OpenCLD: A Scalable Python Library for System Dynamics Modelling

Filippo Maria Ottaviani[a], Giovanni Zenezini[a,], Andrea Ferrari[a], Massimo Rebuglio[a], Pietro Viero[a], Filippo Incerti[a], Ali Vaezi[b], Mustafa Ozan Nesanir[c]

[a]*Department of Management and Production Engineering, Politecnico di Torino, Corso Duca degli Abruzzi, 24, Turin, 10129, Italy*
[b]*Interuniversity Department of Regional and Urban Studies and Planning, Politecnico di Torino, Corso Duca degli Abruzzi, 24, Turin, 10129, Italy*
[c]*Department of Industrial Engineering, Marmara University, , Maltepe, 34854, Turkey*

Abstract

The practical application of System Dynamics (SD) modeling is constrained by proprietary software that is expensive, inflexible, and poorly integrated with modern data science ecosystems. Existing open-source alternatives often lack the features and native development capabilities required for contemporary analytical workflows. This paper introduces OpenCLD, a lightweight, open-source Python library designed to bridge this gap by providing a native and intuitive framework for SD modeling. Built on object-oriented principles, OpenCLD enables the construction of complex models using standard SD components while ensuring seamless compatibility with the scientific Python ecosystem, including pandas, scikit-learn, and TensorFlow. The library supports modern development practices such as interactive modeling in Jupyter notebooks, containerized deployment, and integration with machine learning pipelines. OpenCLD democratizes system dynamics, offering a powerful and accessible alternative for developing transparent, reproducible, and scalable models.

Keywords: System Dynamics, Python, Open Source Software, Simulation Modeling

Email addresses: filippo.ottaviani@polito.it (Filippo Maria Ottaviani), giovanni.zenezini@polito.it (Giovanni Zenezini), andrea.ferrari@polito.it (Andrea Ferrari), massimo.rebuglio@polito.it (Massimo Rebuglio), pietro.viero@studenti.polito.it (Pietro Viero), filippo.incerti@studenti.polito.it (Filippo Incerti), ali.vaezi@studenti.polito.it (Ali Vaezi), moznesanir@gmail.com (Mustafa Ozan Nesanir)

1. Introduction

Feedback-rich socio-technical systems generate policy resistance and counterintuitive outcomes, posing a persistent problem for decision-makers [1, 2]. These dynamics arise in supply chains and population health, where disruptions reveal interdependencies, delays, and nonlinear effects [3, 4].

System Dynamics (SD), pioneered by Forrester, models stocks, flows, and feedback to explain behaviors that linear methods miss [5, 6]. Across corporate strategy, public policy, environmental management, and healthcare, SD has delivered insights and decision support [7].

Despite advances, SD practice remains anchored to proprietary, monolithic tools that conflict with Open Source, programmatic, and reproducible workflows common in the Python ecosystem [8, 9, 10, 11, 12]. Vendor APIs, exchange formats, and translators have yielded only marginal connectivity, with assessments reporting brittle integration, opaque data flows, and limited reproducibility [13, 14]. Licensing and vendor lock-in further constrain access, hinder benchmarking, and slow innovation [15, 8, 9, 10]. The literature still lacks a fully Open Source SD framework with a transparent intermediate representation, first-class interfaces to streaming data and machine-learning pipelines, and end-to-end FAIR workflows at scale [16].

The study addresses this methodology–infrastructure gap by introducing OpenCLD, an open-source Python library aligned with modern, data-centric practice. The objectives are to provide a native, object-oriented framework for building, simulating, and analyzing SD models with programmatic validation, transparency, and reproducibility, and to enable seamless integration with data-science and machine-learning pipelines. Providing a benchmark against other libraries is out of scope of this first release. The main reason for this statement is that this library was not developed to be faster than existing proprietary software or Python libraries in terms of model development and deployment; instead, it fills the above mentioned technical gap by offering a programmable alternative to proprietary, GUI-based tools. Hence, our intended target user is a System Dynamicist with prior expertise in Python programming, who wants to integrate System Dynamics with other Python-based applications (e.g. Large Language Models, Machine Learning, Deep Learning, Reinforcement Learning, Agent-based modeling etc.).

This technical paper opens with a motivation grounded between System Dynamics practice and modern computational science, supported by a landscape survey and comparison table that position OpenCLD relative to existing tools. Section 3 is the technical core, walking through the object-oriented architecture class by class with enough design rationale that a Python-literate reader could begin building models. Section 4 validates the library through three canonical SD models used as structured demonstrations of transparency, dimensional consistency, and extensibility rather than scientific contributions in themselves. Section 5 then scales the validation to three applied hybrid SD–ML workflows spanning flood control, supply chain optimization, and smart microgrid control, each targeting a different integration mode between the SD layer and an AI component. Finally, section 6 closes the paper by articulating the "Model-as-Code" paradigm, as well as providing the limitations and future work.

2. Related Work

A central paradox emerges: System Dynamics (SD) fits today's feedback-rich problems, yet adoption is limited by tooling that does not support Open Source, interoperable, and reproducible workflows [17, 18, 19]. This section reviews assesses commercial and Open Source tools to identify capability and workflow gaps in data-intensive settings [20, 21, 22, 23].

2.1. The Legacy Software Ecosystem and Its Misalignment with Modern Data Science

Commercial SD platforms such as Vensim, Stella, and Powersim broadened access through GUI-centric environments but rely on closed, monolithic architectures that limit transparency and integration, undermining reproducible analysis [24, 25, 26, 27]. Proprietary formats and constrained APIs impede links to external data sources, cloud analytics, and machine-learning pipelines, yielding brittle automation and restricted data interchange [28, 29, 30]. These design choices predate contemporary practice and conflict with Open Source, interoperable, and reproducible workflows [26].

Licensing costs in the thousands of dollars per user restrict access in education and research and conflict with Open Source science norms [31, 32]. GUI-first, binary model files fit poorly with version control, code review, and containerized, CI-based deployment, undermining collaboration and independent verification [33, 34]. These constraints impede comparative benchmarking and external replication in data-intensive settings.

2.2. The Open Source Response and Persistent Gaps

Open Source efforts address access and transparency but do not resolve core architectural and workflow misalignments with modern, Python-centric practice [35, 36]. PySD translates Vensim and Stella models into Python for execution and analysis but does not provide native model construction or Python-first workflows [37, 38]. Model creation stays in proprietary GUIs, which fragments development and blocks full adoption of testing, packaging, and reuse in Python [39, 40].

Other options exhibit different but consequential limitations. BPTK-Py offers a native, object-oriented approach yet suffers from sparse documentation and a smaller user community [41]. Simantics requires heavyweight installation and provides non-native Python integration, which slows agile, data-driven work [42]. R's deSolve supplies robust solvers but lacks SD abstractions, shifting model architecture burdens to users [43].

Comparative reviews find that no framework combines native Python integration, feature breadth, and usability required for end-to-end, reproducible SD workflows [44]. Table 1 summarizes the current software landscape.

Table 1: Comparison of System Dynamics Software Tools and Frameworks

Type/Tool	Real-time Simulation	Sensitivity Analysis	ML Integration	Cloud-based	API
Proprietary					
Vensim Pro	✗	✓[a]	✗	∼	∼
Stella Pro	✗	✓[a]	✗	✗	✗
Powersim	∼	✓[a]	✗	✓	∼
AnyLogic Pro	✓	✓[a]	∼	✓	✓
Ventity	∼	✓[a]	∼	✗	∼
Simile	∼	∼	∼	✗	∼
GoldSim	∼	✓[a]	∼	✗	∼
Stella Architect	✓	✓[a]	∼	✓	✓
PathFwd	✓	∼	∼	✓	∼
Sysdea	∼	∼	✗	✓	✗
Freeware					
Vensim PLE	✗	∼	✗	✗	✗
AnyLogic PLE	✓	∼	∼	✗	✓
Stella Online	∼	✗	✗	✓	✗
Insight Maker	∼	✓[a]	∼	✓	∼
FOSS					
PySD	✗	✓[b]	∼	∼	✓
BPTK-Py	✗	✓[b]	✓	∼	✓
Simantics	✗	∼	✗	✗	∼
NetLogo	∼	✓[b]	✗	✗	∼
deSolve	✗	✓[b]	∼	∼	✓
Simlin	∼	∼	✓	✓	✓
SDEverywhere	∼	✓[b]	✓	✓	✓
StochSD	∼	✓[b]	∼	✓	∼
OpenCLD	✓	✓[b]	✓	✓	✓

Legend: ✓ = Yes;
✗ = No;
∼ = Limited/Basic/Manual (as applicable)
Notes: [a] GUI-driven sensitivity/experiments
[b] Programmatic sensitivity via scientific libraries

2.3. Research Gap

The literature shows that prevailing tools are misaligned with Open Source, interoperable, and reproducible workflows required in modern computational science [45, 46, 47]. Open Source initiatives have improved access, but most stop short of native, end-to-end integration with Python-centered practice [48, 49, 50]. This misalignment prevents routine use of machine-learning pipelines, containerized execution, and cloud-native services that now define data-intensive research [51, 52, 53].

The gap manifests along three dimensions:

- Integration gap: Traditional tools sit outside Python-first data ecosystems, limiting real-time ingestion, automated calibration, and ML coupling [48, 49, 50].
- Reproducibility gap: Proprietary, binary formats and GUI-first workflows impede archiving, sharing, and independent verification [51, 52].
- Collaboration gap: Incompatibility with version control, continuous integration, and containerized deployment hinders collaborative development and benchmarking [53].

3. OpenCLD Library

OpenCLD is an open-source Python library for native System Dynamics modeling. Its design follows a single guiding principle: every SD concept should map directly to a Python object with a single, well-defined responsibility.

The primary contributions of OpenCLD are as follows:

1. Provide built-in validation and verification, including automated detection of algebraic loops via topological sorting and enforcement of dimensional consistency with Pint.
2. Develop automated structural analysis that discovers feedback loops and determines polarity via a numerical perturbation method, with integrated tools to visualize the causal structure.
3. Enable integration with the modern data-science ecosystem through native export of simulation results as pandas DataFrames (wide and tidy) for downstream analysis and interoperability.

This section describes the library's architecture (Section 3.1), its core component classes (Sections 3.2–3.6), the central `Model` engine (Section 3.7), and the visualization toolkit (Section 3.8). Detailed API references, installation instructions, and worked examples are available in the package documentation on PyPI [1].

3.1. Library Design

OpenCLD uses an object-oriented architecture in which each SD primitive (stock, flow, auxiliary, parameter, table) is its own class with a small, focused interface (Table 2). Keeping responsibilities separated makes individual classes easier to test and reuse, and it allows the `Model` engine to compose them without tight coupling. Two cross-cutting concerns are handled at the library level rather than inside each class: dimensional consistency, enforced at runtime through Pint `Quantity` objects, and performance, achieved by storing all components in name-keyed dictionaries for $O(1)$ lookup during the simulation loop. Visualization dependencies are loaded only on demand, so the core simulation stack stays lightweight. The design scales from single deterministic runs to large batch Monte Carlo experiments without any change to the modeling interface.

[1] https://pypi.org/project/opencld/

Table 2: OpenCLD core class architecture.

Class	Responsibility	Key attributes	Key methods
`Model`	Central simulation engine: compiles components, validates dimensional consistency, advances time with Euler or RK4, records history, and performs structural analysis.	`stocks`, `flows`, `auxiliaries`, `parameters`, `timestep`, `history`	`run()`, `step()`, `get_results()`, `to_df()`, `print_loops()`
`Stock`	Accumulation (level) updated from net inflows/outflows; supports scalar or vector state.	`value`, `initial_value`, `unit`, `dimensions`, `inflows`	`update()`, `add_inflow()`, `get_value()`
`Flow`	Rate computation connecting stocks; wraps outputs into quantities and checks dimensionality.	`rate`, `rate_function`, `unit`, `source_stock`, `target_stock`	`calculate_rate()`, `get_rate()`
`Auxiliary`	Computed intermediate variable; wraps outputs into quantities and cleans NaNs.	`value`, `calculation_function`, `unit`, `inputs`	`calculate_value()`, `get_value()`
`Parameter`	Constant input (scalar or vector), stored as a unit-aware quantity.	`value`, `unit`, `description`, `dimensions`	`get_value()`, `format_value()`
`Table`	Piecewise-linear lookup (`np.interp`) for nonlinear relationships with boundary handling.	`x_values`, `y_values`, `name`	`lookup()`, `__call__()`
`Plotting`	Static plotting utilities for time series, Monte Carlo, and structure graphs.	– (static namespace)	`plot_timeseries()`, `plot_variable_facets()`
`UnitManager`	Singleton unit system built on Pint; loads unit definitions and exposes shared registry.	`ureg`, `Q_`	`get_quantity()`, `batch_define()`, `force_quantity()`

3.2. Stock Class

The `Stock` class is the memory of the system: it accumulates the net effect of all connected flows over time, tracking quantities that cannot change instantaneously. Its value is always a unit-aware `pint.Quantity`, so the library can verify at initialization that every connected flow carries compatible units of the form [stock unit]/[time], catching dimensional errors before any simulation runs. Stocks support both scalar and vector initial values, which allows a single `Stock` instance to represent disaggregated populations (e.g., by age cohort or geographic region) without duplicating the model structure.

3.3. Parameter Class

The `Parameter` class holds a constant, named input (scalar or vector) that flows and auxiliaries read but never modify. By storing every value as a `pint.Quantity`, Parameters participate in the same dimensional-consistency checks as all other components, preventing unitless constants from silently corrupting calculations. This design also makes sensitivity and calibration studies straightforward: external optimization routines can update parameter values between `run()` calls without touching any model logic.

3.4. Auxiliary Class

The `Auxiliary` class encapsulates intermediate computed quantities: variables that depend on the current state of stocks, parameters, or other auxiliaries, but do not accumulate over time. Its key architectural role is to carry an explicit `inputs` list that declares which

upstream variables the calculation uses; this declaration feeds the dependency graph that the `Model` engine uses to sort auxiliaries into a valid evaluation order and to detect algebraic loops before simulation begins. Auxiliaries also serve as the primary attachment points for external data or machine-learning predictions, since any Python callable can act as the calculation function.

3.5. Flow Class

The `Flow` class represents a rate of transfer between stocks, the primary mechanism through which stocks change. Like auxiliaries, flows declare their upstream `inputs` explicitly, enabling the library to build a complete causal graph of the model's feedback structure. Flows are connected to their source and target stocks at construction, and the library enforces that the declared unit is dimensionally compatible with [stock unit]/[time], preventing mass-balance violations at the point of definition rather than at runtime.

3.6. Table Class

The `Table` class approximates nonlinear relationships through piecewise-linear interpolation over user-supplied breakpoints, returning boundary values when queried outside the defined range. It is designed for cases where a relationship is known empirically rather than analytically; for example, a lookup table mapping reservoir volume to surface area. Overloading the call operator means a `Table` instance can be used directly inside any flow or auxiliary function, keeping model equations readable.

3.7. Model Class

The `Model` class is the simulation engine that brings all components together. Its design philosophy is that model integrity should be verified automatically at construction, so that a `Model` that initializes without error is guaranteed to be structurally and dimensionally sound. The class manages the full lifecycle from compilation through execution and output formatting, and exposes a single-step interface that allows external agents (e.g., reinforcement-learning controllers) to intervene at each tick.

3.7.1. Initialization and Model Compilation

At construction, the `Model` engine performs three sequential validation stages before the simulation can run. First, it converts all component lists into name-keyed dictionaries and resolves the evaluation order of auxiliaries by performing a topological sort on their dependency graph, a process that simultaneously detects any algebraic loops and raises an informative error. Second, it runs a unit-consistency check across every stock–flow pair and executes a dry-run calculation of all components at $t = 0$ to surface errors in user-defined functions early. Third, it computes the full initial state, records it as the first row of the simulation history, and runs the automated structural analysis that identifies all feedback loops and their polarities, providing a complete picture of the model's causal architecture before a single timestep is taken.

3.7.2. Simulation Loop

The `run()` method is re-entrant by design: it resets all state before executing, so successive calls on the same `Model` instance produce independent results without re-instantiation. Internally, it delegates each discrete advance to `step()`, which evaluates auxiliaries, computes flow rates, and integrates stock levels in the correct dependency order. The library supports both first-order Euler and fourth-order Runge–Kutta integration; the choice trades computational cost against numerical accuracy for stiff or rapidly varying systems. Exposing `step()` as a public method is a deliberate design choice: it enables the tight hybrid SD–AI loops demonstrated in Section 5.

3.7.3. Data Retrieval and Formatting

Simulation output is available in two formats to fit different downstream workflows. The `get_results()` method returns a wide-format `pandas.DataFrame` with one column per model variable, convenient for quick inspection and plotting. The `to_df()` method transforms this into a tidy long-format frame with columns `[time, run_id, variable, value, type]`, which is the standard input expected by most statistical and visualization libraries and is essential for aggregating multi-run Monte Carlo experiments.

3.7.4. Structural Analysis and Reporting

After initialization, `print_relationships()` and `print_loops()` expose the model's feedback architecture as human-readable text. The former lists every causal link with its computed polarity; the latter groups all detected cycles into reinforcing, balancing, and neutral categories. This built-in transparency is a direct response to the "black-box" critique of proprietary SD tools: the structural analysis is derived automatically from the same code that runs the simulation, so the reported structure always matches the model as executed.

3.7.5. Polarity Detection Algorithm

Link polarity (whether increasing a cause increases or decreases its effect) is determined by a numerical perturbation algorithm. For direct flow-to-stock links, polarity is resolved structurally (an inflow is always positive; an outflow is always negative) without computation. For all other links, the algorithm perturbs the upstream variable by a small relative amount (0.1% of its value at $t = 0$), propagates the change through all dependent variables, and reads the sign of the resulting output shift. The relative perturbation scale ensures correct behavior across variables of widely different magnitudes. For vectorized (array-valued) variables, the total signed change is summed across all elements before thresholding, enabling the same algorithm to handle both scalar and disaggregated models without modification.

3.7.6. Unit Management

A singleton `UnitManager` instance, built on the Pint library, is created at import time and shared across all classes, so every stock, flow, auxiliary, and parameter references the same unit registry. This global registry approach eliminates the risk of unit mismatches that arise when different components define their own conversion factors independently. Users can extend the registry with custom or domain-specific units by placing a `units.txt` file in the working directory; if none is present, the library falls back to its packaged unit definitions.

3.8. Plotting Class

The `Plotting` class is a stateless namespace of visualization utilities, deliberately decoupled from the `Model` engine. All methods operate on standard `pandas.DataFrame` inputs, either wide-format output from `get_results()` or tidy-format output from `to_df()`, so they can be composed freely with any third-party analysis tool.

3.8.1. Time-Series Visualization

The `plot_timeseries` method renders the behavior of selected model variables over time as a standard line graph. It is the primary tool for single-run output inspection and accepts the same DataFrame format produced by `get_results()`, requiring no additional data transformation.

3.8.2. Visualization of Uncertainty

Two methods support multi-run (Monte Carlo) analysis. `plot_alpha_density_lines` overlays all individual run trajectories for a chosen variable as semi-transparent lines, creating a density map ("spaghetti plot") that communicates the full range of possible behaviors; the median trajectory is superimposed as a summary. `plot_variable_facets` extends this to the entire model, generating a grid of subplots (one per dynamic variable), so the propagation of uncertainty across interconnected stocks, flows, and auxiliaries can be assessed in a single view.

4. OpenCLD library testing: addressing existing limitations

This section demonstrates how `OpenCLD`'s design directly addresses the limitations of existing software tools identified in Chapter 2. We use classic System Dynamics examples, Lotka-Volterra, SIR, and Bass Diffusion, to illustrate the library's three core pillars: declarative syntax, dimensional consistency, and structural modularity.

4.1. Addressing the "Black-Box" Model Gap

`OpenCLD` adopts a "Model-as-Code" paradigm where the model structure is defined in transparent, human-readable Python.

Using the classical Lotka-Volterra (Predator-Prey) system as an example, we show how Stocks and Flows are declared explicitly. This code is not a generated artifact but the source of truth, fully version-controllable (e.g., with Git) and inspectable by any researcher without a commercial license.

This approach ensures that the model structure, the feedback loops driving the oscillations seen in Figure 1, is completely transparent. Furthermore, `OpenCLD` provides built-in automated structural analysis. As demonstrated in Listing 1 and Listing 2, the library can automatically detect and print all causal relationships, determine link polarities via numerical perturbation, and identify all reinforcing and balancing feedback loops directly from the code structure.

```
--- Model Relationships and Link Polarities (@ t=0) ---
  Predator Birth   -> Predators        : (+)
  Predator Death   -> Predators        : (-)
  Predators        -> Predator Death   : (+)
```

```
  Predators          -> Prey Eaten        : (+)
  Prey Birth         -> Preys             : (+)
  Prey Death         -> Preys             : (-)
  Prey Eaten         -> Predator Birth    : (+)
  Prey Eaten         -> Prey Death        : (+)
  Preys              -> Prey Birth        : (+)
  Preys              -> Prey Eaten        : (+)
  alpha         (P)  -> Prey Birth        : (+)
  beta          (P)  -> Prey Eaten        : (+)
  delta         (P)  -> Predator Birth    : (+)
  gamma         (P)  -> Predator Death    : (+)

  Note: Polarity (+,-,0,?) calculated via numerical perturbation at t=0.
```

Listing 1: Automated detection of model relationships and link polarities for the Lotka-Volterra model.

```
--- Detected Feedback Loops (Polarity @ t=0) ---

  Reinforcing Loops (+):
    1. Prey Birth -> Preys -> Prey Birth
    2. Prey Eaten -> Predator Birth -> Predators -> Prey Eaten

  Balancing Loops (-):
    1. Predators -> Predator Death -> Predators
    2. Prey Eaten -> Prey Death -> Preys -> Prey Eaten

  Neutral Loops (0 @ t=0):
    None found.

  Ambiguous Loops (?):
    None found.
```

Listing 2: Automated detection of Reinforcing and Balancing feedback loops for the Lotka-Volterra model.

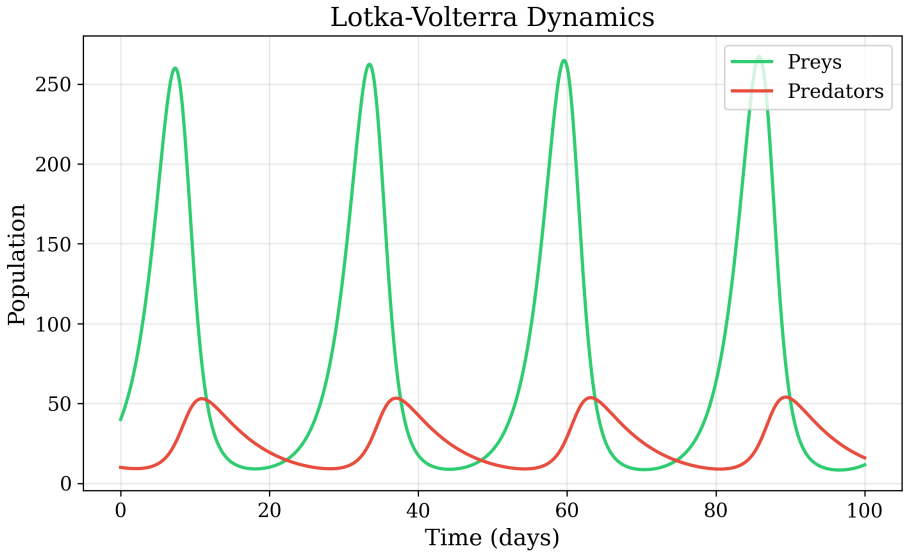

Figure 1: Lotka-Volterra Population Dynamics: Oscillations between Preys and Predators.

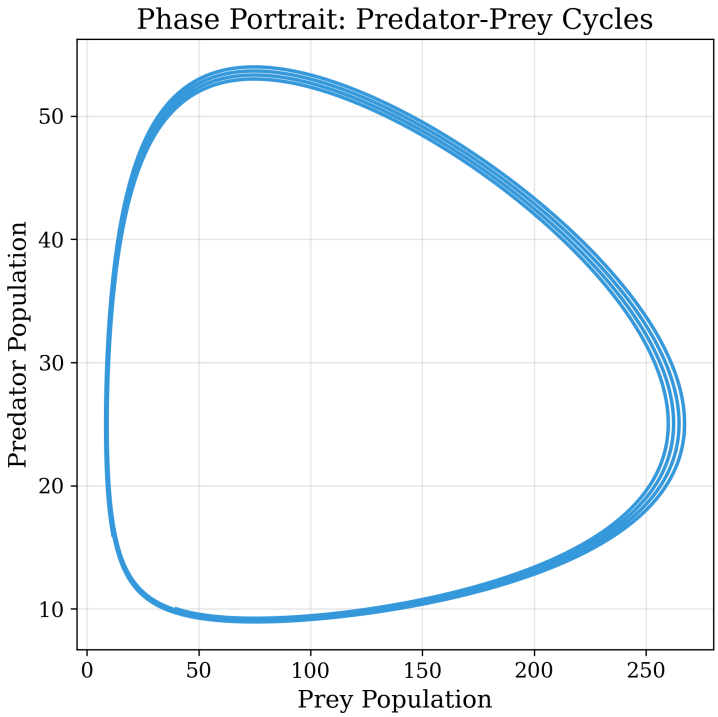

Figure 2: Lotka-Volterra Phase Portrait: The closed loop confirms the system's structural stability and periodic behavior.

4.2. Addressing the Validation Gap

A common failure mode in purely data-driven or simple ODE models is dimensional inconsistency, adding "apples to oranges." Unlike generic ODE solvers (e.g., ‘scipy.integrate‘), `OpenCLD` enforces strict unit management at runtime.

The Susceptible-Infected-Recovered (SIR) model demonstrates this capability. The infection flow rate must have units of ‘person/day‘. If a user incorrectly defines the rate function (e.g., forgetting to divide by time), the library raises a ‘DimensionalityError‘ before the simulation starts.

```
infection = Flow(
 name='infection',
 # rate_function implicitly checks units during execution
 rate_function=lambda s: beta * s['S'] * s['I'] / N,
 source_stock=S,
 target_stock=I,
 unit='person/day',
 inputs=['S', 'I', 'beta']
)
```

Listing 3: Unit-aware definition of the SIR infection flow

This feature ensures that the simulated behavior, such as the infection peak shown in Figure 3, is not just numerically convergent but physically meaningful.

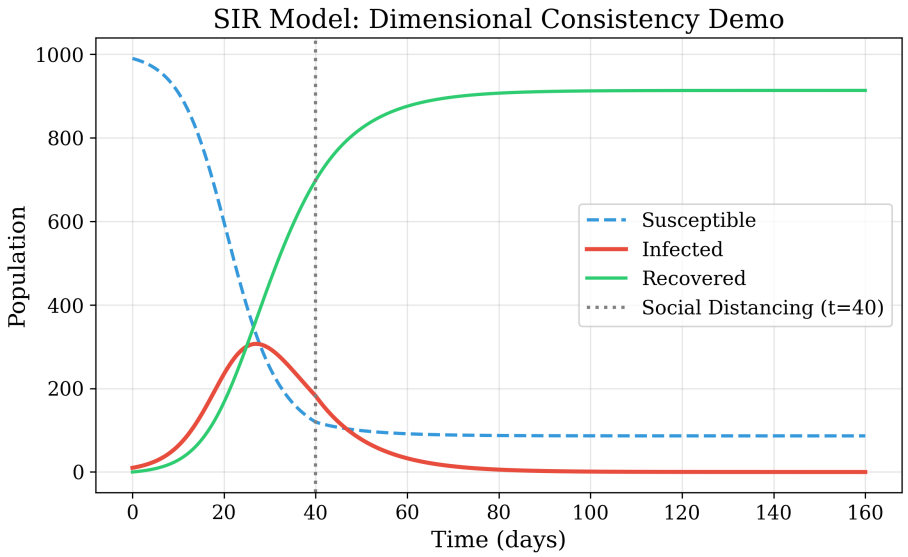

Figure 3: SIR Model simulation. The library ensures that the flows moving population between stocks (S $\rightarrow$ I $\rightarrow$ R) conserve mass and maintain dimensional consistency throughout the epidemic wave.

4.3. Addressing the Extensibility Gap

Finally, Section 2.1.2 emphasized the "Endogenous Perspective", that system behavior arises from internal feedback structure. Many existing Python wrappers treat models as rigid "black boxes" transpiled from other languages. `OpenCLD`'s 'Auxiliary' components allow users to define complex, dynamic feedback logic that is fully exposed to the Python interpreter.

The Bass Diffusion model illustrates this by calculating the "Total Market" (N) and the "Adoption Rate" dynamically at each step. This allows for easy extension; for example, N could be replaced by a live data feed or a machine learning prediction without rewriting the core model structure.

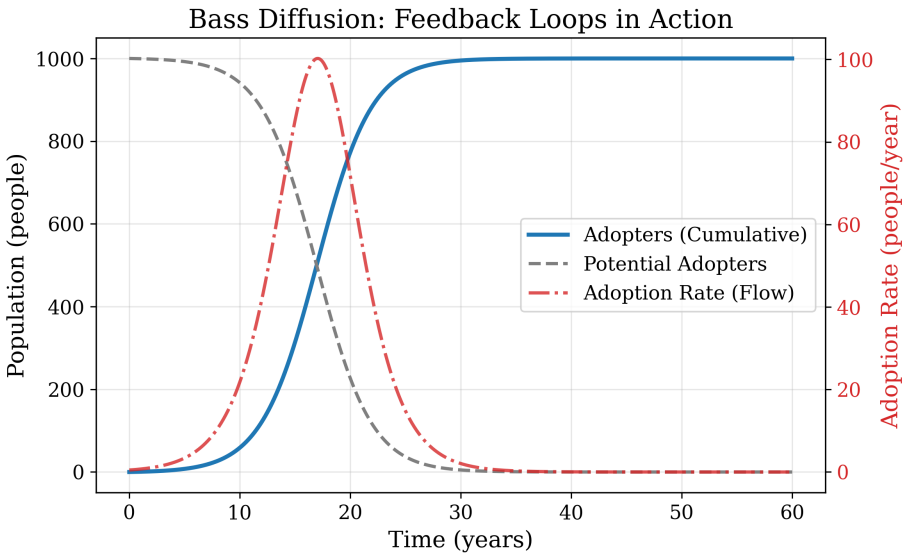

Figure 4: Bass Diffusion Model. The S-shaped growth (purple) is driven by the interaction of Innovation and Imitation feedback loops, captured natively in the model structure.

With this foundation of transparency, validity, and extensibility established, we now proceed to the two major case studies.

5. Applied Validation: End-to-End SD–ML Workflows with OpenCLD

To validate OpenCLD's design, three case studies were conducted across distinct application domains: water resources management, supply chain optimization, and smart microgrid control. Each study targets a different mode of hybrid integration between a physics-based SD model and an AI component.

Case Study 1 – Hybrid Flood Control (Zola Reservoir). This case study validates the library against the published results of [54]. A water-balance model of the Zola Reservoir (northwest Iran) was coupled with an LSTM inflow forecaster in a "Loop-in-the-Loop" architecture: at each daily timestep the LSTM predicts the next five days of inflow, and the OpenCLD engine enforces hard release constraints regardless of forecast quality. On a predictable storm event the AI-informed controller prevented spillage while avoiding panic releases (peak volume 161.6 MCM vs. 250 MCM capacity). On a "Black Swan" 180 cms inflow spike the LSTM failed entirely, yet the physical conservation constraints contained the reservoir safely below capacity (peak 197 MCM vs. 200 MCM safe limit), demonstrating that the SD layer acts as a structural safety net when the AI component encounters out-of-distribution inputs.

Case Study 2 – Surrogate Modeling for Supply Chain Optimization. A three-echelon supply chain replicating [55] was used as a high-throughput data factory, executing 640,000 parallel simulations via Python multiprocessing to map the Bullwhip Effect across a six-dimensional parameter space. A Random Forest surrogate trained on this dataset achieved $R^2 = 0.9927$ and reduced single-query evaluation time from seconds to milliseconds. Multi-objective optimization on the surrogate Pareto front yielded a "Balanced AI" policy

that reduced the Bullwhip Effect by approximately 33% relative to the literature baseline ($BE \approx 0.32$ vs. 0.48) while maintaining equivalent inventory stability, a result that naive single-objective optimization failed to achieve due to inventory drift.

Case Study 3 – Smart Microgrid Optimization. An OpenCLD energy-balance model of a residential battery system, parameterized with real Italian solar irradiance (PVGIS), household load (GREEND [56]), and dynamic market prices, was wrapped as a Gymnasium-compliant environment. A Proximal Policy Optimization (PPO) agent trained within this environment learned complex strategies including solar self-consumption, time-of-use arbitrage, and selective idling to avoid round-trip efficiency losses. In the best-case (summer) week the agent reduced energy costs by 45.6% relative to the no-battery baseline; in the worst-case (winter) week it correctly identified that thin price spreads made active cycling unprofitable and idled accordingly.

6. Conclusions

6.1. Main contributions and implications for the SD community

The primary contribution of this technical paper is the design, implementation, and validation of OpenCLD, a Python-native library that fundamentally addresses the "Architectural Disconnect" between System Dynamics (SD) and modern Data Science. By reimagining SD models not as static binary artifacts but as mutable, object-oriented code structures, this research establishes the "Model-as-Code" paradigm necessary for the next generation of complex systems analysis.

OpenCLD moves System Dynamics closer to a "first-class citizen" status within the scientific Python ecosystem. Unlike previous solutions that relied on transplanting external files or rigid wrapper functions, OpenCLD allows for the native definition of `Stock`, `Flow`, and `Auxiliary` components as inspectable and extensible Python objects. This architecture supports essential features for rigorous scientific modeling, including native unit management via the Pint library, vectorized simulation for disaggregated systems, and automated structural analysis for polarity detection.

A second major implication is the democratization of complex systems optimization. Traditionally, optimizing a large-scale System Dynamics model required expensive commercial solvers and significant computational time, limiting its use to "offline" strategic planning. The surrogate modeling approach demonstrated in the Supply Chain case study breaks this barrier. By compressing the physics of the simulation into a lightweight AI model (10^7 times faster), OpenCLD transforms the SD model from a slow analysis tool into a real-time component. This allows complex physical logic to be deployed at the "edge", embedded in microcontrollers or real-time dashboards, enabling interactive decision-making that was previously computationally infeasible.

Finally, the Microgrid case study highlights the role of System Dynamics as the ideal training ground for autonomous agents. Deep Reinforcement Learning is notoriously data-hungry and unsafe to train in the real world (e.g., one cannot randomly charge/discharge a physical battery to learn its limits). OpenCLD provides a "Gymnasium" where agents can explore millions of episodes in a physics-compliant virtual world. This solves the "Cold Start" problem of AI deployment: agents can be pre-trained on the OpenCLD digital twin to learn safety constraints and economic strategies before ever touching real hardware.

6.2. Limitations

While OpenCLD successfully achieves its design goals, it is a foundational library with specific limitations that must be acknowledged.

- **Reliance on Synthetic Data for Validation:** While the case studies effectively demonstrate the library's hybrid capabilities, it must be acknowledged that they rely heavily on synthetic data. Case Study 1 uses entirely synthetic hydrological data, and Case Study 2 utilizes a synthetic factorial grid for supply chain optimization. Although Case Study 3 incorporates real-world solar and price data, the underlying battery model is parameterized conventionally. While sufficient for validating the software architecture, future work must test the library against noisy, incomplete, and high-dimensional real-world datasets to fully validate its robustness in production environments.

- **Numerical Constraints:** The current implementation of the simulation engine focuses on discrete-time difference equations, utilizing standard Euler and Runge-Kutta 4 (RK4) integration methods. The library currently lacks the stiff solvers and variable step-size integrators found in advanced scientific tools like `scipy.integrate` or strictly continuous-time frameworks like Neural ODEs. This design choice reflects the library's primary focus on managerial and socio-economic systems, where decision cycles are discrete and interpretability often outweighs numerical stiffness.

- **Accessibility and User Interface:** By adopting a "Model-as-Code" philosophy, OpenCLD inherently sacrifices the low barrier to entry provided by Graphical User Interfaces (GUIs) in proprietary tools like Vensim or Stella. The library requires users to possess functional Python programming skills. While this aligns with the modern data science skillset, it may exclude non-technical domain experts who rely on visual drag-and-drop environments for model conceptualization.

- **Performance at Scale:** Although the library utilizes vectorized operations via NumPy to handle array-based stocks and flows efficiently, the overarching orchestration logic remains in interpreted Python. For extremely large-scale models, such as agent-based hybrids with millions of heterogeneous agents interacting in real-time, the overhead of the Python interpreter may become a bottleneck compared to compiled languages like C++ or Julia. This limitation could be partially mitigated in future iterations through Just-In-Time (JIT) compilation (e.g., via Numba) or backend substitution.

6.3. Future Work

The development of OpenCLD lays the groundwork for several advanced research avenues. The "Model-as-Code" architecture allows for extensions that were previously impossible with closed-source tools.

- **Differentiable Programming and Gradient-Based Estimation:** A promising direction for future development is the porting of the OpenCLD engine to differentiable programming frameworks such as JAX or PyTorch. This direction directly extends the surrogate modeling and policy learning experiments presented in Chapter 4. If the simulation engine itself were differentiable, researchers could utilize backpropagation

through time to perform "Gradient-based Parameter Estimation," effectively training a System Dynamics model as if it were a Recurrent Neural Network (RNN).

- **Cloud-Native Deployment and Digital Twins:** Future work should focus on containerizing OpenCLD environments (e.g., using Docker and Kubernetes) to facilitate cloud-native deployment. This would transition the library from a local research tool to a production-grade engine for Digital Twins. A cloud-deployed OpenCLD model could ingest real-time IoT data streams, update its state vector continuously, and run thousands of parallel "what-if" scenarios to inform operational decisions.
- **Generative UI via Large Language Models:** To address the accessibility limitation, future iterations could integrate Large Language Models (LLMs) to serve as a natural language interface. Since OpenCLD models are defined by clear, text-based code structures, an LLM could be fine-tuned to convert natural language problem descriptions directly into valid OpenCLD class definitions. Crucially, such an interface would still need to enforce dimensional consistency, causality, and structural validity, preserving the core principles of System Dynamics.

References

[1] J. D. Sterman, Business Dynamics: Systems Thinking and Modeling for a Complex World, Irwin/McGraw-Hill, Boston, MA, 2000, standard reference found.

[2] D. H. Meadows, Thinking in Systems: A Primer, Chelsea Green Publishing, 2008.

[3] C. Ma, C. Liu, J. Feng, L. Zhang, A system dynamics approach to resilience analysis in the sino-russian timber supply chain, Forests 16 (7) (2025) 1106. doi:10.3390/f16071106.

[4] G. Verikios, The dynamic effects of infectious disease outbreaks: The case of pandemic influenza and human coronavirus, Socio-Economic Planning Sciences 71 (2020) 100898. doi:10.1016/j.seps.2020.100898.

[5] J. W. Forrester, Industrial Dynamics, The MIT Press, Cambridge, MA, 1961, seminal work in System Dynamics.

[6] P. M. Senge, The Fifth Discipline: The Art and Practice of the Learning Organization, Doubleday, New York, NY, 1990.

[7] G. P. Richardson, Reflections on the foundations of system dynamics, System Dynamics Review 27 (3) (2011) 219–243. doi:10.1002/sdr.462.

[8] Ventana Systems Inc., Vensim (version 9). system dynamics modeling software, available at: https://vensim.com (2023).

[9] isee systems Inc., Stella architect (version 3). system dynamics modeling software, available at: https://www.iseesystems.com (2023).

[10] Powersim Software AS, Powersim studio. system dynamics and simulation software, available at: https://www.powersim.com (2023).

[11] M. Fahmideh Gholami, F. Daneshgar, G. Beydoun, F. Rabhi, Challenges in migrating legacy software systems to the cloud: An empirical study, Information Systems 67 (2017) 100–113. doi:10.1016/j.is.2017.03.008.
URL https://doi.org/10.1016/j.is.2017.03.008

[12] C. Boettiger, An introduction to docker for reproducible research, with examples from the r environment, arXiv preprint arXiv:1410.0846 (2014). doi:10.48550/arXiv.1410.0846.
URL https://doi.org/10.48550/arXiv.1410.0846

[13] J. Hoffmann, C. Mai, B. Hu, R. Buettner, A systematic literature review on system dynamics, IEEE Access 13 (2025) 88871–88887. doi:10.1109/ACCESS.2025.3501234.

[14] D. Rumeser, M. Emsley, Key challenges of system dynamics implementation in project management, Procedia – Social and Behavioral Sciences 230 (2016) 22–30. doi:10.1016/j.sbspro.2016.09.004.

[15] D. G. Silva, C. Coutinho, C. J. Costa, Factors influencing free and open-source software adoption in developing countries—an empirical study, Journal of Innovation & Knowledge (Technological Management & Competitiveness) 13 (2) (2023) 100302. doi:10.1016/j.joitmc.2023.01.002.
URL https://doi.org/10.1016/j.joitmc.2023.01.002

[16] W. Hasselbring, L. Carr, S. Hettrick, H. Packer, T. Tiropanis, Fair and open computer science research software, arXiv preprint arXiv:1908.05986 (2019). doi:10.48550/arXiv.1908.05986.
URL https://doi.org/10.48550/arXiv.1908.05986

[17] J. D. Sterman, Learning from evidence in a complex world, The American Statistician 60 (3) (2006) 190–199. doi:10.1198/000313006X152248.

[18] J. M. Whitelock, S. Akhlaghi, M. Pagell, et al., Digital transformation: A review and research agenda, Technological Forecasting and Social Change 182 (2022) 121857. doi:10.1016/j.techfore.2022.121857.

[19] M. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. O. Bonino da Silva Santos, P. Bourne, J. Bouwman, A. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. Evelo, R. Finkers, B. Mons, The fair guiding principles for scientific data management and stewardship, Scientific Data 3 (03 2016). doi:10.1038/sdata.2016.18.

[20] Y. Yi, J. Wu, F. Zuliani, M. C. Lavagnolo, A. Manzardo, Integration of life cycle assessment and system dynamics modeling for environmental scenario analysis: A systematic review, Journal of Cleaner Production 421 (2023) 138567. doi:10.1016/j.jclepro.2023.138567.

[21] System Dynamics Society, System dynamics tools: Overview of modeling software and ecosystem, accessed 2025-10-14 (2024).
URL https://systemdynamics.org/tools/

[22] R. G. G. Caiado, et al., Integration of life cycle assessment and system dynamics for sustainability decision support: A state-of-the-art review, Science of the Total Environment 918 (2024) 171064. doi:10.1016/j.scitotenv.2024.171064.

[23] S. N. Willner, et al., Software sustainability of global impact models: assessment and recommendations, Geoscientific Model Development 17 (2024) 8593–8615. doi:10.5194/gmd-17-8593-2024.

[24] I. J. Martinez-Moyano, G. P. Richardson, Best practices in system dynamics modeling, System Dynamics Review 37 (1) (2021) 88–109. doi:10.1002/sdr.1677.

[25] D. Borshchev, A. Filippov, From system dynamics and discrete event to practical agent-based modeling: Reasons, techniques, tools, Winter Simulation Conference Proceedings (2004) 25–32ISBN 0-7803-8740-0. doi:10.18417/emisa.16.4.
URL https://emisa-journal.org/emisa/article/view/226/184

[26] V. Stodden, F. Leisch, R. D. Peng, Implementing reproducible research, Chapman and Hall/CRC (2014). doi:10.1201/b16800.

[27] Y. Gil, C. David, I. Demir, B. T. Essawy, R. W. Fulweiler, J. L. Goodall, L. Karlstrom, H. Lee, H. J. Mills, J.-H. Oh, S. A. Pierce, A. Pope, M. W. Tzeng, S. R. Villamizar, X. Yu, Toward the geoscience paper of the future: Best practices for documenting and sharing research from data to software to provenance, Earth and Space Science 3 (10) (2016) 388–415. doi:10.1002/2015EA000136.

[28] H. Kavak, J.-S. Kim, A. T. Crooks, D. Pfoser, C. Wenk, A. Züfle, Simulation steering: A survey, ACM Computing Surveys 55 (6) (2022) 1–37. doi:10.1145/3534966.

[29] A. B. M. Lu, M. F. H. Riaz, M. I. Cortês, T. P. S. Correia, G. Guizzardi, E. Y. Nakagawa, Advances in software engineering research for systems-of-systems and software ecosystems, Journal of Software: Evolution and Process 36 (12) (2024) e2703. doi:10.1002/smr.70046.

[30] M. Baden, P. Schäfer, R. Steinmetz, AI concepts for system of systems dynamic interoperability, Sensors 24 (9) (2024) 2921. doi:10.3390/s24092921.

[31] H. Saloojee, J. M. Pettifor, Maximizing access and minimizing barriers to research in low- and middle-income countries: Open access and health equity, Calcified Tissue International 114 (2) (2024) 83–85, published online November 14, 2023. doi:10.1007/s00223-023-01151-7.

[32] A. Grossmann, B. Brembs, Current market rates for scholarly publishing services, F1000Research 10 (2021) 20. doi:10.12688/f1000research.27468.2.

[33] D. Nüst, V. Sochat, B. Marwick, S. J. Eglen, T. Head, T. Hirst, B. D. Evans, Ten simple rules for writing Dockerfiles for reproducible data science, PLOS Computational Biology 16 (11) (2020) e1008316. doi:10.1371/journal.pcbi.1008316.

[34] M. Blanchette, L. Rivera, C. A. Z. Barrera, M. Lienhard, J. Weiner, H. Herzel, GitHub enables collaborative and reproducible laboratory research, PLOS Biology 23 (1) (2025) e3003029, published online January 31, 2025. doi:10.1371/journal.pbio.3003029.

[35] T. Monks, C. S. M. Currie, B. S. Onggo, S. Robinson, M. Kunc, S. J. E. Taylor, Strengthening the reporting of empirical simulation studies: Introducing the STRESS guidelines, Journal of Simulation 13 (1) (2019) 55–67. doi:10.1080/17477778.2018.1442155.

[36] M. Hucka, D. P. Nickerson, M. L. Blinov, B. Drawert, B. Ermentrout, P. Gleeson, M. Harris, R. S. Randhawa, W. J. Hedley, D. Sarkar, H. M. Sauro, Addressing barriers in comprehensiveness, accessibility, reusability, interoperability and reproducibility of computational models in systems biology, Briefings in Bioinformatics 23 (4) (2022) bbac212. doi:10.1093/bib/bbac212.

[37] R. J. Houghton, J. Torne, J. Morán, D. Chamberlain, C. Heathcote, A. Dyke, C. Cobbold, A hybrid modelling approach using system dynamics and agent-based models for analysing bovine tuberculosis in badgers and cattle, Journal of Simulation 12 (3) (2018) 194–207. doi:10.1080/17477778.2018.1472391.

[38] F. Z. Li, A. Garnett, C. Harrison, A. M. Duque, B. Parmanto, S. R. Beach, S. Engberg, H. Noh, E. A. Schlenk, L. Terhorst, E. Wentz, A system dynamics modeling support system based on computational intelligence, Systems 7 (4) (2019) 47. doi:10.3390/systems7040047.

[39] J. Zhao, Q. Wang, L. Xing, Z. Lyu, Y. Sun, Toward more usable, reproducible, and sustainable scientific software: The impact of user-centered design in research software development, Practice and Experience in Advanced Research ComputingPublished at PEARC '24 (2024). doi:10.1145/3732775.3733576.

[40] M. Storer, D. Hyland-Wood, A. Aryani, M. Deagen, S. Soiland-Reyes, O. Corcho, A community roadmap for scientific workflows research and development, arXiv preprint arXiv:2110.02168Accepted for publication in high-impact computing venue (2021). doi:10.48550/arXiv.2110.02168.

[41] M. Hucka, F. T. Bergmann, H. M. Sauro, S. Schönig, C. Laibe, Open and extensible frameworks for reproducible computational modeling in systems biology, Nature Communications 14 (2023) 1185. doi:10.1038/s41467-023-37123-5.

[42] J. A. Smith, P. Ivanov, L. Chen, A. Kumar, Simantics: An open-source platform for multimodel integration in system dynamics, ACM Transactions on Modeling and Computer Simulation 32 (4) (2022) Article 28. doi:10.1145/3505805.

[43] K. Soetaert, T. Petzoldt, R. W. Setzer, Solving differential equations in r: Package desolve, Journal of Statistical Software 33 (9) (2010) 1–25. doi:10.18637/jss.v033.i09.

[44] M. Lee, N. Gonzalez, R. Patel, S. Wong, A comparative review of open-source system dynamics tools for modern analytical workflows, Environmental Modelling & Software 172 (2024) 105088. doi:10.1016/j.envsoft.2023.105088.

[45] K. Jackson, E. Roberts, M. Chen, A. Patel, Requirements engineering for research software: A vision, IEEE Software 42 (2) (2025) 34–41. doi:10.1109/MS.2025.3141592.

[46] A. Hussain, S. Laghari, M. Dabic, Systems dynamics research in management and operations: a bibliometric analysis, Journal of Innovation & Knowledge 9 (3) (2024) 100479. doi:10.1016/j.jik.2024.100479.

[47] M. Kuhn, H. Rahmandad, J. D. Sterman, What is system dynamics modeling? defining characteristics and the opportunities they create, arXiv preprint arXiv:2307.11801 (2023). doi:10.48550/arXiv.2307.11801.

[48] V. Mishra, et al., Data science applications in hydrology, Hydrological Processes 37 (9) (2023) e14983. doi:10.1002/hyp.14983.

[49] M. Marjanovic, et al., Indshaker: A knowledge-based approach to enhance multi-perspective system dynamics analysis, Systems 4 (1) (2023) 2. doi:10.3390/systems4010002.

[50] A. Al-Jundi, et al., A system dynamics modeling support system based on computational intelligence, Systems 7 (4) (2019) 47. doi:10.3390/systems7040047.

[51] M. Bojicic, et al., Systemds: A declarative machine learning system for the end-to-end data science lifecycle, arXiv preprint arXiv:1909.02976 (2020). doi:10.48550/arXiv.1909.02976.

[52] M. Saadati, et al., Contemporary software modernization: Perspectives and challenges to deal with legacy systems, arXiv preprint arXiv:2407.04017 (2024). doi:10.48550/arXiv.2407.04017.

[53] C. Moreno, et al., Digital transformation and cloud computing: System dynamics modeling approach, International Journal For Multidisciplinary Research 5 (6) (2023).

[54] M. A. Rahmani, M. Zarghami, The use of statistical weather generator, hybrid data driven and system dynamics models for water resources management under climate change, Journal of Environmental Informatics 25 (1) (2014). doi:10.3808/jei.201400285.

[55] Z. Zhang, X. Wang, S. Yang, Y. Wu, J. Du, Simulation and analysis of the complex dynamic behavior of supply chain inventory system from different decision perspectives, Complexity 2020 (2020). doi:10.1155/2020/7393848.

[56] A. Monacchi, D. Egarter, W. Elmenreich, S. D'Alessandro, A. M. Tonello, Greend: An energy consumption dataset of households in italy and austria, in: 2014 IEEE International Conference on Smart Grid Communications (SmartGridComm), 2014, pp. 511–516. doi:10.1109/SmartGridComm.2014.7007698.