase sequence of transformations to merge the contextual metadata with the abstract graph topology:

1. **Model loading:** The service loads the available model catalog (via the Data Fetcher).

2. **Graph analysis:** Non-runtime nodes are excluded, while batch bindings are detected.
3. **Instance expansion:** Bindable models are expanded into concrete execution instances.
4. **Connection resolution:** Edges are converted into engine connections, applying broadcast rules.
5. **Formatting and serialization:** Internal dictionaries are mapped to the YAML structure.

**Instance Expansion and Parameter Injection.** When the compiler encounters an edge flagged as `isBatchToModel`, it reads the corresponding `assignments` array to generate concrete runtime instances (as detailed conceptually in Section 7.5). Conceptually, expansion produces two identifiers: a process-level container (`sim_id`) and an entity-level instance (`model_key`). The number of processes is derived from the authored `numberOfParallelProcesses`; entity identifiers are split into approximately equal chunks and assigned to these containers.

To configure each generated instance, the compiler dynamically resolves model parameters using a strict priority hierarchy:

1. **Entity Properties:** Exact, case-sensitive matches found in the fetched building attributes.
2. **Configuration Aliases:** Mappings defined in an internal `config_parameter_mappings.yaml` file to handle minor schema discrepancies.
3. **Catalog Defaults:** The default values provided by the Model Registry, used only when no entity-specific data is found.

Certain critical model parameters are protected from being overwritten by entity data via a `model_parameter_exclusions` list.

**Connection Resolution.** After instances and parameters are resolved, the compiler maps the connections. Using a rule set defined in `config_connection_types.yaml`, the compiler evaluates tags to determine if an edge requires a standard point-to-point mapping or a complex broadcast topology. The specific rules governing this 1-to-N and N-to-1 topology generation are detailed in Section 7.5.

### 7.2.3 Caching and Operational Resilience

To ensure performance and prevent compilation failures due to external service outages, the `yaml_builder` implements a multi-layered caching and fallback strategy.

- **Model Registry Caching:** Catalog data is stored in an in-memory dictionary with a configurable Time-To-Live (TTL). A background asynchronous task (`_catalog_refresher`) pre-warms this cache at startup and continuously refreshes it, decoupling the heavy catalog fetch from the user's synchronous `/build` request.

- **Resilience and Fallbacks:** If the external Model Registry becomes unreachable, the service falls back to its stale in-memory cache, and subsequently to a persistently saved `registry_fallback_models.json` snapshot on disk.
- **Error Handling:** Unhandled exceptions during compilation are caught and returned to the frontend as `HTTP 400 Bad Request` with diagnostic details, preventing opaque `HTTP 500` server errors from breaking the user workflow.

## 7.3 Input Model: Scenario Graph Contract

The `yaml_builder` consumes a scenario graph serialized from the frontend editor. The input is validated only loosely at the API boundary (accepting a generic JSON dictionary); strict checks are applied during compilation to ensure that the scenario can be rendered into an engine-compatible configuration.

**Top-level fields.** The request payload is centered on four main collections:

- `nodes`: model instances, data sources, and auxiliary graph items (e.g., notes).
- `edges`: directed connections between node ports, encoding data-flow topology.
- `assignments`: batch definitions binding entity identifiers (e.g., buildings) to the scenario.
- `scenarioSettings`: global simulation settings (start date, duration, and step size).

**Node structure.** Each node includes an `id` and a `data` object. For runtime-relevant nodes, `data` includes: (i) the model identifier (`modelDefId`), (ii) a display name, (iii) configuration parameters, and (iv) declared input/output variables that expose the ports available for coupling. This separation reflects the frontend design choice to keep model-specific information schema-driven and to avoid hardcoding model forms.

```
{
  "project": "...",
  "scenarioSettings": {
    "scenarioName": "...",
    "startDate": "...",
    "days": 1,
    "stepSize": 600
  },
  "nodes": [ ... ],
  "edges": [ ... ],
  "assignments": [ ... ]
}
```

Listing 7.3. Minimal Scenario JSON request submitted to `POST /build`.

**Edge semantics.** Edges link ports and carry metadata used during connection resolution. Two edge fields are particularly important for correct engine semantics:

- `data.connectionType`: distinguishes standard connections (`"Same Time"`) from time-shifted connections (`"Next Time"`) used to represent delayed feedback.
- `data.isBatchToModel`: flags edges that encode batch-to-model binding in the editor, enabling entity expansion during compilation.

## 7.4 Time-Shifted Connections and Loop Handling

Urban energy scenarios often include feedback couplings (e.g., controller → plant → controller), which can create algebraic loops if evaluated at the same simulation instant. To support such scenarios in a simple and explicit way, the editor provides a `"Next Time"` connection type, which the compiler maps to a time-shifted link in the engine configuration.

**Representation in the authored graph.** A time-shifted edge is identified by `data.connectionType="NextTime"`. Standard edges use `"Same Time"`.

Feedback couplings are common in energy-system workflows (e.g., controller–plant interactions), but can create algebraic loops if all signals are exchanged at the same simulation instant. Figure 7.3 shows how an explicit `Next Time` connection breaks the instantaneous loop by introducing a one-step delay, preventing an instantaneous algebraic loop and requiring an initial value for the first step.

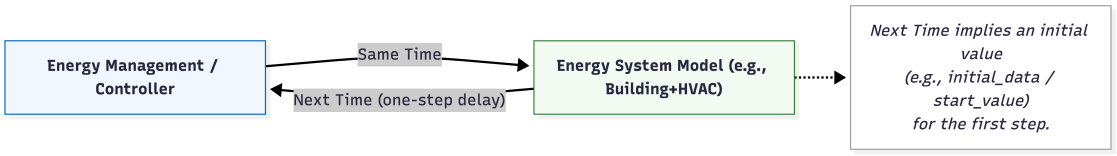


Figure 7.3. Time-shifted loop handling.

**Compilation semantics.** During connection resolution, `"Next Time"` edges are assigned a parameter flag such as `time_shifted: true`. This instructs the engine to use the previous-step value of the source when computing the target at the current step, thereby breaking the instantaneous loop.

**Initialization via `initial_data`.** Time-shifted connections introduce an initial-condition requirement: at the first step, there is no *previous* value to use. To address this, the compiler extracts a `start_value` when available from the source variable metadata and attaches it as `initial_data` within the connection parameters. This keeps loop handling explicit and inspectable in the generated YAML, while avoiding additional manual boilerplate for common feedback patterns.

## 7.5 Broadcasting and Scaling Mechanisms

At district scale, scenario graphs often include *shared* drivers (e.g., weather or schedule time series) and *aggregation* components (e.g., district-level KPIs). If authored naively, connecting a shared input to a large set of entity instances would require repetitive edges and manual graph duplication. To keep the authoring workflow scalable, the `yaml_builder` implements *broadcasting* and *expansion* rules that translate a compact authored graph into an explicit execution-time topology.

**Entity-bindable (bindable) nodes.** In the editor, certain model nodes are *entity-bindable*: they can be bound to a set of entity identifiers through `assignments`. During compilation, each bindable node is expanded into one execution instance per entity (and optionally partitioned across parallel processes). Broadcasting operates on top of this expansion: it replicates authored connections when one endpoint is a shared driver or an aggregation sink.

**Broadcast triggers.** Broadcasting is enabled through tag-based detection. Nodes representing shared time-varying inputs (e.g., tagged as `timeseries`, `weather`, or `schedule`) are treated as broadcast-capable. Similarly, nodes representing aggregation or collection semantics can be treated as sink-like endpoints.

**Broadcast patterns.** Two complementary patterns are supported:

- **1-to-N (driver broadcasting).** If the *source* node is a shared driver, a single authored edge is expanded into connections targeting *all* instantiated entity-level models of the bindable target.
- **N-to-1 (sink aggregation).** If the *target* node is an aggregation sink, all instantiated entity-level sources are connected into the single target instance.

These patterns let users express intent once at the authored-graph level, while the compiler generates the fully explicit engine configuration required for execution and inspection.

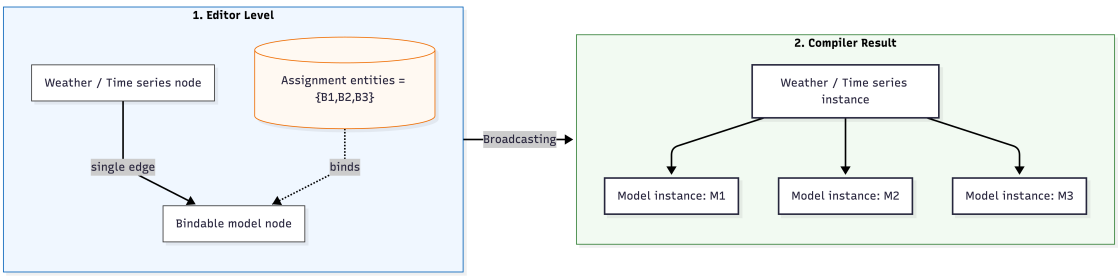


Figure 7.4. Batch expansion and broadcasting.

Table 7.1. Broadcasting expansion rules applied during connection resolution.

| Pattern | Expansion rule (conceptual) |
|---|---|
| 1-to-N (driver broadcasting) | If the source is a shared driver (e.g., weather/time-series), replicate the authored edge to all instantiated targets. |
| N-to-1 (sink aggregation) | If the target is an aggregation/sink component, connect all instantiated sources into the single sink instance. |

Figure 7.4 illustrates the practical effect of these rules: the editor can remain compact (one driver edge, one bindable node), while the emitted configuration contains all concrete links required by the runtime (one per entity instance, or one per source instance for aggregation).

**Parameter scaling from entity attributes.** Beyond topology expansion, the compiler supports *parameter scaling* across entity instances. When entity metadata includes per-entity properties (e.g., floor area, system capacity, retrofit level), the compiler can populate instance parameters automatically when parameter names align with available property keys. This enables a single authored bindable node (e.g., a PV system or a thermal demand model) to instantiate entity-specific configurations without manually editing parameters for each entity.

## 7.6 Validation and Type Coercion

Because the engine configuration must be strict enough to avoid late-stage runtime failures, the `yaml_builder` performs a set of validation and normalization steps during compilation.

**Type coercion for parameters.** Parameters coming from UI forms are often represented as strings (or loosely typed JSON). The compiler therefore applies a coercion function based on declared `data_type` metadata, including: (i) string-to-boolean conversion (e.g., `"true"` → `True`), (ii) parsing list-like strings into Python lists when valid, and (iii) mapping empty strings to `None` for numeric types. When coercion fails, the compiler remains permissive and may fall back to the raw value, allowing the engine to raise a more domain-specific error if needed. This trade-off favors robustness and preserves workflow continuity in the UI, while still providing early catches for common formatting issues.

## 7.7 Output Schemas and Compatibility

The output of the compiler is a YAML configuration that matches the schema expected by the execution engine.

Table 7.2. Examples of parameter normalization performed during compilation.

| Input form | Normalized value |
|---|---|
| `"true"` / `"false"` | Converted to boolean `true`/`false`. |
| Numeric as string (e.g., `"5.0"`) | Parsed to numeric type when the parameter declares `float`/`int`. |
| Empty string for numeric field | Treated as missing/`null` (implementation-dependent) to avoid invalid numeric parsing. |
| List-like strings (e.g., `"[1,2]"`) | Parsed to a list when the declared type expects a list. |

**Schema overview.** The generated YAML is organized into four top-level sections:

- `SCEN_CONFIG`: scenario metadata (name, start date, duration, real-time factor when applicable).
- `SIM_CONFIG`: process definitions and per-model instance configuration, including run commands, exposed attributes, and parameter dictionaries.
- `CONNECTIONS`: an indexed set of connections, each describing `FROM`, `TO`, and the attribute mapping pairs `ATTRS`, optionally with additional parameters (e.g., time shift).
- `SCEN_OUTPUTS`: output persistence and reporting configuration (e.g., database step size and selected attributes).

Conceptually, `SIM_CONFIG` is the result of instance expansion, while `CONNECTIONS` is the result of connection resolution (including broadcasting and time-shift semantics).

**Legacy YAML support and what the platform uses today.** For compatibility reasons, the service can also generate a legacy YAML rendering in addition to the V2 YAML. However, the current system workflow consumes the V2 schema; legacy output is retained primarily to support older configurations, transitional debugging and future migration needs. In practice, this means:

- **The authoritative executable artifact for the current runtime is YAML V2.**
- **Legacy YAML is optional and not required for the current end-to-end execution path.**

# Chapter 8

# Case Study and Demonstration

This chapter provides an end-to-end *workflow demonstration* of the two thesis contributions: (i) the **Coesi UrbanSim Frontend** for interactive scenario authoring and (ii) the `yaml_builder` microservice that compiles user-authored graphs into executable YAML for the co-simulation runtime.

**Scope note.** The simulation engine and most backend services (execution runtime, scenario storage, results serving) are external platform components. Therefore, the objective of this chapter is *not* to validate physical model accuracy or to produce domain conclusions (e.g., "PV saves X%"). Instead, the chapter demonstrates that the UI and compiler can reliably: (1) bind urban entities to model nodes, (2) express typed couplings, (3) generate valid YAML, (4) submit and monitor runs through platform APIs, and (5) retrieve and visualize produced outputs as evidence that the pipeline is operational.

## 8.1 Study Area and Data Sources

### 8.1.1 Study area

The demonstration uses a small neighbourhood dataset in San Salvario, a district in Turin (Italy). In the current prototype, the study area is loaded from an internal GeoJSON feature collection stored in the Frontend codebase. The dataset is encoded in WGS84 (CRS84) and contains 360 building footprints represented as `MultiPolygon` geometries.


Figure 8.1. Study area used in the demonstration (San Salvario, Turin) as displayed in the Coesi UrbanSim Frontend.

### 8.1.2 Entities and identification

In this thesis scope, the primary entities are **buildings**. The Frontend allows users to select one or more building features on the map and group them into a *Batch*. The batch acts as an authoring-time abstraction: users connect and configure a scenario once, then the compiler expands that batch into multiple runtime instances (based on user preferences for distribution of load) during YAML generation.

This mechanism is central to the UI/compilation contribution: it supports multi-entity scenarios without requiring the user to manually replicate nodes and connections for each building.

## 8.2 Scenario Definitions

To keep the demonstration aligned with the thesis scope, the scenario is intentionally simple and designed to exercise the *authoring* and *compilation* features:

- typed connections between models (time series → PV, PV → aggregator),
- entity binding (PV instantiated for a selected batch),
- executable YAML generation and run submission,
- result discovery and plotting in the UI.

### 8.2.1 Demonstration scenario: PV production and aggregation

**Intent.** Demonstrate a minimal end-to-end pipeline where meteorological time series drive PV production and the produced power is aggregated into grid-level indicators by a power-node component.

**Graph nodes.** The scenario uses the following model blocks on the canvas:

- **METEO_TIMESERIES**: a time-series source exposing outputs such as `T_ext` and `ghi`.
- **PV**: a photovoltaic model producing `power_dc` from irradiance and temperature.
- **POWERNODE**: an aggregation node exposing outputs such as `Pload`, `Pprod`, and `Pnet`.
- **BATCH_MAP_X**: a batch binding node representing the selected set of buildings (authoring-time abstraction).

**Typed connections (data edges).**

- METEO_TIMESERIES.`T_ext` → PV.`T_ext`
- METEO_TIMESERIES.`ghi` → PV.`ghi`
- PV.`power_dc` → POWERNODE.`Prod` (*production injection*)

**Entity binding (batch edge).** The PV node is bound to the batch node (dashed binding edge), indicating that PV is instantiated for each building in the selected batch when compiling YAML.

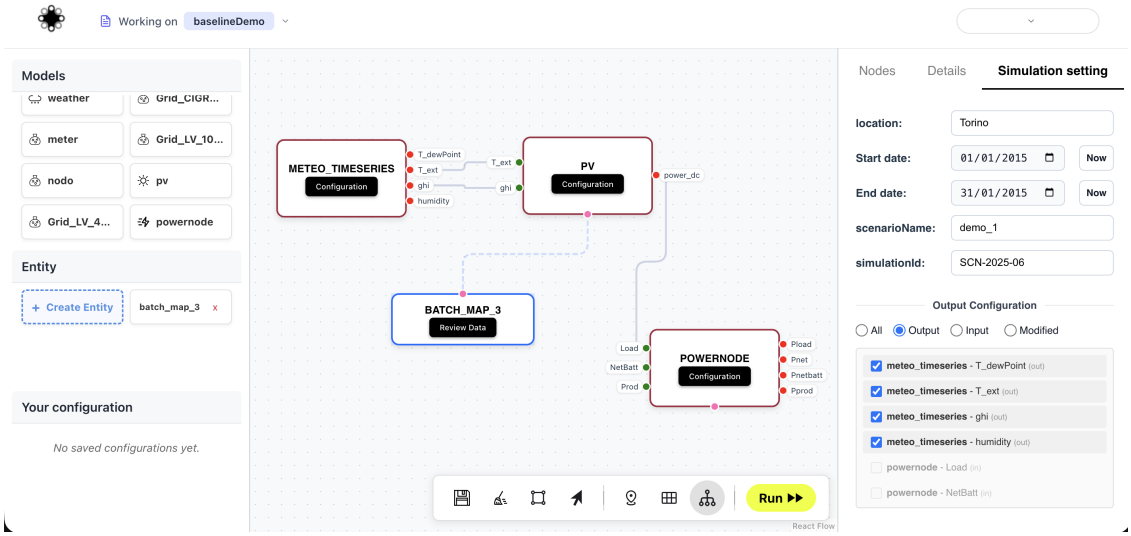


Figure 8.2. Demonstration scenario authored in the Frontend.

**Parameterization.** The scenario uses a small set of parameters to keep the workflow focused. For example, PV can be configured with a nominal system size (e.g., `P_system = 5000` for 5 kWp) and the simulation horizon can be set to a short window (e.g., one day at 15 min resolution) to enable quick iteration during development and demonstration.

## 8.3 End-to-End Workflow Demonstration

This section documents the reproducible workflow from project setup to execution and inspection. The steps instantiate the lifecycle described in Chapter 5, but the evidence here is explicitly focused on the two thesis components (UI and compiler).

### 8.3.1 Service startup and access

The prototype is executed locally using containerized services. The minimal startup steps are:

```
# Build and start services
docker compose up --build coesi_frontend coesi_yaml_builder

# Open the frontend
http://localhost/urbansim
```

### 8.3.2 UI walkthrough

A representative run of the demonstration scenario proceeds as follows:

1. **Open the case-study scenario.** From the dashboard, open the case-study project, select a scenario, and go to the Node Interface. Then, in the left sidebar, use the `Create Entity` button.

2. **Select buildings and create a batch.** In the *Map* view, select a small set of buildings or define a rule (e.g., *height, area, floor, ...*), and click *Create*. The batch name becomes an authoring artifact used later for expansion (e.g., `BATCH_MAP_3`).

3. **Compose the scenario graph.** Drag the batch onto the canvas, add `METEO\_TIMESERIES`, `PV`, and `POWERNODE` nodes, and then create typed connections: `T_ext` and `ghi` into PV, and PV `power_dc` into POWERNODE `Prod`. Finally, bind PV to the batch using a dashed binding edge.

4. **Configure parameters.** Set the PV parameters or accept the default values provided by the model.

5. **Compile to YAML and review the output.** Click *Run*. The Frontend presents a preview of the generated artifacts (scenario JSON/YAML). This step is central to transparency, because users can verify that the compiler expansion matches their intent before submitting a run.

6. **Submit and monitor the run.** Submit the compiled scenario and use the run-monitoring UI to follow the status transitions (queued, running, completed).

7. **Open the results and plot a sanity-check time series.** After completion, open the results viewer and plot one or two series (e.g., PV `power_dc`, or POWERNODE `Pprod`/`Pnet`) to confirm that the outputs are discoverable and can be visualized through the UI.

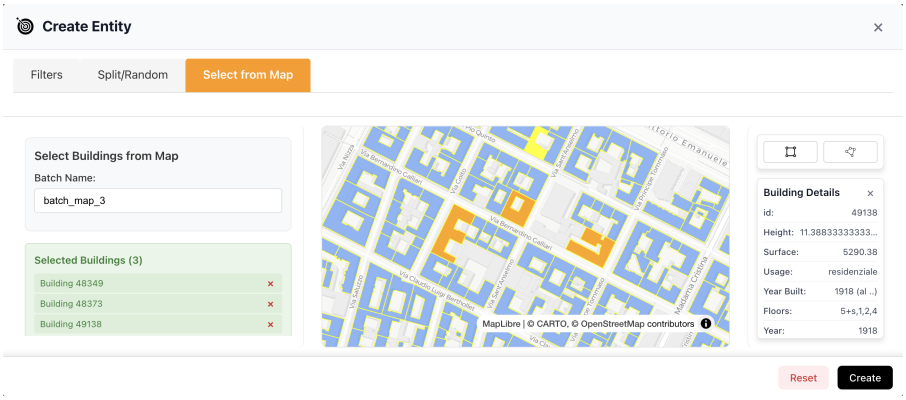


Figure 8.3. Selecting buildings in the map view and creating a batch (authoring-time entity grouping).

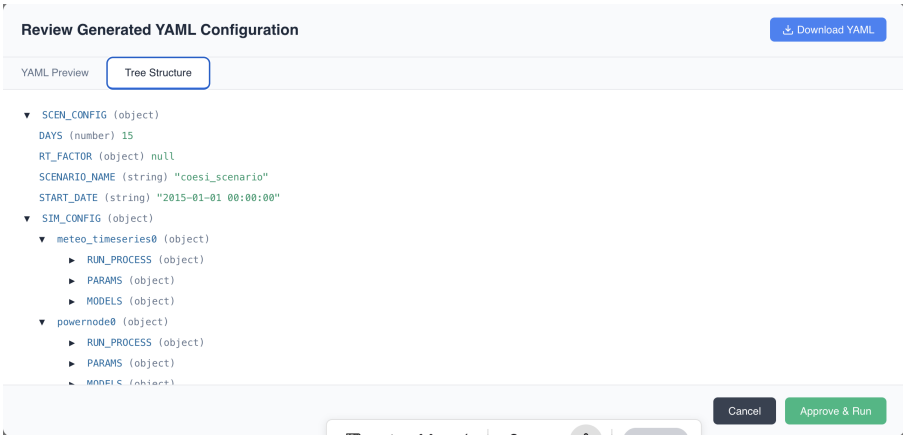


Figure 8.4. YAML/JSON preview shown after clicking “Build YAML” (pre-execution transparency of the compiler output).

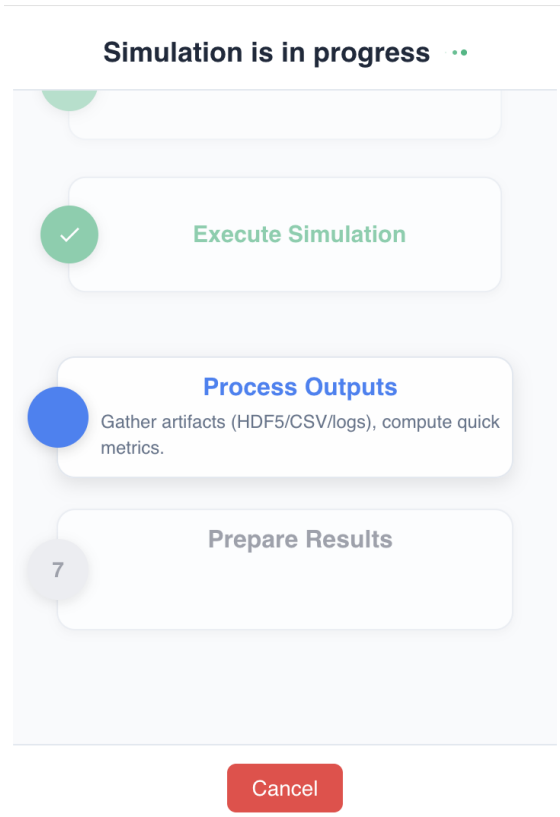


Figure 8.5. Run monitoring view: UI evidence that the compiled scenario is accepted and executed by the external runtime.

### 8.3.3 Time-shifted connections and loop breaking (example)

Time-shifted coupling is introduced in Chapter 2 (Time and synchronization) and its compiler-side handling (including the `time_shifted` flag and `initial_data` seeding) is detailed in Section 7.4. This subsection therefore focuses on the end-to-end *demo interaction*: how a user marks one edge as **Next Time** in the frontend and how that choice is reflected in the compiled YAML.

**Identify the feedback coupling in the graph.** Figure 8.6 shows the bidirectional coupling between a heating model and a building model used in the demonstration. In particular, the heating component produces a return heat-flow signal `Qt_return` that is connected to a building input (e.g., `OthEquRadWatt`), while the building exposes outputs (e.g., `HeatingLoadTarget`) that conceptually influence the heating component.

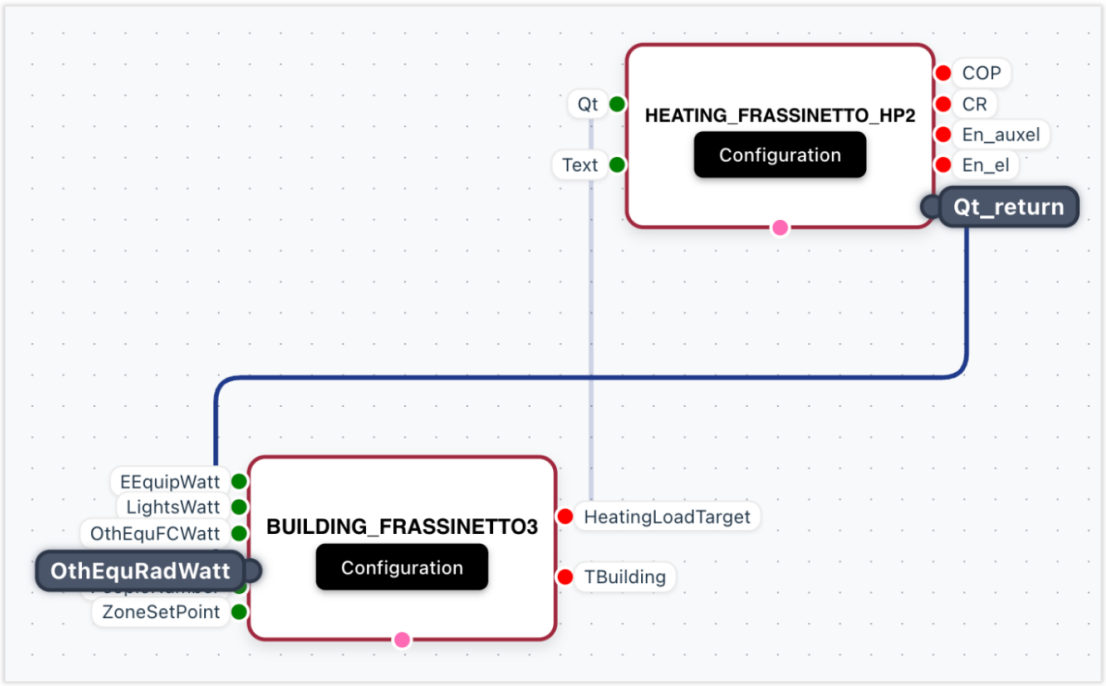


Figure 8.6. Example of a feedback coupling between a heating component and a building component.

**Set the edge type to Next Time in the UI.** In the connection settings modal, the user selects **Next Time** for one direction of the coupling (Fig. 8.7). This introduces an explicit one-step delay on that edge (the value produced at step $t$ is applied at step $t + \Delta t$), making the scenario executable under the platform's time-stepped orchestration model.

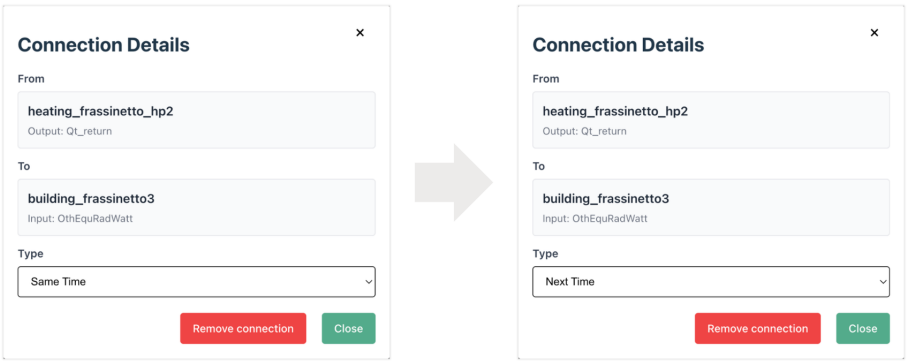


Figure 8.7. Connection configuration in the Frontend. Setting the edge type to **Next Time** explicitly marks the link as time shifted to break same-step feedback loops.

**Verify the mapping in the emitted YAML.** During compilation, `yaml_builder` translates the UI choice into a time-shifted engine connection (`time_shifted: true`) and injects an initial value to seed the first step when required:

```
- from: heating_frassinetto_hp2
  to: building_frassinetto3
  data: { Qt_return: OthEquRadWatt }
  time_shifted: true
  initial_data: { OthEquRadWatt: 0 }
```

The key point for the demonstration is traceability: a single UI toggle (Same Time vs. Next Time) produces an explicit, inspectable change in the executable configuration.

## 8.4 Results Visualization and Inspection

### 8.4.1 Result access in the Frontend

Simulation outputs are stored by the external platform (e.g., as HDF5 `.h5` artifacts). The Frontend retrieves metadata about available series and exposes an interactive viewer where users select an entity (instance) and an attribute to plot.

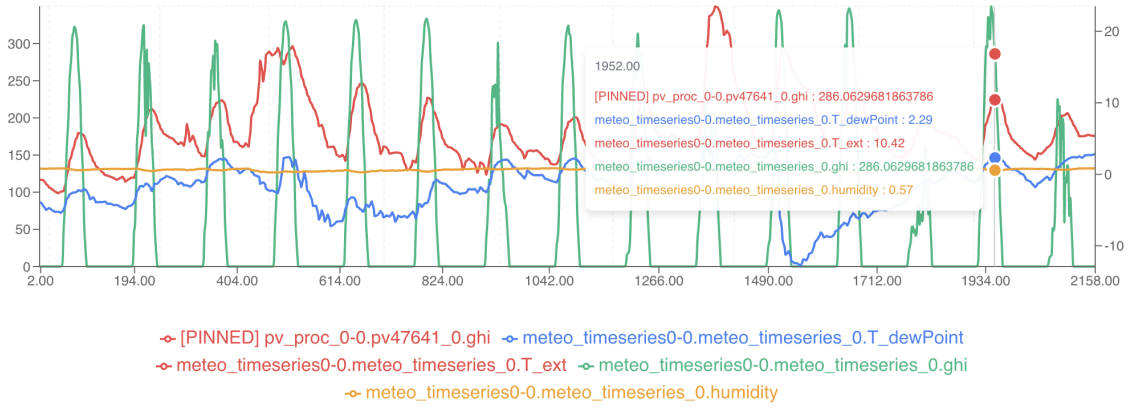


Figure 8.8. Results viewer: selecting an output series (e.g., PV `power_dc` or POWERNODE `Pprod`) and plotting it.

### 8.4.2 Sanity-check plots recommended for the demo

For the PV pipeline scenario, the most defensible plots are those that demonstrate consistent data flow:

- **Irradiance vs PV output:** `ghi` and `power_dc` show correlated daily patterns.
- **Aggregated production:** POWERNODE `Pprod` reflects PV output aggregation.

- **Net signal (if load is present):** POWERNODE `Pnet` demonstrates correct aggregation semantics.

## 8.5 Discussion

### 8.5.1 What the demonstration validates (UI + compiler scope)

The San Salvario demonstration validates the thesis contributions at the *workflow* level:

- **From map to scenario:** urban entities can be selected and grouped into batches that become reusable authoring abstractions.
- **Graph-based scenario authoring:** the node-and-edge canvas supports typed connections and model composition without requiring users to edit YAML directly.
- **Transparent compilation:** the YAML preview step makes the compiler output inspectable before execution.
- **Scalable semantics:** batch expansion and broadcasting reduce repetitive authoring work while preserving traceability.
- **Operational integration:** runs can be submitted and monitored, and outputs can be retrieved and plotted in the UI.

### 8.5.2 Limitations observed in the prototype

- **Static study-area dataset.** The building dataset is currently bundled as a test GeoJSON in the Frontend. A production-grade workflow requires a data ingestion pipeline and persistent project storage.
- **Evaluation depth.** The demonstration provides evidence of executability and usability-oriented design choices, but does not include a controlled usability study with multiple participants and quantitative metrics.
- **External platform dependency.** Execution/runtime behaviour and result formats are determined by external services. The Frontend and compiler must therefore remain robust to API evolution and heterogeneous backend deployments.

Overall, this chapter provides concrete evidence that the proposed Frontend and `yaml_builder` support a practical scenario authoring workflow for urban co-simulation, while keeping the interpretation of physical results outside the thesis scope.

# Chapter 9

# Conclusion and Future Work

This thesis addressed the practical difficulty of configuring urban-scale co-simulation experiments for multi-energy systems: even when the underlying simulators and execution runtime exist, creating a correct, executable scenario definition remains time-consuming and error-prone. The work therefore focused on the *authoring layer* of the platform—making scenario construction more accessible and less fragile—by developing (i) a web-based scenario authoring Frontend tailored to urban entities and (ii) a compilation microservice (`yaml_builder`) that transforms high-level scenario definitions into executable YAML for the co-simulation runtime. The case study in Chapter 8 demonstrated the complete end-to-end workflow on a realistic urban subset.

## 9.1 Summary of Contributions

The main contributions of this thesis can be summarized as follows:

**C1 — A scenario authoring Frontend for urban co-simulation.** A dedicated web UI was designed and implemented to support the full authoring workflow from *urban entity selection* to *graph-based co-simulation composition*. Concretely, the Frontend provides: (i) a map-centric interface to select buildings and group them into batches; (ii) a node-and-edge canvas to compose scenarios using model blocks and typed connections; (iii) parameter configuration and scenario metadata management; (iv) pre-execution feedback mechanisms (validation and preview) that reduce trial-and-error before runs. The result is an interaction model that replaces direct YAML editing with a structured, visual workflow aligned with the scenario lifecycle described in Chapter 5.

**C2 — A compilation microservice to generate executable YAML from high-level scenarios.** The `yaml_builder` microservice was implemented as the bridge between user-level scenario definitions and the execution runtime. Its core functionality is the transformation of a scenario description (graph, entities, parameters) into an executable YAML configuration (YAML v2) with the following key features:

- **Batch expansion:** high-level batches defined in the UI are expanded into concrete simulator instances while preserving a clear naming convention and entity

traceability.

- **Broadcasting support:** shared sources such as weather time series can feed many targets without redundant manual wiring at the UI level.
- **Loop-breaking via time shifting:** when bidirectional couplings create algebraic loops, the compilation step can inject time-shifted links and initial values, producing a well-defined execution order without forcing the user to restructure the graph.

This compilation approach makes the scenario executable while keeping authoring at a higher level of abstraction suitable for non-expert users.

**C3 — A reproducible end-to-end workflow demonstrated on an urban case study.** The thesis integrated the Frontend and `yaml_builder` with the broader platform services through stable APIs, resulting in a complete workflow: select entities, compose the graph, compile YAML, submit a run, and visualize results. Chapter 8 provided evidence of this pipeline using a small district subset in San Salvario, Turin, and a demonstration scenario focused on PV production and aggregation, showing that the approach supports realistic modeling patterns such as multi-instance expansion and shared-input distribution.

## 9.2 Limitations

The thesis contributions intentionally focus on the authoring and compilation layers; several limitations remain, either due to scope boundaries or prototype maturity:

**L1 — Limited breadth of entity types and model coverage.** The demonstrated workflow primarily targets buildings as entities. Extending the approach to additional urban infrastructure (e.g., district heating networks, substations, mobility assets) requires both data-model support and UI abstractions that were outside the current scope. Similarly, the case-study model set is representative rather than exhaustive.

**L2 — Prototype data ingestion and persistence.** The case study used a packaged neighbourhood dataset to ensure reproducibility. A production workflow would require robust ingestion pipelines (geometry, attributes, archetypes, time series), persistent storage of projects/scenarios, and more explicit provenance tracking for inputs and preprocessing assumptions.

**L3 — Evaluation scope (usability and correctness).** While the design is guided by UI/UX principles and the workflow is demonstrated end-to-end, the thesis does not include a full-scale controlled usability study with a diverse participant pool. In addition, the demonstration validates *workflow executability* (i.e., the ability to produce and run scenarios) rather than *model validity* against measured energy data.

**L4 — Scalability and performance at large urban scale.** Batch expansion and broadcasting address conceptual scaling, but the prototype demonstration runs on a small district subset. Performance at thousands of buildings depends on runtime distribution, data throughput, and model computational cost. Systematic benchmarking and optimization at city scale remain future work.

**L5 — Platform dependency boundaries.** Several backend services (registry, scenario storage, runtime execution, result serving) are treated as external dependencies accessed via APIs. Their internal behavior and non-functional guarantees (fault tolerance, scheduling, multi-tenancy) constrain the end-to-end system but are not redesigned in this thesis.

## 9.3 Future Work

The prototype and case study suggest a clear roadmap for advancing both research and engineering aspects of the platform.

### 9.3.1 Usability and human-centered validation

- **Structured usability study:** conduct task-based evaluations with target users (researchers, energy analysts) to measure task completion time, error rates, and perceived workload when authoring scenarios with the Frontend versus manual YAML editing.
- **Iterative UI refinement:** use issue triage from user sessions to improve discoverability, reduce cognitive load in graph composition, and strengthen error explanations (e.g., actionable fixes, link to offending node/edge).
- **Scenario templates and guided wizards:** add reusable templates for common experiment classes (baseline, PV integration, storage, demand response) and wizards that progressively reveal complexity.

### 9.3.2 Richer modeling and semantic correctness

- **Broader model library integration:** extend the registry bindings and UI palette to include additional domains (district heating, thermal storage, control/optimization blocks, grid constraints), enabling more realistic multi-energy studies.
- **Stronger semantic validation:** incorporate richer type systems and constraint checks (units, admissible ranges, required connections, temporal alignment) so that more invalid configurations are caught before compilation and execution.
- **Calibration and external validation workflows:** connect the platform to measured data where available and define standard validation pipelines (e.g., error metrics, sensitivity analysis) to separate workflow correctness from model fidelity.

### 9.3.3 Scalability, robustness, and reproducibility

- **City-scale performance:** benchmark compilation and runtime execution with large batches, introduce incremental compilation (compile only affected subgraphs), and optimize data transfer paths.
- **Distributed execution support:** improve orchestration so that simulator instances can be scheduled across multiple nodes, with appropriate monitoring and fault recovery.
- **Provenance and reproducibility:** store full experiment provenance (input datasets, scenario JSON, generated YAML, runtime versions, and result hashes) to support repeatability and traceability across teams and publications.

### 9.3.4 Productization and platform integration

- **Project persistence and collaboration:** add multi-user project spaces, scenario versioning, and sharing/export mechanisms to support collaborative urban-energy studies.
- **Interoperability:** support import/export from common formats (e.g., CityJSON/CityGML for geometry, standardized time-series formats) and enable integration with external planning tools.
- **Operational hardening:** strengthen authentication/authorization boundaries, logging, and observability so the workflow can be safely deployed beyond research prototypes.

In summary, this thesis demonstrates that a workflow-centered authoring interface combined with an explicit compilation step can substantially reduce friction in configuring urban co-simulation experiments. With further validation, broader model coverage, and scalability engineering, the approach can evolve from a prototype demonstration into a robust platform component for systematic, reproducible urban energy studies.

# Bibliography


[1] Mahmoud M. Abdelrahman, Edgardo Macatulad, Binyu Lei, Matias Quintana, Clayton Miller, and Filip Biljecki. What is a Digital Twin anyway? deriving the definition for the built environment from over 15,000 scientific publications. *Building and Environment*, 274:112748, 2025.

[2] Katherine Arneson, Lijiang Fu, and Lisa Gatzke. Toward more usable, reproducible, and sustainable scientific software: The impact of user-centered design in research software development. In *PASC '25: Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–11. Association for Computing Machinery, 2025.

[3] Silvio Brandi, Pietro Rando Mazzarino, Daniele Salvatore Schiera, Matteo Bilardo, Luca Barbierato, Lorenzo Bottaccioli, Edoardo Patti, and Alfonso Capozzoli. Scenario analysis for renewable energy communities through advanced co-simulation infrastructure. *Applied Energy*, 394:126106, 2025.

[4] John M. Bryson. What to do when stakeholders matter: stakeholder identification and analysis techniques. *Public Management Review*, 6(1):21–53, 2004.

[5] Youssef Elomari, Giorgos Aspetakis, Carles Mateu, Adedamola Shobo, Dieter Boer, M. Marín-Genescà, and Qian Wang. A hybrid data-driven co-simulation approach for enhanced integrations of renewables and thermal storage in building district energy systems. *Journal of Building Engineering*, 104:112405, 2025.

[6] Jose Juan Hernandez, Jose Evora-Gomez, and Jean-Philippe Tavella. Semantic interoperability in co-simulation: use cases and requirements. In *Proceedings of the 30th European Simulation and Modelling Conference (ESM 2016)*, Las Palmas de Gran Canaria, Spain, October 2016.

[7] Tao Hu, Taiping Liu, Venkat Sai Divyacharan Jarugumalli, Samuel Cheng, and Chengbin Deng. FAIR principles in workflows: A GIScience workflow management system for reproducible and replicable studies. *International Journal of Applied Earth Observation and Geoinformation*, 138:104477, 2025.

[8] International Organization for Standardization. ISO 9241-210:2019 — ergonomics of human-system interaction — part 210: Human-centred design for interactive systems, 2019. ISO standard overview page.

[9] Catriona Macaulay, David Sloan, Xinyi Jiang, Paula Forbes, Scott Loynton, Jason R. Swedlow, and Peter Gregor. Usability and user-centered design in scientific software development. *IEEE Software*, 26(1):96–102, 2009.

[10] Chiara Nardelli, Riccardo Colombo, Alessia Banfi, Martina Ferrando, Xing Shi, and Francesco Causone. A comparative analysis of two urban building energy modelling

tools via the case study of an italian neighbourhood. *Energies*, 18(10):2618, 2025.

[11] Jakob Nielsen. 10 usability heuristics for user interface design, April 1994. Nielsen Norman Group article.

[12] Dario Niermann, Tobias Doernbach, Christoph Petzoldt, Melvin Isken, and Michael Freitag. Software framework concept with visual programming and digital twin for intuitive process creation with multiple robotic systems. *Robotics and Computer-Integrated Manufacturing*, 82:102536, 2023.

[13] Francisco Queiroz, Raniere Silva, Jonah Miller, Sandor Brockhauser, and Hans Fangohr. Good usability practices in scientific software development. In *Proceedings of the Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE5.1)*, pages 1–6. figshare, 2017.

[14] Pietro Rando Mazzarino, Martina Capone, Elisa Guelpa, Lorenzo Bottaccioli, Vittorio Verda, and Edoardo Patti. A modular co-simulation platform for comparing flexibility solutions in district heating under variable operating conditions. *IEEE Transactions on Sustainable Computing*, 10(2):408–417, 2025.

[15] Pietro Rando Mazzarino, Salvatore Finocchiaro, Luca Barbierato, Daniele Salvatore Schiera, Lorenzo Bottaccioli, and Edoardo Patti. An automated tool for urban building energy modelling: from sparse datasets to CityJSON. In *2024 IEEE International Conference on Environment and Electrical Engineering and 2024 IEEE Industrial and Commercial Power Systems Europe (EEEIC / I&CPS Europe)*, pages 1–7. IEEE, 2024.

[16] Pietro Rando Mazzarino, Alberto Macii, Lorenzo Bottaccioli, and Edoardo Patti. A Multi-Agent framework for smart grid simulations: strategies for power-to-heat flexibility management in residential context. *Sustainable Energy, Grids and Networks*, 34:101072, 2023.

[17] Mark S. Reed, Anil Graves, Norman Dandy, Helena Posthumus, Klaus Hubacek, Joe Morris, Christina Prell, Claire H. Quinn, and Lindsay C. Stringer. Who's in and why? a typology of stakeholder analysis methods for natural resource management. *Journal of Environmental Management*, 90(5):1933–1949, 2009.

[18] Felix Rehmann, Martín Mosteiro-Romero, Clayton Miller, and Rita Streblow. Enhancing urban energy modeling: a case study of data acquisition, enrichment, and evaluation in berlin. *Energy and Buildings*, 346:116070, 2025.

[19] Giuseppe Russo, Laura Pompei, Giovanni Francesco Giuzio, Gabriele Umberto Magni, Daniele Groppi, Gianfranco Cipolla, Francesca Vecchi, Roberto Stasi, Simona Semeraro, Davide Astiaso Garcia, Umberto Berardi, and Annamaria Buonomano. Modelling the complexity of interconnected energy systems at different urban scales: a critical review. *Renewable and Sustainable Energy Reviews*, 223:116007, 2025.

[20] Daniele Salvatore Schiera, Luca Barbierato, Andrea Lanzini, Romano Borchiellini, Enrico Pons, Ettore Bompard, Edoardo Patti, Enrico Macii, and Lorenzo Bottaccioli. A distributed multimodel platform to cosimulate multienergy systems in smart buildings. *IEEE Transactions on Industry Applications*, 57(5):4428–4440, 2021.

[21] Daniele Salvatore Schiera, Silvio Brandi, Fulvio Corno, Marco Martini, Pietro Rando Mazzarino, Francesco Demetrio Minuto, Enrico Pons, and Edoardo Patti. Streamlined deployment of a distributed and scalable co-simulation platform for

integrated energy systems. In *2024 20th International Conference on Intelligent System Application to Power Systems (ISAP)*, pages 1–6. IEEE, 2024.

[22] Daniele Salvatore Schiera, Francesco Demetrio Minuto, Lorenzo Bottaccioli, Romano Borchiellini, and Andrea Lanzini. Analysis of rooftop photovoltaics diffusion in energy community buildings by a novel GIS- and agent-based modeling co-simulation platform. *IEEE Access*, 7:93404–93432, 2019.

[23] Peter Sun and John A. Marohn. mmodel: A workflow framework to accelerate the development of experimental simulations. *The Journal of Chemical Physics*, 159(4):044801, 2023.

[24] Matia Torlini, Pietro Rando Mazzarino, Daniele Salvatore Schiera, Luca Barbierato, Lorenzo Bottaccioli, and Edoardo Patti. An ontology-based methodology to assist complex energy co-simulation setup. In *2025 IEEE International Conference on Environment and Electrical Engineering and 2025 IEEE Industrial and Commercial Power Systems Europe (EEEIC / I&CPS Europe)*, pages 1–6. IEEE, 2025.

[25] Maarten van Emmerik and Esther Waardenburg. A web-based visual programming environment for design and operation of digital twins. *Digital Twin*, 2025.

[26] Jingfeng Zhou, Jiantong Li, Jiayu Xie, Xinqiao Dong, Kaixuan Wang, Rui Jing, Rui Tang, and Meng Wang. State-of-the-art review of urban building energy modelling on supporting sustainable development goals. *Applied Energy*, 402:126924, 2025.