Politecnico di Torino

OpenCLD: A Python-Native Library for Hybrid System Dynamics and Machine Learning Integration

MSc in Engineering and Management

Supervisors

Prof. Filippo Ottaviani

Prof. Giovanni Zenezini

Candidate

Pietro Viero

A.A. 2025/2026

Contents

List of Figures

List of Tables

Abstract

Modern complex systems, from supply chains to water infrastructure, resist adequate analysis by either purely theory-driven or purely data-driven methods alone. System Dynamics (SD) provides causal, physics-grounded simulation but remains trapped in closed, proprietary ecosystems incompatible with modern machine learning (ML) workflows. Conversely, ML delivers powerful pattern recognition but lacks the structural transparency and physical constraints necessary for safety-critical applications. Bridging this divide requires not merely methodological innovation, but a new computational foundation. This thesis introduces OpenCLD, an open-source, Python-native library that resolves this structural disconnect by implementing a strict "Model-as-Code" paradigm. Built on an object-oriented architecture, the framework treats SD components, Stocks, Flows, Auxiliaries, and Parameters, as mutable, inspectable Python objects. This design enables seamless, bidirectional in-memory data flow between the simulation engine and any Python-compatible tool, including `pandas`, `NumPy`, `scikit-learn`, and `PyTorch`, at any point during execution and without file I/O overhead or custom adapters. The simulation engine is designed for computational efficiency and correctness, featuring automated topological sorting to resolve algebraic dependencies, runtime dimensional consistency enforcement, and support for both Euler and Runge-Kutta 4 integration. Automated structural analysis, including causal polarity detection via numerical perturbation, provides immediate feedback on model feedback structure.

The library's capacity for deep hybrid integration is empirically validated through three distinct architectures. First, a "loop-in-the-loop" controller embeds an LSTM forecaster directly within the simulation time-step for proactive reservoir flood management; critically, when the AI component fails on an out-of-distribution "Black Swan" event, the physical constraints of the SD model provide a robust safety fallback. Second, a high-throughput "Data Factory" executes 640,000 parallel simulation runs via Python multiprocessing to train a Random Forest surrogate model, compressing supply chain optimization time by a factor of $\sim 10^2$ to 10^3 and

enabling a multi-objective policy that reduces the Bullwhip Effect by approximately 33% without sacrificing inventory stability. Third, the SD engine is deployed as a Gymnasium-compliant environment to train a Proximal Policy Optimization (PPO) agent for residential microgrid energy management, where the agent learns economically sophisticated behaviors including solar self-consumption and price arbitrage under real-world Italian market data. Together, these results demonstrate that treating SD models as executable, first-class Python objects, rather than static diagrams or transpiled artifacts, enables a "Hard Convergence" of physics-based simulation and data-driven AI that is transparent, scalable, and reproducible. OpenCLD is publicly available at `https://github.com/log-lab-polito/OpenCLD` and installable via PyPI.

Chapter 1

Introduction

1.1 Context and Motivation

The analysis of complex socio-technical systems, characterized by feedback loops, nonlinear interactions, and time delays, requires modeling tools that can capture endogenous behavior rather than attribute causality to external shocks. System Dynamics (SD), developed by Jay Forrester at MIT in the late 1950s, provides precisely this capability. By formalizing systems as networks of stocks, flows, and feedback loops, SD has enabled rigorous policy analysis across domains including supply chain management, public health, and environmental governance [1]. Its utility was demonstrated during the COVID-19 pandemic, where dynamic models integrating epidemiological, social, and economic subsystems informed government decisions about second- and third-order policy consequences [2]. Despite these recognized benefits, the adoption of SD in modern computational workflows remains constrained by significant technological and methodological limitations [3].

In parallel, Python has established itself as the dominant language for scientific computing and Artificial Intelligence. Its ecosystem, spanning numerical computation (NumPy), data analysis (pandas), and machine learning (scikit-learn, PyTorch, TensorFlow), constitutes the de facto infrastructure for reproducible, data-driven research [4, 5]. However, traditional SD tools, whether proprietary platforms or transpiler-based open-source libraries, were not designed to operate within this ecosystem. They remain architecturally isolated, creating a fundamental barrier to integrating SD with modern AI methods.

This architectural incompatibility is consequential because the modeling community is converging toward "gray-box" or "hybrid" approaches that combine the causal interpretability of

"white-box" SD models with the predictive power of "black-box" Machine Learning [6, 7]. Three classes of hybrid integration are particularly relevant to this thesis:

Embedded Forecasting: Deep learning architectures, such as Long Short-Term Memory (LSTM) networks, can be embedded directly within the SD simulation loop to provide real-time predictions of exogenous variables (e.g., river inflow), enabling proactive rather than reactive control policies [8].

Surrogate Modeling: When exploring high-dimensional parameter spaces, full SD simulations become computationally prohibitive. ML models trained on large batches of simulation output can serve as fast surrogate approximations, compressing days of computation into milliseconds of inference and enabling real-time optimization [9].

Policy Optimization via Reinforcement Learning: SD models can serve as rich, physics-compliant training environments for Reinforcement Learning (RL) agents. By formulating the SD model as a Gymnasium-compliant environment, RL algorithms such as Proximal Policy Optimization (PPO) can autonomously discover decision policies that outperform human-designed heuristics [10].

Each of these integration patterns requires the SD model to be a mutable, programmable object that can exchange state with ML components at every simulation step. Current SD tools, whether proprietary or open-source, do not provide this capability. This unmet architectural requirement is the central motivation for the development of `OpenCLD`.

1.2 Problem Statement

For over six decades, System Dynamics has provided a rigorous methodology for understanding complex behavior driven by feedback loops, accumulations, and time delays. Over the past two decades, Python has established itself as the standard language for scientific computing [11]. Advancing the analysis of complex systems requires combining the causal transparency of SD ("white-box" modeling) with the predictive capabilities of Machine Learning ("black-box" modeling) [6]. However, the current software landscape imposes a fundamental architectural barrier to this integration.

Proprietary platforms such as Vensim and Stella, while feature-rich, operate as closed ecosystems. Their proprietary file formats and lack of robust programmatic APIs isolate SD

from the broader scientific computing community and render large-scale experimentation impractical. Licensing costs further restrict accessibility for researchers and students [12].

Existing open-source Python libraries have attempted to bridge this gap, but with limited success. Libraries such as PySD function primarily as "transpilers," converting models defined in external proprietary formats into executable Python code. This architecture imposes a "translation tax": the model structure remains dependent on external GUIs, and the resulting Python object is structurally static at runtime, preventing the deep, dynamic integration required for advanced hybrid workflows. Other libraries, such as BPTK-Py, face a "generalist's dilemma," making architectural compromises to support multiple modeling paradigms simultaneously, which limits their fidelity for SD-specific applications [13, 14]. Consequently, researchers are forced to develop bespoke, non-reproducible integration scripts for each hybrid model, an approach that does not scale [10, 15].

The absence of a Python-native SD library that treats models as mutable, first-class code objects constitutes a critical gap. Without such a foundation, developing standardized methods for hybrid tasks, from ML-driven parameter estimation to reinforcement learning-based policy optimization, remains impractical. This thesis addresses this gap through the development of `OpenCLD`, a library built on the "Model-as-Code" paradigm to enable deep, bidirectional integration between System Dynamics and Machine Learning. In doing so, it also seeks to extract broader design principles for hybrid systems, moving beyond bespoke implementations to formalize how physical constraints, decision time horizons, and AI failure modes shape robust integration architectures.

It is useful at this point to introduce a conceptual distinction that underpins the thesis. The term "Soft Convergence" refers to the sequential or parallel use of SD and ML as independent tools, for example, using ML to forecast exogenous inputs to an SD model, or running both in parallel for comparative analysis, without any structural coupling between the two paradigms. While valuable, this mode of integration leaves the two approaches architecturally separate. In contrast, "Hard Convergence" denotes a tightly coupled, bidirectional integration in which an ML component operates as a live constituent of the SD simulation structure, replacing flow equations, receiving real-time state information, or guiding control decisions step-by-step. This requires the SD model to be a mutable, programmable object rather than a static simulation artifact. The development of `OpenCLD` is motivated precisely by the observation that existing tools can support "Soft Convergence" but impose architectural barriers that prevent "Hard Con-

vergence".

1.3 Research Objectives

The primary objective of this research is to design, implement, and validate `OpenCLD`, an open-source Python-native library that bridges the architectural gap between traditional System Dynamics tools and the modern data science ecosystem. The library establishes a "Model-as-Code" paradigm in which SD models are defined as mutable Python objects, enabling deep, bidirectional integration with Machine Learning frameworks for transparent, scalable, and reproducible hybrid simulations.

1.3.1 Specific Objectives

- To conduct a systematic review of hybrid SD-ML modeling approaches and a comparative analysis of existing software tools, identifying the specific architectural and integration gaps that hinder reproducible research.
- To design an object-oriented architecture for SD modeling in Python that treats models as mutable, first-class code objects, prioritizing modularity, extensibility, and adherence to scientific software design principles.
- To implement the core functionalities of the `OpenCLD` library, including a robust simulation engine with multiple integration methods, runtime dimensional consistency, and efficient data handling structures compatible with the Python data science ecosystem (e.g., NumPy, pandas).
- To validate native interoperability with established ML frameworks (e.g., scikit-learn, PyTorch, TensorFlow) by enabling advanced hybrid workflows such as embedded forecasting, surrogate modeling, and policy optimization via Reinforcement Learning.
- To empirically demonstrate the library's utility and flexibility through case studies spanning both classical SD models and hybrid SD-ML applications across distinct domains.
- To synthesize cross-case insights from these applications, establishing foundational design principles for hybrid modeling, specifically regarding the use of physical constraints

as a safety layer, the alignment of integration architecture with decision time horizons, and the predictability of AI failure modes within dynamic systems.

1.4 Scope and Limitations

1.4.1 Scope

The primary scope of this research is the software architecture and implementation of `OpenCLD`, a Python-native library designed to bridge the gap between System Dynamics and modern data science. The work is concentrated in three areas:

Library Development: The core contribution is the design and implementation of an object-oriented SD engine supporting the "Model-as-Code" paradigm. This includes fundamental SD classes (Stocks, Flows, Auxiliaries, Parameters) and a discrete-time simulation stepper with multiple integration methods [16].

Integration Capabilities: A central design goal is native interoperability with the Python scientific ecosystem. The library is validated for seamless data exchange with `NumPy` and `pandas`, and for the direct embedding of Machine Learning models (from `scikit-learn`, `PyTorch`, or `TensorFlow`) within the simulation loop [4, 5].

Methodological Validation: The library's utility is demonstrated through case studies ranging from classical SD models to hybrid SD-ML workflows. These examples are selected to validate the architecture's flexibility across diverse integration patterns, rather than to address domain-specific research questions.

1.4.2 Limitations

Numerical Constraints: The simulation engine focuses on discrete-time difference equations, implementing Euler and Runge-Kutta 4 integration methods standard in management and social science applications. Continuous-time solvers for stiff differential equations or Partial Differential Equations (PDEs) are not included [1, 17].

Feature Parity: As a foundational academic contribution, `OpenCLD` is not intended to achieve feature parity with mature commercial platforms (e.g., Vensim, Stella) in terms of graph-

ical user interfaces, exhaustive function libraries, complex array subscripts, or built-in optimization suites [12].

Demonstrative Case Studies: The case studies presented in Chapter 4 are designed to validate the technical integration workflows (e.g., embedded forecasting, surrogate training, reinforcement learning) rather than to provide empirical policy recommendations for specific industries.

Standard ML Algorithms: This research focuses on the integration architecture rather than the development of novel Machine Learning algorithms. The ML models employed in the hybrid examples use standard, established architectures to demonstrate interoperability [18].

1.5 Structure of the Thesis

The remainder of this thesis is organized into four chapters, structured to guide the reader from the theoretical foundations of the field to the technical implementation and experimental validation of the proposed solution.

Chapter 2: Literature Review establishes the theoretical context for the research. It begins by defining the core principles of System Dynamics and tracing the historical evolution of its applications from industrial operations to modern socio-technical policy. The chapter then critically examines the emerging convergence of System Dynamics and Artificial Intelligence, proposing a taxonomy for "Hybrid Modeling" (Sequential, Parallel, and Embedded). Finally, it conducts a comparative analysis of the current software landscape, identifying the "Architectural Disconnect" in existing tools, specifically the lack of native Python integration and collaboration features, that justifies the development of a new library [15, 19].

Chapter 3: Design and Architecture of `OpenCLD` details the technical contribution of this work. It presents the object oriented architecture of the `OpenCLD` library, explaining how the "Model-as-Code" paradigm is implemented through core classes (Stock, Flow, Auxiliary). This chapter also describes the library's unique features, including its native unit management system, its vectorized simulation engine, and the automated structural analysis algorithms used for polarity detection and loop discovery.

Chapter 4: Case Studies and Validation provides empirical evidence of the library's utility. We provide empirical proof that the library works by walking through specific use cases. These examples are designed to directly address the gaps we found in the literature:

- **Hybrid Control:** An application of "Loop-in-the-Loop" control where a Deep Learning forecaster (LSTM) is embedded directly into the simulation of a water reservoir to manage flood risks proactively [20].
- **Surrogate Modeling:** High-throughput batch simulation (320,000+ runs) to train a machine learning proxy for optimizing supply chain stability, demonstrating the library's capacity as a "Data Factory" [21].
- **Policy Learning:** A Reinforcement Learning application where an AI agent (PPO) learns to optimize the economic dispatch of a residential microgrid, navigating the trade-offs between battery efficiency and dynamic market prices.

Chapter 5: Conclusions and Future Work synthesizes the research findings, discussing the implications of the "Hard Convergence" between System Dynamics and Data Science. It summarizes the limitations of the current implementation and outlines a roadmap for future development, including the potential for differentiable programming and cloud-native deployment [15].

Chapter 2

Literature Review

2.1 System Dynamics: Principles and Applications

2.1.1 Principles

In recent years, the relevance of System Dynamics (SD) has increased significantly due to the rising complexity of global systems. The need for such methodologies is evident across diverse domains, from stabilizing fragile supply chains to managing the climate crisis. In these contexts, traditional linear logic often proves inadequate. Decisions rarely lead to immediate, predictable results; instead, they become entangled in time delays and feedback loops that are difficult to foresee [1].

The core tools within SD include Causal Loop Diagrams (CLDs) and Stock-and-Flow Diagrams (SFDs).

Causal Loop Diagrams (CLDs)

CLDs are used to map the causal relationships and feedback structures within a system, providing a qualitative understanding of its underlying dynamics. They consist of variables connected by arrows representing causal influences, with each link assigned a polarity (positive or negative) to indicate how the dependent variable changes when the independent variable changes. They are used to quickly capture hypotheses about what causes a system's behavior and to elicit the mental models of individuals or teams. Furthermore, CLDs are an effective tool for communicating the important feedback loops believed to be responsible for a problem.

Loops are identified as either:

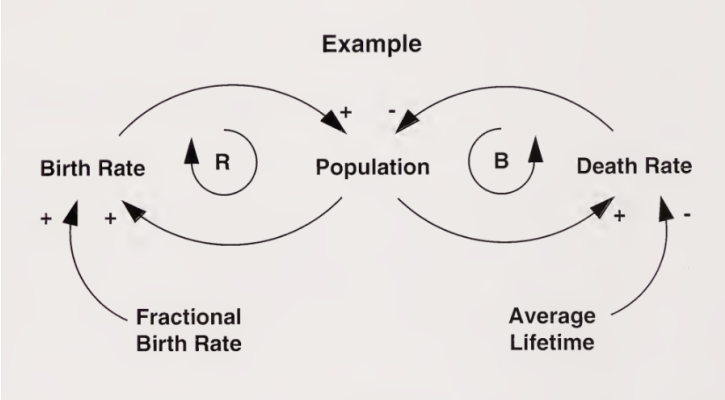

Figure 2.1: Causal Loop Diagram [1]

- **Reinforcing (R):** Loops which amplify change in the same direction.
- **Balancing (B):** Loops which seek to counteract disturbances and achieve equilibrium.

CLDs are valuable for mapping out connections, but they possess a major limitation: they do not differentiate between stocks and flows. Because they obscure this accumulation structure, users can easily misinterpret the actual physics of the system [1]. This is where Stock and Flow Diagrams (SFDs) are required to bridge the conceptual gap.

Stock and Flow Diagrams (SFDs)

SFDs quantify these relationships by representing accumulations (stocks) and flows that change these accumulations, enabling the creation of quantitative simulation models.

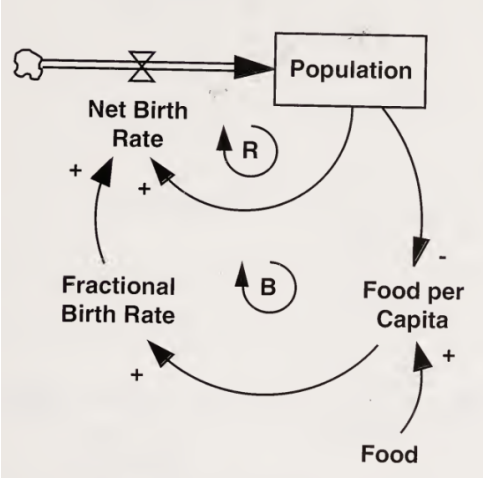

Figure 2.2: Stock and Flow Diagram [1]

- **Stocks:** Represented by rectangles, they characterize the state of the system and generate

information for decisions. They provide systems with inertia and memory, create delays, and are the source of disequilibrium dynamics.

- **Flows:** Represented by pipes with valves entering or leaving stocks. Sources and sinks for these flows are depicted as "clouds".

Mathematically, stocks are the integrals of their net flows (inflows minus outflows), meaning the change in a stock is determined by its rates:

$$Stock(t) = \int_{t_0}^{t} (Inflow(s) - Outflow(s))\, ds + Stock(t_0) \tag{2.1}$$

Understanding the relationship between stocks and flows is crucial for deriving system behavior intuitively, even without advanced mathematics [1].

SD is interdisciplinary, drawing on principles from mathematics, physics, and engineering, and applying them to human, physical, and technical systems, integrating concepts from cognitive and social psychology and economics. It aims to help decision-makers understand and overcome policy resistance, unintended consequences, and the counterintuitive behavior often observed in complex environments [1].

SD is also based on several other core elements:

Delays: A crucial component; in the real world, action and reaction are rarely instant. There is almost always a time lag between making a decision and seeing the result. This delay is often the culprit behind instability, causing systems to overshoot their goals or swing back and forth unpredictably.

Auxiliary variables: Represent all the elements that influence the system but are not stocks or flows; their values can change throughout a simulation based on their mathematical relationship with other parts of the system.

Parameters: Variables that affect the system's behavior but remain constant throughout a single simulation run [1].

2.1.2 Endogenous Perspective

The methodological tools of System Dynamics are built upon a distinct philosophical foundation known as the endogenous point of view. This perspective means that the dynamic behavior

of a system, its patterns of growth, oscillation, or collapse over time, arises from its internal structure. In contrast to conventional analyses that often attribute system behavior to external shocks or exogenous variables, the SD approach seeks explanations from within the system itself. The structure in question is composed of the interacting network of feedback loops, the accumulations represented by stocks, and the time delays inherent in physical and decision-making processes [22].

External events remain relevant, acting as triggers rather than fundamental causes. They unleash tendencies that are already inherent within the system, rather than generating the behavior entirely. Sterman illustrates this with the analogy of a pendulum: the behavior of the pendulum (oscillation) is a product of its internal structure (the mass of the bob, the length of the string, and the force of gravity). An external push (an exogenous event) may set the pendulum in motion, but the nature of that motion is dictated entirely by its endogenous structure. Consequently, two systems with different internal structures will react differently to the same external shock. This "system-as-cause" thinking shifts the focus of inquiry from attributing blame to outside agents or unpredictable events toward understanding how a system's own policies and structure generate its problematic behaviors [23].

This philosophy profoundly influences the practice of modeling. The first and most critical step in formulating a model is to define its boundary. The principle of the endogenous viewpoint dictates that the model boundary should be "closed" with respect to the problem at hand. This does not imply that the system is isolated from its environment, but rather that the boundary must be drawn wide enough to include the feedback mechanisms responsible for generating the behavior of concern. The goal is to create a model that is "causally closed," meaning it can reproduce the dynamic problem of interest without relying on external time series inputs to drive its behavior. This epistemological stance forces the modeler to move beyond identifying simple cause and effect correlations and instead to develop a theory of how the system's structure generates its own dynamics. This realization fundamentally alters the approach to policy design. Instead of attempting to shield the system from every external shock (a reactive approach), the focus shifts to identifying "leverage points" internally. By redesigning these structural connections, practitioners can address the root cause and engineer more resilient long-term outcomes [23].

2.1.3 The Rationale of Formal Modeling

The primary justification for employing the formal, computer-based methods of System Dynamics lies in the well-documented limitations of human cognition in understanding complex systems. Decision-making in any context is guided by mental models: the internal, conceptual representations of external reality that individuals use to reason, make judgments, and anticipate the future. Jay Forrester, the founder of System Dynamics, introduced this concept to the field, describing mental models as the "mental images or verbal descriptions" that we substitute in our thinking for the real system. All decisions, from the mundane to the strategic, are based on these models [1].

The System Dynamics methodology can be understood as a process designed to elicit, formalize, test, and ultimately improve the mental models of those managing a system. This formalization is necessary because unaided human intuition is notoriously unreliable in the face of dynamic complexity. Mental models are inherently limited and often flawed; they tend to oversimplify reality, omit critical feedback loops, misjudge time delays, and fail to account for the nonlinear relationships that govern system behavior. Due to the limited capacity of working memory, it is impossible to mentally simulate the dynamic consequences of even moderately complex feedback structures. Forrester observed that most mental models are dynamically simple, likely no more complex than what could be represented by a "fourth-order differential equation". This cognitive gap leads to what Sterman terms "misperceptions of feedback," where decision-makers' actions, based on flawed mental models, often lead to policy resistance, where policies fail or even worsen the very problems they are intended to solve [1].

Formal SD modeling provides a way to overcome these cognitive limitations. The process externalizes mental models, translating them from implicit assumptions and vague causal theories into a precise, explicit representation in the form of causal loop diagrams and a fully specified simulation model. This act of formalization makes the assumptions within a mental model transparent and debatable among a team of stakeholders. The resulting computer model then serves as a "management flight simulator" or a "virtual world" where decision-makers can conduct experiments, explore the long term consequences of their choices, and discover counterintuitive behaviors safely and in compressed time. The process is therefore one of accelerated organizational learning [1].

2.1.4 The System Dynamics Modeling Process

The application of System Dynamics is not a rigid, linear procedure but an iterative process of inquiry, learning, and refinement. While various authors present the process with slightly different terminology and levels of granularity, a strong consensus exists on the core intellectual workflow. This process guides the modeler from a vaguely defined problem to a robust, model-based understanding that can inform policy and strategy. The key stages of this iterative journey are outlined below [24].

1. **Problem Articulation (Problem Definition):** The process begins not with a system to be modeled, but with a specific, dynamic problem to be understood. The focus is on a problematic behavior that persists over time. This stage involves [1]:

    - **Identifying Key Variables:** Stakeholders brainstorm a list of variables relevant to the problem. Crucially, the primary criterion for inclusion is the variable's perceived importance to the problem, not the immediate availability of numerical data.
    - **Developing Reference Modes:** The problem is defined dynamically by sketching graphs of the key variables' behavior over time. These "reference modes" are not plots of raw data but rather "cartoons" that capture the essential character of the dynamic pattern, such as oscillation, S-shaped growth, or overshoot and collapse. They establish the time horizon for the study and serve as the empirical phenomenon the model must be able to explain [1].

2. **Dynamic Hypothesis (System Conceptualization):** With a clear, dynamic problem definition, the next step is to formulate a dynamic hypothesis, an initial theory explaining how the system's internal feedback structure could be generating the reference mode behaviors. The primary tool for this stage are the CLDs, these are powerful tools for capturing and communicating the assumptions in mental models and for facilitating group discussion about the system's structure [24].

3. **Formulation (Model Formulation):** In this phase, the qualitative hypotheses from the Causal Loop Diagram are translated into formal mathematical structures. The conceptual "story" of the system is formalized into a quantitative simulation model through the construction of Stock and Flow Diagrams (SFDs) [1].

- **Stocks** (also known as levels or accumulations) represent the state variables of the system. They accumulate the history of the system's behavior and create inertia and memory (e.g., population, inventory, staff morale).
- **Flows** (also known as rates) represent the activities that alter the stocks over time. Every flow must originate from and/or terminate at a stock [3].

Equations are then developed to define the flows based on the stocks and other variables in the model. A key discipline in this stage is ensuring that all parameters have a real world meaning and that all equations are dimensionally consistent, which serves as a powerful method for error detection. The best practice is to start with a simple model representing a core feedback process and incrementally add complexity [24].

4. **Testing (Model Testing and Evaluation):** Model testing is not a discrete final step but an ongoing process that occurs throughout formulation. Modelers are encouraged to "simulate as early as possible and often". This stage involves a comprehensive suite of tests (detailed in the following section) designed to build confidence in the model's structure, behavior, and policy implications. The model is rigorously challenged to expose flaws and identify areas for improvement [24].

5. **Policy Formulation and Evaluation (Model Use and Implementation):** Once sufficient confidence in the model has been established, it is used as a tool for policy design and analysis. The model becomes a laboratory for asking "what if?" questions, allowing for the simulation and evaluation of various policy options before they are implemented in the real world. This stage involves working closely with stakeholders to explore the short and long term consequences of different strategies, identify high leverage policies, and build consensus around a path forward [23].

2.1.5 Validation and Verification in System Dynamics

A common misconception in computational modeling is that a model can be proven "true" or "valid" in an absolute sense. In System Dynamics, this notion is rejected in favor of a more pragmatic approach focused on building confidence in a model's usefulness relative to its specific purpose. The process of building this confidence involves two distinct activities: verification and validation [25]. Verification is the process of ensuring the model has been built

correctly and matches its conceptual specification (i.e., "building the model right"). Validation is the process of determining the degree to which the model is an accurate representation of the real world for the purposes of the study (i.e., "building the right model") [26, 27].

Building confidence in a model is an iterative process that integrates multiple stages throughout the project lifecycle. A comprehensive "portfolio" of tests is employed to challenge the model from various perspectives. Because System Dynamics emphasizes endogenous causality, the validity of the structural formulation is the primary priority. This structural rigor becomes even more critical when developing hybrid SD-ML frameworks. The challenge lies in ensuring that while an ML component may achieve high statistical accuracy on historical data (verification), the overall system must also adhere to logical and physical consistency (validation). If a hybrid model predicts the future accurately but violates basic conservation laws or causal logic, its utility for robust decision-making is severely compromised.

Therefore, a model's ability to replicate historical behavior is considered meaningful only if its underlying causal structure is a plausible representation of the real system. A model that fits historical data perfectly but for the wrong reasons (i.e., with an unrealistic structure) offers no insight into how to design better policies for the future. This philosophy leads us to a specific hierarchy of tests, famously outlined by Sterman. We focus on structure first, and behavior second [25].

The tests of model structure are foundational: they assess whether the model adequately reflects the physical and logical reality of the system under study.

Structure Verification: This test involves comparing the model's diagrammed structure, the stocks, flows, and feedback loops, against descriptive knowledge of the real system. This is done through interviews with experts, direct observation, and review of relevant literature to ensure that the causal relationships depicted in the model are consistent with what is known about the system.

Parameter Verification: This test examines the model's parameters to ensure they correspond to tangible, real world concepts and that their numerical values are consistent with empirical data or are at least plausible. A parameter that cannot be conceptually or numerically justified represents a flaw in the model structure.

Extreme Condition Tests: One of the most powerful structural tests involves pushing model parameters to extreme (often physically impossible) values to see if the model behaves

in a plausible manner. For example, if a production delay is set to zero, output should respond instantly to orders. If the hiring rate is set to zero, the workforce should not increase. If the model fails under these extreme conditions, it indicates a flaw in the formulation of its equations, even if it behaves reasonably under normal conditions [25].

Reference Mode Replication: The model is simulated to determine if it can endogenously generate the qualitative patterns of behavior identified in the reference modes. The focus is on replicating the pattern of behavior (e.g., the period and phase of oscillations, the shape of S-shaped growth), not on achieving a point-for-point match with historical data. The use of standard statistical goodness-of-fit measures (e.g. R^2) is often viewed with caution, as they can reward models that are over-fitted to noise in the data and have poor structural integrity [28].

Once sufficient confidence in the model's structure is established, its behavior is tested:

Sensitivity Analysis: This involves systematically varying the values of uncertain parameters to determine if the model's conclusions, its policy recommendations, are sensitive to those assumptions. A model is considered robust if its fundamental behavioral patterns and policy implications do not change when reasonable alternative values for uncertain parameters are used. If the model's conclusions are highly sensitive to a particular parameter, it highlights the need for better empirical estimation of that parameter or the design of policies that are robust to its uncertainty [26, 27].

This approach to validation aligns SD modeling more closely with the scientific method of theory building than with purely statistical forecasting. The model is treated as a formal theory of the system's structure. Structural tests assess the validity of the theory's premises, while behavioral tests assess its explanatory power. A model is deemed useful for policy design only when it passes a battery of tests that establish confidence in both its structure and its ability to explain the dynamics of concern.

2.1.6 Evolution of System Dynamics Applications

The application of System Dynamics (SD) has undergone significant evolution over the past six decades, shifting from industrial operations to complex socio-economic policy and, most recently, to hybrid computational methods. This evolution can be categorized into four distinct eras (see Figure 2.3).

The Foundational Era (1960s–1970s)

The field originated with Jay Forrester's seminal work, *Industrial Dynamics* (1961) [29], which applied feedback control theory to corporate stability, production, and inventory problems. This era quickly expanded beyond the factory floor to social systems. In 1969, *Urban Dynamics* [30] challenged conventional wisdom on city planning, arguing that low-cost housing programs often exacerbated urban decay. The era culminated in *The Limits to Growth* (1972) [31], a global model commissioned by the Club of Rome that famously linked resource depletion with population and economic growth. These early works established the field's capability to model complex, counter-intuitive systems.

The Standardization and Software Era (1980s–1990s)

This period was characterized by the formalization of the field and the democratization of modeling tools. The founding of the System Dynamics Society in 1983 created a professional hub for practitioners. A major technological leap occurred with the release of Stella in 1985 [32], which introduced a graphical user interface that made modeling accessible to non-mathematicians. Simultaneously, the focus returned to corporate operations but with a behavioral twist. Sterman's work on Behavioral Supply Chain Management (1989) [33] and the "Beer Distribution Game" experimentally proved that supply chain instability (the Bullwhip Effect) stems from bounded rationality and misperception of feedback, rather than just external shocks.

The Policy & Society Era (2000s–2010s)

As simulation power increased, SD moved into detailed public policy and epidemiology. A key milestone was the development of PRISM (Prevention Impacts Simulation Model) in 2005 [34], used by the CDC to model chronic disease prevention and health policy. Climate policy also became a central application. Building on earlier global models, Sterman et al. (2012) and the Climate Interactive team developed rapid-simulation tools like C-ROADS and later EN-ROADS (2019) [35], which allowed policymakers to test climate strategies in real-time during UN negotiations.

The Modern Computational Era (2020s–Present)

The current era is defined by the integration of System Dynamics with modern data science and Artificial Intelligence. The release of PySD (2016) [13] bridged the gap between traditional SD software (Vensim) and the Python data science ecosystem. This integration paved the way for "Hybrid System Dynamics." A prime example is SDGym, which combines SD environments with Deep Reinforcement Learning (RL), allowing AI agents to learn optimal policies within complex dynamic systems [10]. This marks a shift from purely human-designed policies to AI-assisted policy discovery.

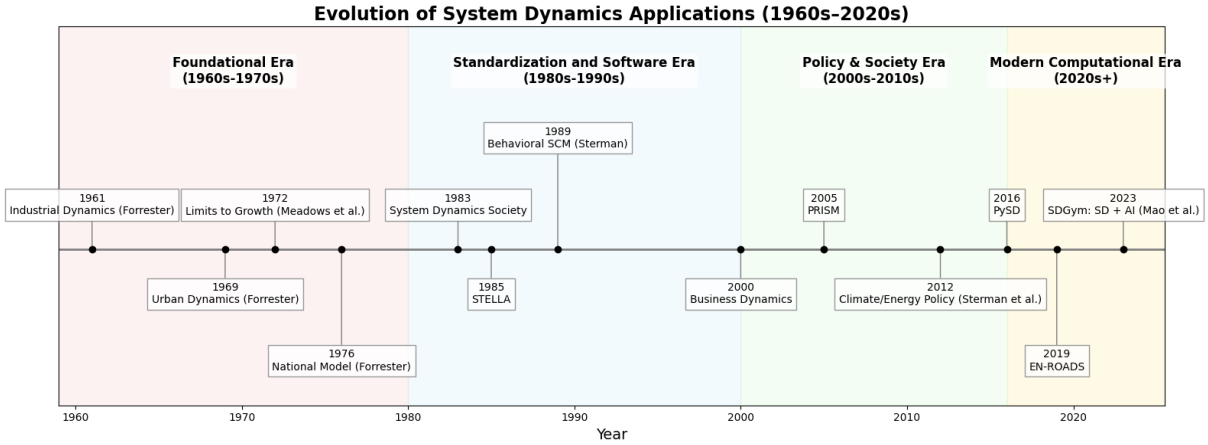

Figure 2.3: The Evolution of SD

The relevance of System Dynamics continues to grow in the post-pandemic era. Recent studies confirm this momentum, counting over 2,000 peer-reviewed SD applications published in 2023 alone [11]. Contemporary research has shifted toward "hybrid" applications, particularly in sustainability transitions, climate policy, and smart city governance, highlighting a demand for tools that can handle data-intensive and feedback-rich complexity [19].

2.2 Integration between SD and ML

2.2.1 The Rationale for a Hybrid Paradigm

The growing necessity of integrating SD and ML stems from the inherent limitations each approach exhibits when applied independently to complex, real-world dynamic systems. Neither traditional physics-informed models, which often form the analytical foundation of SD, nor purely data-driven deep learning models are sufficient on their own to fully capture and man-

age systems characterized by massive, noisy datasets [6, 7]. Consequently, the convergence of these two paradigms offers a synthesis that leverages their respective methodological strengths.

System Dynamics (SD) as the "White-Box"

Due to the explicit nature of its components (SFDs, CLDs), SD serves as a quintessential "White-Box" methodology. Every element is formally defined, from the causal relationships between variables to the differential equations governing the system state. This structural transparency facilitates stakeholder alignment, allowing teams to interrogate assumptions and establish consensus on the problem boundary. SD is particularly effective for policy testing and scenario analysis, enabling decision-makers to simulate the long-term consequences of strategic interventions [29, 36]. However, SD models are highly sensitive to parameter accuracy, and the calibration process can be computationally expensive and methodologically challenging, particularly when confronted with high-dimensional or noisy empirical data [37].

Machine Learning (ML) as the "Black-Box"

In parallel with the evolution of SD, the last two decades have seen the rise of Machine Learning as a transformative tool for data analysis. ML algorithms, particularly deep learning models, excel at identifying subtle patterns and complex, non-linear relationships within vast, high-dimensional datasets. Unlike SD, ML models are generally data-driven; they learn system behavior directly from historical or simulated data without explicit reliance on pre-defined causal structures or system equations. This makes them exceptionally powerful for prediction, forecasting, and classification tasks where the volume of data is large, but theoretical understanding of the system is incomplete [38].

While ML demonstrates superior predictive performance, this capability often comes at the cost of interpretability. Deep neural networks frequently operate as "black boxes," obscuring the internal logic driving their outputs. This opacity presents a critical barrier in domains where decisions carry significant operational or socio-economic consequences, as stakeholders require causal justification for algorithmic recommendations. Without structural transparency, establishing trust and verifying the physical plausibility of these models remains a fundamental challenge [7, 39].

The "Gray-Box" Synthesis

The recognition of these complementary strengths and weaknesses has led to the emergence of "gray-box" or hybrid modeling, a paradigm that seeks to combine the causal interpretability of "white-box" SD models with the predictive power and adaptability of "black-box" ML techniques. This convergence is not merely a technical exercise but represents a more pragmatic and powerful philosophy of modeling. It acknowledges that for most complex real world systems, our knowledge is partial. The "white-box" SD structure can be used to represent the well-understood causal mechanisms, physical laws, and feedback structures of the system. Simultaneously, the "black-box" ML component can be used to learn the parts of the system that are unknown, too complex to model from first principles, or for which rich data is available [40].

This approach, sometimes referred to in other fields as Physics-Guided Deep Learning or the integration of first principles knowledge with data-driven methods, allows each paradigm to mitigate the limitations of the other. SD provides a causal framework that can constrain the ML model, improving its generalizability and preventing it from learning spurious correlations that are not physically plausible. In return, ML provides a data-driven method to estimate unknown parameters, learn complex non-linear relationships, and adapt the model as new data becomes available, thereby increasing the SD model's accuracy and relevance [41].

This synthesis is not a simple trichotomy of "white," "black," and "gray" models, but rather a continuous spectrum of hybridization. The degree to which a model is driven by theory versus data can vary significantly depending on the specific integration method and the problem context. At one end of the spectrum, a model might be predominantly a "white-box" SD structure, with ML used sparingly to estimate complex parameters from data. In this case, the human-defined causal theory of the system remains paramount. At the other end, a model might be primarily a "black-box" deep learning architecture, such as a Neural Ordinary Differential Equation, which learns the entire system dynamics from data with only minimal structural priors provided by the modeler. In between lie a vast range of possibilities, such as replacing a specific, poorly understood subsystem within a larger SD model with a neural network. This spectrum-based view provides a more sophisticated framework for evaluating different hybrid approaches, where the optimal balance between theory and data depends on the degree of prior knowledge about the system versus the availability and quality of observational data [42, 43].

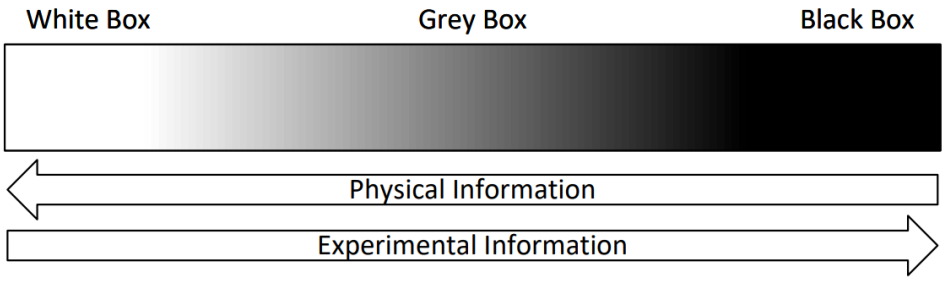

Figure 2.4: System knowledge abstraction scale [6]

2.2.2 A Taxonomy of SD-ML Integration Frameworks

As the field of hybrid modeling matures, a variety of conceptual frameworks and typologies have emerged to classify the diverse ways in which System Dynamics and Machine Learning can be combined. Moving beyond the "why" of integration to the "how," these taxonomies provide a structured way to understand the relationships between the component models, the flow of information, and the ultimate purpose of the hybrid construct. A comprehensive understanding of these frameworks is essential for selecting the appropriate integration strategy for a given research problem.

One of the most straightforward ways to classify hybrid models is by their structural architecture, how the SD and ML components are arranged and interact with one another. Drawing from the broader literature on multi-method simulation, three primary patterns can be identified:

Sequential Integration: In this configuration, the models operate in a pipeline, where the output of one model serves as the input for the other. This is a loosely coupled approach. A common example is using a validated SD model to generate large volumes of synthetic time-series data, which is then used to train an ML model for a specific task, such as anomaly or disruption detection. This leverages the SD model's ability to simulate a wide range of systemic behaviors under different conditions, providing a rich training set that may not be available from real world data alone. The reverse is also common: an ML model, such as a time-series forecasting algorithm, can be used to predict exogenous variables (e.g., future market demand, raw material prices) that are then fed as inputs into an SD model to simulate their impact on the system's dynamics [19, 44].

Parallel Integration: In this pattern, the SD and ML models run independently to address the same problem or different facets of it. Their results are then compared, combined, or used

for cross-verification. For instance, an SD model might be built to explain the long term strategic dynamics of a market, while an ML model is trained to make short term tactical forecasts. By comparing the outputs, analysts can gain confidence in their understanding of the system at multiple timescales or identify discrepancies that point to flaws in one of the models' assumptions. This approach leverages the different perspectives of each paradigm to build a more robust and holistic understanding [19].

Embedded (or Hierarchical) Integration: This represents the most tightly coupled and deeply integrated form of hybrid modeling. In this architecture, an ML model functions as an active, internal component within the SD model's structure, typically replacing a specific equation or an entire sub-model. For example, a complex and non-linear decision rule within an SD model, such as a company's pricing policy, might be difficult to formulate from first principles. If sufficient historical data on pricing decisions exists, a neural network can be trained to learn this behavior and embedded directly into the SD model to represent that flow equation. This allows the modeler to represent data-rich components with high fidelity while retaining the overall causal structure of the broader system [6].

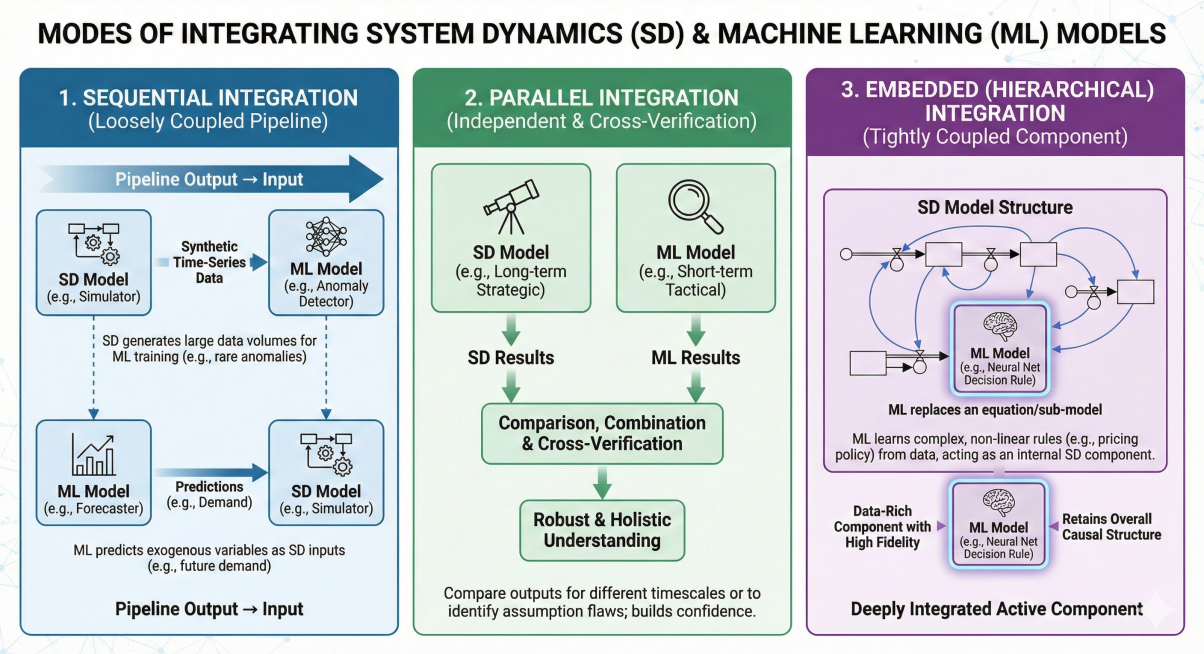

Figure 2.5: SD-ML Integrations Typologies

Purpose-Driven and Information-Flow Taxonomies

Another valuable way to classify hybrid models is by their primary purpose and the direction of influence between the theory-driven and data-driven components. This perspective focuses on which paradigm is being augmented by the other. The literature often distinguishes between two main categories:

ML-informed SD (or PBM-informed DL): In this mode, the primary goal is to enhance a process-based model (the SD model) using insights from data. Machine learning serves as a tool to improve the fidelity, accuracy, or scope of the theory-driven model. This category includes the most common applications of hybrid modeling, such as using ML for parameter estimation, where algorithms learn the values of model constants from observational data. It also includes more advanced techniques like using data to discover the mathematical form of an unknown relationship within the model or incorporating physical constraints derived from the SD model's equations into the loss function of a deep learning model to guide its training (a technique known as Physics-Informed Neural Networks) [19, 45].

SD-informed ML (or DL-informed PBM): Here, the roles are reversed. The primary goal is to improve a data-driven ML model by incorporating structural knowledge from an SD model. The SD model provides a theoretical or causal prior that enhances the ML algorithm's performance, robustness, or interpretability. Examples include using an SD model to generate synthetic data for training an ML model, as seen in the sequential pattern, or using the SD model as a rich, dynamic simulation environment for training a Reinforcement Learning agent to find optimal policies. In this case, the SD model's feedback structure provides a realistic "world" for the agent to learn in, leading to more robust and generalizable policies than could be learned from static historical data alone [44, 46].

Emerging Conceptual Frameworks

Beyond these general classifications, recent research has begun to propose more sophisticated, purpose-built frameworks that aim for a deeper and more principled synthesis of SD and ML. A leading example is the "Interpretable Neural System Dynamics" (INSD) framework. This framework addresses the core weakness of many ML models, their lack of causal reliability and

interpretability, by proposing a structured pipeline to build hybrid models that are "interpretable by design" rather than attempting to explain a black box post-hoc [39].

The INSD pipeline consists of three integrated steps:

1. **Concept Learning:** This initial step uses techniques from Concept Based Interpretability to transform raw, low level, and often heterogeneous operational data (e.g., sensor readings, transaction logs) into high level, semantically meaningful variables or "concepts" that are understandable to human experts (e.g., 'supply chain fragility', 'market saturation', 'workforce morale') [39].

2. **Causal Learning:** The second step employs methods from Causal Machine Learning and causal discovery to identify the underlying cause and effect relationships among the high level concepts extracted in the first step. This moves beyond identifying mere statistical correlations to uncovering the potential causal structure of the system [39].

3. **System Dynamics Modeling:** In the final step, the learned causal graph and relationships are formalized into a quantitative simulation model. This can be done by constructing a traditional SD model or by using advanced techniques like Neural Ordinary Differential Equations to learn the precise mathematical form of the dynamic relationships between the concepts [39].

The INSD framework represents a significant theoretical advance because it provides a roadmap for leveraging the power of ML not just for prediction, but for the automated discovery of causal structure from data, which aligns with the foundational goals of System Dynamics modeling [39].

The following table synthesizes these various classification schemes into a unified taxonomy, providing a clear and comparative overview of the primary approaches to SD-ML integration.

2.2.3 Methodological Pillars of Hybrid Integration

The practical implementation of hybrid SD-ML models is supported by a growing set of specific methodologies and algorithms. These techniques can be separated into three main pillars, each addressing a different aspect of the modeling process: using ML to calibrate and ground SD models in data, using deep learning to represent and discover dynamic relationships, and using reinforcement learning to optimize policies within SD simulations.

Table 2.1: Taxonomy of SD-ML Integration Patterns

Pattern	Sub-Type	Primary Purpose	Coupling & Data Flow	Architectural Req.	Example Methodologies
Sequential	Pipeline	Data Generation / Forecasting	Loose (File/IO) Unidirectional	Compatible with any tool (Vensim/Stella).	SD model generates synthetic data to train a classifier; ML forecasts demand as exogenous input [44].
Parallel	Comparative	Cross-Validation / Triangulation	None (Independent) Bidirectional Comparison	Independent execution.	SD explains long term strategy; ML gives short term tactical predictions. Results are triangulated [19].
Embedded	Structural	Representation / Surrogate Modeling	Tight (In-Memory) ML $\rightarrow$ SD (Equation replacement)	Native Function Interop (Requires Model-as-Code)	A Neural Network replaces a complex flow equation (e.g., decision rule) inside the SD loop. PINNs constrained by physical laws [6].
	Inference	Calibration / Parameter Estimation	Tight (Optimization Loop) Data $\rightarrow$ SD (Parameter search)	Fast Iteration / Differentiability	Amortized Bayesian Inference (ABI) estimates posterior parameter distributions from noisy data [43].
	Control	Policy Optimization (RL)	Tight (Step-wise Loop) SD $\leftrightarrow$ ML (State/Action exchange)	Low-Latency Stepping (No file I/O overhead)	SD model acts as a Gymnasium environment for a Reinforcement Learning agent to learn optimal policies [10].
Holistic	Synergistic	Causal Discovery	Unified Pipeline Bidirectional (Automated)	End-to-End Framework	Interpretable Neural System Dynamics (INSD) learns concepts and causal links from data, formalizing them into a dynamic model [7].

Machine Learning for Model Calibration and Parameter Estimation

One of the most fundamental and widespread applications of ML in the context of SD is for model calibration and parameter estimation. A significant challenge in building credible SD models is determining the numerical values for the model's parameters. Traditionally, this has often been a manual process of trial and error, relying on expert opinion or limited data points. ML offers a more rigorous, data-driven, and automated approach to this critical task [47].

The methodologies employed span a wide range of complexity. At the simpler end, classical regression techniques and gradient-based optimization algorithms can be used to find the parameter values that minimize the error between the model's output and historical time series

data. These methods from the field of system identification provide a formal basis for fitting the model to evidence [48].

More recently, research has moved towards advanced Bayesian methods that not only provide point estimates for parameters but also quantify their uncertainty. A particularly promising approach is Amortized Bayesian Inference (ABI), often framed as Likelihood-Free Inference (LFI) or Simulation-Based Inference (SBI). Instead of running computationally expensive sampling algorithms like Markov Chain Monte Carlo (MCMC) for each new dataset, ABI trains a neural network (e.g., using Neural Posterior Estimation) to learn the direct mapping from observational data to the full posterior probability distribution of the model parameters. Once trained, this network can instantly generate the posterior distribution for any new data, making the process highly efficient and scalable. The primary advantage of these Bayesian approaches is the shift from single point parameter estimates to a complete characterization of uncertainty. This allows for more robust sensitivity analysis and provides a principled way to assess confidence in the model's structure and its policy implications, directly addressing a key aspect of the SD validation process [43, 47].

Deep Learning for Representing System Dynamics

Beyond parameter estimation, a more profound level of integration involves using deep learning models to represent the dynamic equations of the system itself. This moves ML from a supporting role in calibration to a central role in model formulation. This approach is particularly useful when parts of a system are poorly understood from first principles but are rich in data [41].

Surrogate Modeling The most straightforward application in this category is surrogate modeling. If a particular component or sub-model within a larger SD simulation is computationally intensive to run (e.g., it involves solving complex partial differential equations or calls another detailed simulation), its input-output behavior can be approximated by a trained ML model, such as a neural network or a gradient boosting machine. This ML model then acts as a computationally cheap "surrogate" for the original component. The primary motivation for this technique is to improve computational efficiency, enabling extensive sensitivity analysis, uncertainty quantification, or optimization runs that would be infeasible with the original, slow model [6, 9].

Physics-Informed Neural Networks (PINNs) A more sophisticated and powerful "gray-box" technique is the use of Physics-Informed Neural Networks (PINNs). A PINN is a deep neural network that is trained to satisfy two objectives simultaneously: fitting observed data and obeying the known governing physical laws of the system, which are typically expressed as differential equations. This is achieved by incorporating the differential equations directly into the network's loss function. The total loss is a combination of the data mismatch error (how well the network's output matches measurements) and a "physics" error (how well the network's output satisfies the differential equations) [6].

This methodology is a natural fit for System Dynamics, as SD models are fundamentally systems of coupled ordinary differential equations (ODEs). By embedding the SD model's equations into the loss function, the PINN is strongly constrained by the known causal structure of the system. This provides a powerful inductive bias that allows the network to learn effectively even from sparse, noisy, or incomplete data, as the "physics" guides the model towards plausible solutions in regions where no data is available. PINNs thus represent a true synthesis, blending the flexibility of neural networks with the rigor of first principles modeling [41].

Neural Ordinary Differential Equations (Neural ODEs) The deepest level of integration is achieved with Neural Ordinary Differential Equations (Neural ODEs). This paradigm reimagines a deep neural network not as a sequence of discrete layers, but as a continuous-time dynamical system. The core idea is to model the evolution of the hidden state h of the network with an ordinary differential equation:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta) \tag{2.2}$$

where the function f, which defines the dynamics of the system, is itself parameterized by a neural network with weights θ. To compute the output of the network (the state at a future time t_1), a numerical ODE solver is used to integrate this equation forward from the initial state $h(t_0)$ (the input data) [49].

The connection to System Dynamics is profound. An SD model is a system of ODEs that describes the rates of change of its stocks. A Neural ODE can therefore be used to learn the entire dynamic structure of a system (or a complex part of it) directly from time-series data, without the modeler needing to manually specify the functional form of the equations. This

shifts the role of ML from simply enhancing a pre-specified model towards the automated discovery of the model itself [41, 45].

However, it is critical to evaluate the relevance of Neural ODEs within the specific context of management science and socio-economic systems. While Neural ODEs excel in modeling continuous physical phenomena (e.g., fluid dynamics, orbital mechanics) where the underlying laws are smooth and immutable, they struggle with the discrete, event-driven nature of human decision-making. Management systems are characterized by discrete decision intervals (e.g., weekly ordering cycles), non-linear logical constraints (e.g., non-negativity of inventory), and operational delays (e.g., shipping times) that do not map cleanly onto continuous-time derivative functions [1]. In these contexts, attempting to learn the entire system end-to-end with a Neural ODE often results in a model that violates basic operational logic. Therefore, for socio-economic applications, a "structural embedding" approach, where ML components operate within a discrete-time, explicitly defined causal structure, is often more appropriate and robust than pure continuous-time learning [6].

Symbolic Regression and Equation Discovery While Neural ODEs offer powerful predictive capabilities, they often remain “black boxes” lacking interpretability. To address this, Symbolic Regression algorithms (such as genetic programming or AI-DARWIN) have been applied to System Dynamics to perform “White-Box” discovery. Unlike neural networks that approximate functions via weights, these algorithms search a library of mathematical operators to identify the precise symbolic equation (e.g., $dx/dt = a \cdot x - b \cdot x \cdot y$) that best describes the data. This allows modelers to discover interpretable structural laws from empirical observations, which can then be directly embedded into the SD stock-and-flow structure [6, 49].

The Gap: Continuous Physics vs. Discrete Operations While Neural ODEs and PINNs represent a profound synthesis of deep learning and differential equations, they are predominantly applied in physics and engineering contexts where the governing laws are continuous, immutable, and smooth [6]. However, as noted above, in socio-economic domains such as Supply Chain Management or Public Policy, systems are often characterized by discrete decision intervals, non-linear logical constraints (e.g., non-negativity), and operational delays that do not map cleanly onto continuous-time derivative functions [1].

Consequently, a distinct need arises for “discrete hybrid” architectures. Unlike Neural ODEs, which often seek to learn the entire derivative function $f(x)$ from data end-to-end,

practical management models require "structural embedding." In this approach, a specific machine learning component (e.g., a pricing policy trained on data) functions as a discrete block within a broader, explicitly known causal structure (e.g., the inventory flow conservation) [6]. This distinction defines a critical operational niche: frameworks are needed that can bridge the gap between continuous system theory and the discrete, time-stepped reality of operational data. Purely physics-informed deep learning methods currently leave this gap unaddressed, highlighting the necessity for libraries like `OpenCLD` that prioritize discrete-time structural embedding [10].

Reinforcement Learning for Dynamic Policy Optimization

The third methodological pillar focuses on decision-making and control. While traditional SD modeling explores policy impacts through manual scenario analysis, integrating Reinforcement Learning (RL) provides a powerful framework for automatically discovering optimal decision-making policies in dynamic environments [50].

In this hybrid setup, the System Dynamics model functions as the simulated "environment" within which the Reinforcement Learning agent operates and learns. The conceptual mapping between the two paradigms is structurally coherent. First, the agent observes the model's stocks and key auxiliary variables to determine the current system "state." Subsequently, the agent executes "actions" by manipulating specific policy levers, such as adjusting production rates or setting dynamic prices. Finally, the simulation environment calculates a "reward" based on predefined performance metrics, such as cumulative profit or market share. This reward signal provides the necessary feedback to guide the agent's learning process toward optimal policy discovery [10].

By repeatedly interacting with the SD simulation, taking actions, observing the resulting state and reward, and updating its internal policy, the RL agent learns a strategy that maximizes its cumulative reward over time. This approach is particularly well suited for discovering adaptive policies, where the optimal action depends on the current state of the system [10].

The technical barrier to this integration has been significantly lowered by the development of specialized software libraries. A notable example is SDGym, a low-code Python library that acts as a wrapper to automatically convert SD models (parsed via engines like PySD) into environments that comply with the Gymnasium standard (formerly OpenAI Gym), the de facto interface for RL research. This allows SD practitioners to leverage state-of-the-art

Deep Reinforcement Learning (DRL) algorithms, such as Proximal Policy Optimization (PPO) for continuous control and Deep Q-Networks (DQN) for discrete decisions. These algorithms have demonstrated particular success in learning robust recovery policies that mitigate the "ripple effect" of supply chain disruptions, outperforming traditional heuristics under conditions of uncertainty. This integration offers a powerful alternative to traditional optimization methods in SD, which often struggle with the path-dependent, nonlinear, and high-dimensional nature of dynamic policy problems. RL can uncover novel and counter-intuitive strategies that outperform those designed by human intuition alone [10].

2.3 Existing System Dynamics Software and Tools

2.3.1 Tools/Software for System Dynamics

The practice of System Dynamics (SD) is intrinsically linked to the software tools used for model construction, simulation, and analysis. The current landscape of these tools can be broadly categorized into two groups: established, proprietary platforms with graphical user interfaces (GUIs), and more recent open-source libraries, predominantly developed in Python. While powerful, each category presents a distinct set of trade-offs regarding accessibility, extensibility, and integration with modern computational workflows.

The proprietary software market has long been dominated by tools such as Vensim, Stella/iThink, and AnyLogic. These platforms offer feature-rich, integrated development environments that have been refined over decades. Vensim, for instance, is noted for its strong market position, particularly in published academic research. Stella and its counterpart, iThink, are recognized for their intuitive interfaces, with Stella often targeting academic users and iThink focusing on business applications. AnyLogic stands apart as a multi-paradigm simulation tool that supports SD alongside agent-based modeling (ABM) and discrete event simulation (DES) within a single model. However, academic reviews have noted that the technical interconnection between these different paradigms within AnyLogic can be "rather opaque". The primary limitations of these commercial tools are their closed-source nature, which hinders research reproducibility and methodological extension, and their significant licensing costs, which can be a barrier to access for students and researchers [12, 51, 52, 53].

In response to these limitations, the open-source community has developed several libraries to bring SD modeling into the scientific Python ecosystem. The most prominent of these is

PySD, a library designed to facilitate the integration of SD with data science by "transpiling" models written in Vensim's proprietary language or the open XMILE standard into executable Python code. This allows models developed in traditional GUI environments to be run and analyzed programmatically. Another notable library is BPTK-Py, which provides a framework for both SD and ABM. While these libraries have been instrumental in bridging the gap between SD and Python, they are not without their own architectural limitations. Academic comparisons have highlighted that both PySD and BPTK-Py have incomplete feature coverage relative to their commercial counterparts and specific implementation constraints [13, 14].

This software landscape highlights a critical divide: on one hand, commercial tools (Vensim, Stella, etc.) are feature-rich and user-friendly but closed-source, costly, and difficult to integrate programmatically. On the other hand, open/free tools (PySD, BPTK-Py, InsightMaker) offer accessibility and integration (especially with Python), but often exhibit incomplete feature sets or scalability constraints (e.g., PySD's function gaps [13], BPTK's simplified solver implementations [14]). This landscape underscores the necessity for a novel open framework (such as `OpenCLD`) that is fully extensible and architecturally aligned with modern AI ecosystems [54].

It is also worth distinguishing System Dynamics libraries from general-purpose Ordinary Differential Equation (ODE) solvers, such as Python's `scipy.integrate` or R's `deSolve`. While these tools offer robust numerical integration, they lack the domain-specific abstractions of System Dynamics. They require the modeler to manually flatten the system into a matrix of differential equations, stripping away the semantic layer of 'Stocks,' 'Flows,' and 'Feedback Loops' that makes SD valuable for communication and mental modeling. `OpenCLD` aims to bridge this gap by providing the numerical power of a solver while retaining the high level semantic abstractions of the SD methodology [17, 18].

2.3.2 Feature Comparison: Openness, Extensibility, and Integration with AI

The choice of a modeling tool has profound implications beyond mere functionality; it shapes the potential for scientific inquiry, collaboration, and innovation. A deeper analysis of the existing SD software landscape through the lenses of openness, extensibility, and AI integration reveals a significant technological gap. This gap provides the central justification for a new, Python-native library designed to overcome the architectural barriers that currently constrain

Table 2.2: 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

the field.

Openness and Reproducibility

The scientific community is increasingly grappling with a "reproducibility crisis," where findings are difficult or impossible to verify independently. In computational fields like simulation modeling, open-source software and transparent "model-as-code" paradigms are considered essential components of the solution. Proprietary tools such as Vensim and Stella, by their very nature, present a challenge to this ideal. Their use of closed-source algorithms and proprietary binary file formats means that the model's implementation details are opaque. Researchers who do not have access to an expensive commercial license cannot run, inspect, or validate the models, hindering peer review and collaborative model building. Open source tools like PySD

and BPTK-Py represent a significant step forward by allowing models to be executed within an open and accessible environment, but the problem is not fully solved as long as the model definition itself remains locked within a proprietary GUI tool [9].

Ultimately, the shift toward open-source SD tools is not merely a matter of licensing, but of adherence to the FAIR Data Principles (Findable, Accessible, Interoperable, and Reproducible). Proprietary binary formats violate the 'Interoperability' and 'Reusability' standards, as model logic is often locked within the GUI. A Python-native architecture ensures compliance with these principles by treating the model structure itself as transparent, version-controllable code that can be indexed and reused across different computational environments [55].

Extensibility and the "Model-as-Code" Paradigm

Proprietary, GUI-based platforms offer a fixed set of vendor defined features. While often extensive, this paradigm fundamentally limits the researcher to the tools provided by the software developer. Methodological innovation, such as developing a novel sensitivity analysis technique or a custom integration algorithm, is difficult or impossible. In contrast, a "model-as-code" approach, where the model is defined directly in a general purpose programming language, offers limitless extensibility. A model defined as a Python object can be seamlessly integrated with the vast ecosystem of scientific libraries, such as NumPy for numerical operations or SciPy for advanced statistical functions. This allows researchers to extend their models with custom logic, connect to external data sources via APIs, and embed their simulations within larger computational workflows, such as automated experimental frameworks or digital twins. This code-first philosophy transforms the model from a static artifact into a dynamic, extensible, and integrable component of a larger research pipeline [16].

Integration with AI and Machine Learning

The most critical limitation of the current software landscape lies in its inadequate support for the deep, native integration of SD and AI/ML models. The future of complex systems analysis resides in hybrid modeling, yet existing tools impose significant architectural barriers.

While these tools offer powerful, feature-rich graphical user interface (GUI)-based environments for model construction and simulation, their closed-source architectures, proprietary file formats, and lack of robust, high level Application Programming Interfaces (APIs) render them fundamentally incompatible with modern, automated, and reproducible research work-

flows. Integrating these legacy systems into the Python ecosystem requires cumbersome, low level wrappers or file-based data exchange, creating a rigid barrier that isolates SD from the broader scientific computing community and hinders large scale experimentation and continuous integration [12].

Legacy commercial platforms were not designed for this purpose. Integrating them with ML frameworks typically requires slow, file-based data exchange or the use of brittle, low level wrappers that are difficult to maintain. This approach is only suitable for loosely coupled, sequential integrations and is fundamentally inadequate for more advanced hybrid modeling tasks [3, 12].

The existing Python libraries, while a step in the right direction, carry architectural limitations that prevent seamless integration. PySD, a valuable contribution, functions primarily as a transpiler that cross-compiles existing Vensim or XMILE model files into executable Python code. Because the model is defined externally and then translated into a Python object, its structure is effectively static at runtime. This design imposes a "translation tax": model creation and structural modification remain dependent on external, often proprietary, GUIs; the translation is incomplete, with numerous Vensim functions remaining unsupported; and the resulting Python object is a stateful but structurally rigid representation of the original model, making deep, dynamic integration, such as replacing a flow equation with a live neural network, a complex and non standard task [13].

BPTK-Py faces the "generalist's dilemma" inherent in supporting multiple modeling paradigms (SD and ABM) within a single architecture. To maintain broad compatibility, the framework makes trade-offs in simulation fidelity specific to System Dynamics. For instance, it relies heavily on simple Euler integration methods by default, which can introduce significant numerical errors in models with non-linear feedback loops or stiff dynamics. It lacks the robust, native implementation of higher-order solvers (e.g., Runge-Kutta 4) that are standard in dedicated SD environments, limiting its utility for high-fidelity scientific modeling [14].

These limitations create a significant bottleneck for innovation. Advanced hybrid methods, such as using an SD model as a rich, interactive environment for a reinforcement learning agent or embedding a neural ordinary differential equation within a larger SD structure, require a tool where the SD model is a native, first-class citizen of the Python ecosystem. Such a library would allow for the dynamic, on-the-fly modification and deep coupling of model components with ML algorithms. The absence of such a tool forces researchers into bespoke,

non-reproducible software projects for each new hybrid model, stifling the development of standardized methods and best practices. This clear and unmet need for a foundational, Python-native, and extensible SD library is the central motivation for the development of `OpenCLD`.

2.4 Gap Analysis and Research Contributions

2.4.1 Identified Literature Gaps

The convergence of System Dynamics and Machine Learning is a promising but nascent field, characterized by significant open challenges that limit its widespread application and methodological rigor. A review of the current literature reveals two distinct but interconnected gaps: a theoretical gap concerning the conceptual integration of the two fields, and a methodological gap related to the practical implementation and validation of hybrid models [15].

Recent systematic reviews confirm that the integration of AI into SD use cases remains rare. This reflects a state of "Soft convergence," where the two fields are often applied in parallel or sequentially, but a "Hard convergence," involving deep, synergistic integration, remains largely unexplored. This theoretical gap is perpetuated by the siloed nature of the respective research communities, which have yet to establish a shared language or a unified set of principles to guide the development of truly integrated models [3, 15].

This theoretical divide contributes directly to a significant methodological gap. The field currently lacks standardized frameworks for designing, implementing, and validating hybrid models. Most hybrid applications are bespoke, one-off case integrations, making it difficult to establish best practices or ensure reproducibility. Key methodological challenges identified in the literature include [15, 19]:

The Validation Gap: There is a lack of unified validation frameworks capable of simultaneously assessing the structural plausibility of the SD component (a "white-box" requirement) and the predictive accuracy of the ML component (a "black-box" metric). Current studies often sacrifice one for the other [7].

The Interpretability Barrier: Ensuring causal reliability remains difficult as the complexity of embedded ML models increases. This opacity acts as a critical barrier for high-stakes decision-making systems where understanding the "why" is as important as the prediction [7].

The 'Continuous-Discrete' Application Gap: Significantly advanced hybrid methods like Neural ODEs have emerged for continuous physical systems. However, there is a scarcity of frameworks that translate these capabilities into the discrete event or time-stepped environments typical of socio-economic modeling, where "black-box" learning must coexist with "white-box" policy structures [6].

The Tooling Void: The field lacks accessible, rapid prototyping tools for hybrid architectures. This forces researchers to build bespoke integrations for each study, hindering the reproducibility and standardization of hybrid modeling workflows [6].

The Collaboration & DevOps Gap: Beyond modeling features, a critical friction point lies in the 'Software Development Life Cycle' of SD models. Proprietary tools utilize binary or XML-heavy file formats that are incompatible with modern version control systems (e.g., `Git`). This creates a "Collaboration Gap" where teams cannot effectively diff, merge, or review model changes. Furthermore, the lack of containerization support (e.g., `Docker`) hinders the deployment of SD models into automated Continuous Integration/Continuous Deployment (CI/CD) pipelines, isolating the field from modern MLOps practices [6].

Recent perspectives continue to highlight these open challenges, calling for new frameworks and tools that can bridge the gap between the causal world driven by the theory of SD and the predictive power driven by the data of ML [7, 19].

2.4.2 Limitations of Existing Tools

The technical landscape described in Section 2.3 reveals a critical disconnect: the tools available to researchers actively limit the exploration of deep SD-ML integration. This forces a trade-off between the feature-rich but isolated environments of proprietary platforms and the architectural rigidity of current open-source libraries [3].

While commercial platforms like Vensim and Stella remain the standard for model construction, their closed ecosystems fundamentally conflict with the requirements of modern, reproducible data science. These platforms do offer external connectivity through proprietary APIs or shared libraries (e.g., Vensim's DLL interface), but these solutions function fundamentally as "black-box" wrappers. They allow external scripts to parameterize inputs and retrieve outputs, but they do not expose the internal mathematical structure of the model to the data science pipeline. Furthermore, these interfaces are often platform-dependent (e.g.,

relying on Windows-specific DLLs) and introduce significant inter-process communication overhead. This latency renders them computationally prohibitive for "loop-in-the-loop" hybrid workflows, such as Reinforcement Learning, which require millions of rapid, low-latency interactions between the agent and the simulation environment [3, 12].

Conversely, while existing Python libraries such as PySD and BPTK-Py have bridged the language gap, they have not resolved the structural gap. As detailed in Section 2.3.2, the "transpiler" architecture of PySD renders models as static artifacts rather than mutable code objects. The library is designed to parse a model file (XMILE or .mdl) and compile it into a fixed Python function. This architecture is fundamentally "read-only" regarding the model's causal structure. Attempting to modify the model dynamically, for example, by swapping a specific flow equation for a neural network during a simulation step, would require refactoring the core translation engine entirely. Such a fundamental architectural change would negate the utility of forking the library, as the core advantage of PySD lies in its parsing capability, not its dynamic extensibility. Therefore, a ground-up "model-as-code" approach is required to treat model components as mutable objects [13, 14].

Consequently, the absence of a tool that is both native to the Python ecosystem and architecturally designed for mutability creates a notable gap. Researchers are currently forced to develop custom, non-reproducible integration scripts for each new hybrid model, stifling the development of standardized best practices [7].

2.4.3 Contribution of OpenCLD

This thesis introduces `OpenCLD`, a novel open source Python library designed specifically to address the identified theoretical, methodological, and technological gaps. The contributions of this research are designed to provide the foundational tools necessary to accelerate innovation in hybrid systems modeling.

Addressing the Technological Gap: To address the technological gap of closed and inextensible tools, `OpenCLD` contributes a fully open source, Python-native library. By enabling models to be defined entirely as Python objects, `OpenCLD` removes the dependency on external GUIs and proprietary file formats. This approach directly supports the principles of open science by ensuring that models are transparent, extensible, and fully reproducible, which is critical to overcome the "reproducibility crisis" in computational

modeling. The "model-as-code" paradigm empowers researchers to programmatically construct, modify, and analyze models, aligning SD with modern scientific software development practices [3, 7, 16].

Bridging the Architectural Divide: In response to the architectural limitations of existing libraries, `OpenCLD` contributes a framework designed for deep, bidirectional integration with the AI/ML ecosystem. Unlike transpiler-based approaches, `OpenCLD`'s native architecture allows ML models (e.g. from PyTorch or scikit-learn) to be seamlessly embedded as dynamic components within an SD model, such as representing a complex flow equation. This capability provides the necessary tooling to bridge the methodological gap, allowing researchers to move beyond "Soft convergence" and explore tightly coupled, "Hard convergence" hybrid models. This facilitates advanced applications such as parameter estimation driven by ML, surrogate modeling, and policy optimization through reinforcement learning [7, 15].

Standardizing Hybrid Workflows: To address the methodological gap concerning the lack of standardized and reproducible workflows, `OpenCLD` contributes a robust and accessible foundation for hybrid experimentation. By providing a single, open, and extensible environment, `OpenCLD` lowers the barrier to entry for developing and sharing novel hybrid methods. Its seamless integration with the scientific Python ecosystem (e.g., NumPy, pandas) allows researchers to leverage a vast array of established tools for data analysis, visualization, and validation. This fosters a more unified approach to hybrid modeling, enabling the community to build upon prior work, establish best practices, and develop the standardized validation frameworks that the field currently lacks [3, 43].

Chapter 3

Design and Architecture of OpenCLD

This chapter presents the OpenCLD library. It first introduces the object oriented architecture and class diagram (Section 3.1), then describes the core classes (Sections 3.1.1–3.1.5) and the Model engine (Section 3.1.6), including the built-in validation checks executed at initialization. Section 3.1.7 covers visualization utilities for both single-run and multi-run analyses. OpenCLD is released as an open-source project under the CC-BY-NC-ND-4.0 license. The source code, documentation, and installation instructions are available at `https://github.com/log-lab-polito/OpenCLD`. The package can be installed directly via the Python Package Index at `https://pypi.org/project/opencld/`.

3.1 Library Design

The library uses an object oriented design that maps each core System Dynamics construct to a class with a single, clear responsibility. This structure keeps interfaces small and improves reuse and testability. Unit-aware quantities (Pint `Quantity`) and runtime dimensionality checks enforce dimensional consistency across models. During simulation, model components are stored in dictionaries keyed by name, enabling $O(1)$ lookups in the main loop. Plotting utilities are lazily imported to avoid loading heavy visualization dependencies unless needed. The design supports scalable workflows from single runs to batch multi-run (Monte Carlo) experiments.

Figure 3.2 illustrates the class diagram.

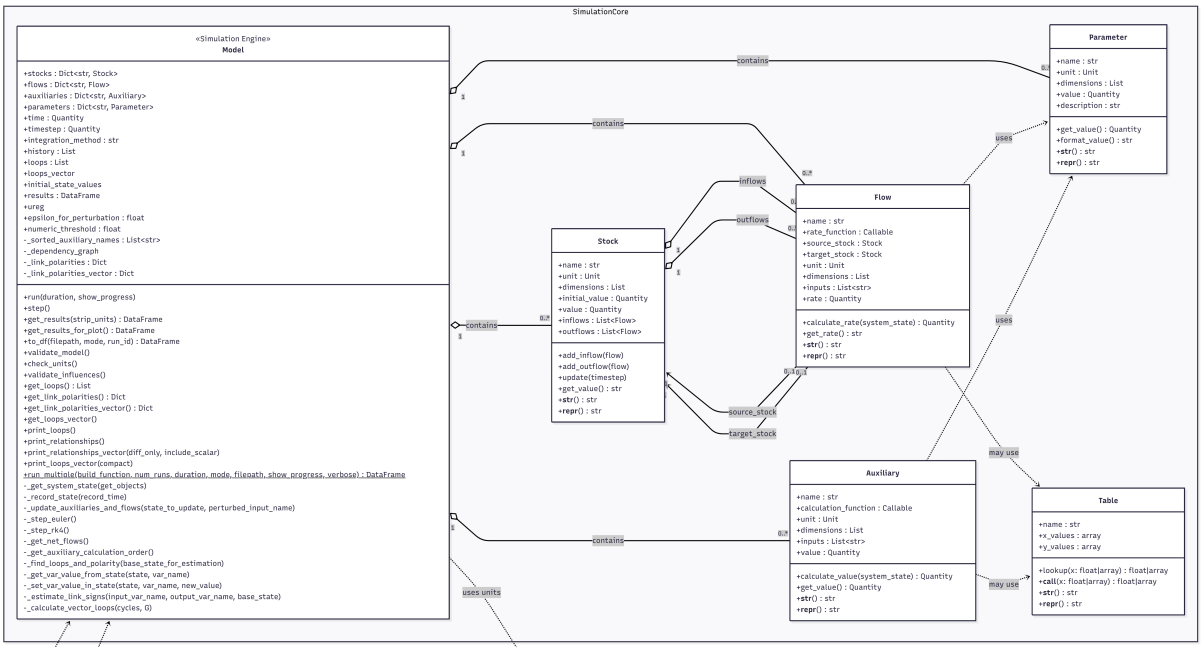

Figure 3.1: Class diagram core classes

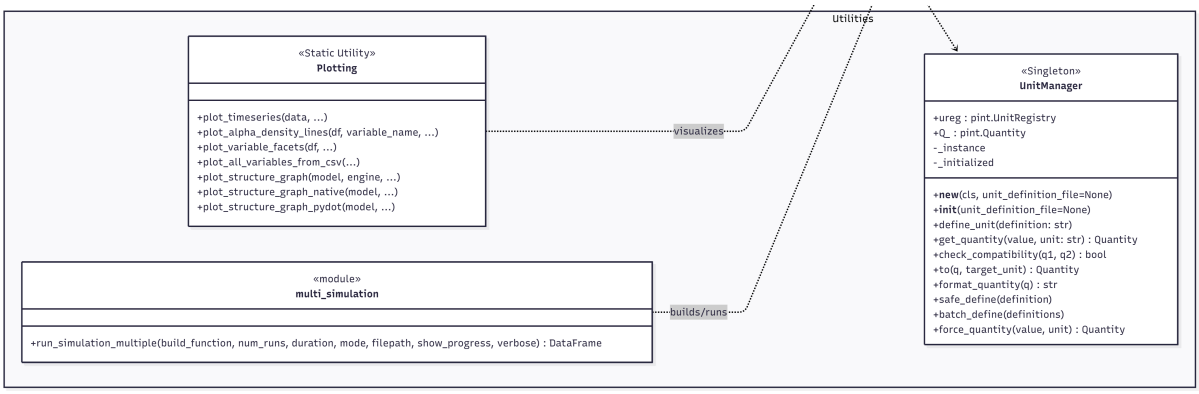

Figure 3.2: Class diagram utilities

Table 3.1: 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__()`

Continued on next page

Table 3.1: OpenCLD core class architecture (continued)

Class	Responsibility	Key attributes	Key methods
`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()`

3.1.1 Stock Class

The `Stock` class represents an accumulation in the system, allowing the user to define quantities that increase through inflows and decrease through outflows. A stock is defined by `name`, `initial_value`, `unit`, and optionally `dimensions`. The `initial_value` can be a scalar or a vector; when `dimensions` are provided, they document the index sets for vectorized stocks. If `unit` is omitted, the stock is treated as dimensionless.

Dimensional consistency is enforced so that flows have unit [stock unit]/[time].

Stocks connect to the model through incoming and outgoing `Flow` objects via `add_inflow()` and `add_outflow()` methods. The time evolution follows the discrete integration rule:

$$x(t + \Delta t) = x(t) + \Delta t \left(\sum_i f_i^{\text{in}}(t) - \sum_j f_j^{\text{out}}(t) \right),$$

where Δt is the simulation step in the chosen timestep unit. For vector stocks, contributing flow rates are expected to be compatible with the stock shape (NumPy broadcasting rules apply; incompatible shapes typically raise runtime errors).

Internally, the class maintains a `value` attribute as the current state during simulation. Users do not set `value` at construction; it is initialized from `initial_value` and updated by `update()`. The current state can be retrieved via `get_value()`, and string representations support debugging and reporting.

```
# --- Preys stock ---
```

```
prey = Stock(
    name="Preys",
    initial_value=40.0,
    unit="animal"
)

# --- Predators stock ---
predator = Stock(
    name="Predators",
    initial_value=10.0,
    unit="animal"
)
```

Listing 3.1: Examples of initializing Stock objects for the Predator-Prey model

3.1.2 Parameter Class

The `Parameter` class represents a constant input to the system used by flows and auxiliaries. A parameter is defined by `name`, and optionally `value` (scalar or vector), `unit`, `description`, and `dimensions`. If `unit` is omitted, the parameter is treated as dimensionless while if `value` is omitted, 1 is automatically assigned. Values can be plain numbers or arrays and are stored internally as `pint` quantities, ensuring dimensional consistency in computations.

Each parameter is uniquely identified by `name`. The current quantity is accessible as the `value` attribute or via `get_value()`. For human-readable output, `format_value()` returns a compact scalar magnitude with unit (primarily intended for scalar parameters); `__str__` includes shape information for vector parameters. When `dimensions` are provided, they document index sets for vector values (informational metadata).

```
birth_rate = Parameter(
    name="alpha",
    value=0.50,
    unit="1/day",
    description="Prey birth rate"
)

predation_rate = Parameter(
    name="beta",
```

```
    value=0.02,
    unit="1/(animal*day)",
    description="Predation rate"
)
```

Listing 3.2: Initializing Parameters for the Predator-Prey model

3.1.3 Auxiliary Class

The `Auxiliary` class represents a computed variable (scalar or vector) that depends on other variables of the system. It is defined by `name` and `calculation_function`, with optional `unit`, `inputs`, and `dimensions`. If `unit` is omitted, the auxiliary is treated as dimensionless. When `dimensions` are provided, they document index sets for vectorized auxiliaries (informational metadata).

The class evaluates its state via `calculate_value(state)`, which runs the user-defined function, wraps the result into a `pint.Quantity` using the declared unit, performs dimensionality checks, and stores the result internally as `value`. Users do not supply `value` at construction; it is computed during simulation. The current quantity is available through `get_value()` as a `pint.Quantity` (from which magnitude and unit can be accessed).

`inputs` lists the upstream model variable names (e.g., `Stock`, `Parameter`, `Auxiliary`, or `Flow` names) used for dependency ordering and for the Polarity Detection Algorithm (see 3.1.6).

```
def prey_eaten_eq(state):
    prey_val = state["stocks"]["Preys"].value
    predator_val = state["stocks"]["Predators"].value
    pred_rate = state["parameters"]["beta"].value
    return pred_rate * prey_val * predator_val

prey_eaten = Auxiliary(
    name="Prey Eaten",
    calculation_function=prey_eaten_eq,
    unit="animal/day",
    inputs=["Preys", "Predators", "beta"]
)
```

Listing 3.3: Auxiliary example: calculating prey eaten per day in the Predator-Prey model

3.1.4 Flow Class

The `Flow` class represents a rate that transfers quantity from an optional `source_stock` to an optional `target_stock`. A flow is defined by `name`, `rate_function`, and optionally `unit`, `inputs`, and `dimensions`. If `unit` is omitted, the flow is treated as dimensionless; flows connected to dimensional stocks must declare a unit compatible with [stock unit]/[time].

The rate can be scalar or vector. The `rate_function` returns either a scalar value or an array; constant flows are modeled by a function that returns a constant. When `dimensions` are provided, they specify index sets for vectorized flows (informational metadata).

`inputs` lists the upstream model variable names that the flow depends on (e.g., `Stock`, `Auxiliary`, or `Parameter` names). These names are used for dependency graph construction and the Polarity Detection Algorithm (see 3.1.6).

The method `calculate_rate()` evaluates the user function at the current simulation state, wraps the result with the declared `unit` when needed, and validates dimensionality. Flows attach to stocks via `add_outflow()` and `add_inflow()` (also auto-wired when `source_stock` or `target_stock` are passed at construction). The current rate is available through `get_rate()`, and string representations report magnitude, unit, and shape for debugging.

```
def prey_births_eq(state):
    return state["parameters"]["alpha"].value * state["stocks"]["Preys"].
    value

prey_birth = Flow(
    name="Prey Birth",
    rate_function=prey_births_eq,
    unit="animal/day",
    inputs=["Preys", "alpha"],
    source_stock=None,
    target_stock=prey
)
```

Listing 3.4: Flow example: defining a flow for prey births in the Predator-Prey model

3.1.5 Table Class

The `Table` class provides a way to approximate nonlinear relationships through a set of discrete breakpoints. It includes attributes `x_values`, `y_values`, and `name`, ensuring that each table is clearly defined. The class evaluates its behaviour through the `lookup()` method, which performs linear interpolation between the defined breakpoints (and returns boundary values outside the breakpoint range). The `Table` class also overloads the call operator, allowing users to write expressions such as `y = myTable(x)`, making it intuitive to use within model equations. By providing this functionality, the class enables the representation of empirical data and complex functions efficiently.

3.1.6 Model Class

The `Model` class is the architectural cornerstone of the OpenCLD library, serving as the primary engine for the entire modeling workflow. It is designed as a high level orchestrator that encapsulates the complete simulation lifecycle, providing a unified interface for model compilation, validation, execution, and analysis. Its core purpose is to transform a collection of user-defined, declarative components, which can represent both scalar and vector quantities, into a runnable, introspectable dynamic system.

Initialization and Model Compilation (`__init__`)

The constructor of the `Model` class serves as the primary entry point for model compilation and validation. Its workflow is designed to be robust, providing immediate feedback on model integrity through a multi-stage setup process.

First, the constructor accepts lists of user-defined model components (stocks, flows, auxiliaries, parameters) and converts them into internal dictionaries, where component names serve as keys. This design choice provides $O(1)$ average time complexity for component lookups, which is critical for efficient access during the iterative calculations of the main simulation loop. A significant feature of this architecture is its native support for vectorized components, allowing `Stock`, `Flow`, and `Auxiliary` values to be represented as `numpy` arrays or `pandas` Series, thus enabling the modeling of disaggregated systems (e.g., by region or product type) within a single `Model` instance.

Next, it establishes the simulation's temporal context. The user-provided scalar `timestep`

and string `timestep_unit` are converted into a `pint.Quantity` object, ensuring that all subsequent time-based calculations are dimensionally aware. Key attributes such as `self.epsilon_for_perturbation` and `self.numeric_threshold` are also initialized to configure the sensitivity of the numerical polarity detection algorithm.

Before execution, a critical pre-computation and validation sequence is performed. The `_get_auxiliary_calculation_order()` method constructs a directed graph of dependencies among `Auxiliary` variables and performs a topological sort using the `networkx` library, guaranteeing a valid computational sequence and inherently detecting algebraic loops. Concurrently, the `validate_model()` method performs essential checks: `check_units()` enforces dimensional consistency across the model, and `validate_influences()` executes a "dry run" calculation of all dynamic components at $t = 0$ to catch errors in user-defined functions early.

Finally, the `__init__` method calculates the initial state of every component at $t = 0$. This fully computed state is immediately recorded via `_record_state()` to ensure the simulation output includes the pristine initial conditions. This state also serves as the baseline for the automated structural analysis. The `_find_loops_and_polarity()` method constructs the model's complete causal dependency graph, finds all simple cycles, and determines the polarity of each causal link via the `_estimate_link_sign()` numerical perturbation algorithm, providing immediate insight into the model's feedback structure.

The Simulation Execution Engine (`step()` and `run()` methods)

Understanding the execution mechanics of the `run()` and `step()` methods is essential to grasping why the "Model-as-Code" paradigm enables Hard Convergence with ML components, while traditional SD tools cannot. This subsection describes the full control flow of a simulation run.

State Representation and the System State Dictionary Before describing the loop itself, it is necessary to understand how the model's state is represented and passed between components. At every point during execution, the complete system state is encapsulated in a Python dictionary assembled by the internal `_get_system_state()` method. This dictionary maps category keys (`'stocks'`, `'flows'`, `'auxiliaries'`, `'parameters'`, `'time'`) to the live, mutable Python objects registered with the model. This is a critical archi-

tectural decision: because the dictionary contains direct references to the component objects (not serialized copies), any external Python component, including a neural network or an RL agent, can read and modify the simulation state at any point in the loop without any data conversion or file I/O overhead. This is the low-level mechanism that makes the bidirectional in-memory data flow described in Chapter 1 possible.

Execution Order and the Topological Sort A core challenge in evaluating a System Dynamics model at each step is respecting the algebraic dependencies among `Auxiliary` variables. An auxiliary may depend on the output of another auxiliary, which in turn depends on a stock value. Evaluating these in the wrong order produces incorrect results. The `_get_auxiliary_calculati` method, called once during `__init__`, resolves this by constructing a directed acyclic graph (DAG) of inter-auxiliary dependencies and applying topological sorting (via `networkx`). The resulting ordered list, stored as `self._sorted_auxiliary_names`, is reused at every single simulation step, ensuring that auxiliaries are always evaluated in a causally valid sequence without repeated graph traversal.

The per-Step Execution Protocol The `step()` method advances the simulation by one timestep Δt, executing the following protocol:

1. **Gather State:** Call `_get_system_state(get_objects=True)` to assemble the live state dictionary.
2. **Evaluate Auxiliaries:** Iterate over `self._sorted_auxiliary_names` and call `calculate_value(state)` on each `Auxiliary` in topological order. Each call updates the auxiliary's `.value` attribute in place.
3. **Evaluate Flows:** Iterate over all `Flow` objects and call `calculate_rate(state)`, updating each flow's `.rate` attribute. Because auxiliaries were already recalculated in step 2, flows may safely depend on auxiliary outputs.
4. **Update Stocks:** Iterate over all `Stock` objects and call `stock.update(timestep)`, which applies the net flow rate (sum of inflows minus outflows) to advance the stock's value by Δt.
5. **Advance Clock and Record:** Increment `self.time` by `self.timestep` and append the new state snapshot to `self.history`.

For improved numerical accuracy, a fourth-order Runge-Kutta (`rk4`) method is also supported. Rather than evaluating derivatives once at the beginning of the interval (Euler), RK4 evaluates the full auxiliary-flow-stock pipeline four times per step at intermediate substates, then applies a weighted average of the resulting slopes. The structural execution protocol is identical; only the number of evaluations per step changes.

The `run()` Controller and Re-entrancy The `run(duration)` method is the primary user-facing interface. Its first action is to reset the simulation to its initial conditions: all stock values are restored from their stored `initial_value` attributes, the clock is set to zero, and the history log is cleared. This re-entrant design ensures that consecutive calls to `run()` are fully independent, making parameter sweeps and multi-scenario analyses straightforward. After recording the $t = 0$ state, the method enters the main loop, calling `step()` for each of the `duration / timestep` discrete intervals and returning the final results as a `pandas.DataFrame`.

The `step()` method is also exposed as a public API. This allows external code to bypass the `run()` loop entirely and drive the simulation manually, one step at a time. This is the technical enabler of the "Loop-in-the-Loop" hybrid architectures demonstrated in Chapter 4, where an ML agent intercepts execution between steps to observe the system state and inject a control action before the next integration step proceeds.

Data Retrieval and Formatting (`get_results`, `to_df`)

The library provides powerful methods for accessing simulation data. The primary retrieval method, `get_results(strip_units=False)`, transforms the simulation history into a `pandas.DataFrame`. This process is optimized by first creating a list of flattened dictionaries, which is then passed to the `pandas.DataFrame()` constructor in a single, efficient operation. The `strip_units` parameter allows the user to retrieve either a DataFrame with `pint.Quantity` objects or one with simple scalar magnitudes.

Building on this, the `to_df()` method provides a utility for advanced data analysis. It transforms the wide-format DataFrame from `get_results()` into a "tidy," long-format DataFrame with columns `['time', 'run_id', 'variable', 'value', 'type']`. This format is the standard for many modern visualization and statistical analysis libraries and is essential for handling the output of multi-run experiments, such as those conducted by the

`run_simulation_multiple` function.

Analysis and Reporting Methods

The `Model` class includes built-in methods for the textual reporting of its structural analysis, allowing for rapid model inspection. The `print_relationships()` and `print_loops()` methods provide formatted, human-readable text summaries of the structural analysis performed at initialization. `print_relationships()` lists every causal link and its calculated polarity, while `print_loops()` groups the identified feedback loops by their polarity (Reinforcing, Balancing, or Neutral), allowing for a quick assessment of the model's dominant feedback structures.

Polarity Detection Algorithm (`_estimate_link_sign()` method)

The internal `_estimate_link_sign()` method serves as the engine for the automated structural analysis feature, algorithmically determining the polarity ($+1$, -1, or 0) of any causal link within the model. The process begins with a definitional check, an optimization that handles the special case of a Flow to a Stock link. For these direct relationships, polarity is determined structurally ($+1$ for an inflow, -1 for an outflow) without requiring numerical computation.

For all other links, the algorithm proceeds with a robust relative numerical perturbation. It perturbs the value of the input variable by a small percentage (e.g., 0.1%) of its value at $t = 0$, creating both a "plus" and a "minus" state. This relative approach ensures that the perturbation is appropriately scaled to the magnitude of the variable being tested. The effect of this change is then propagated through the system's causal chain by recalculating all dependent variables.

Finally, the polarity is determined by comparing the output variable from the "plus" and "minus" states. For vectorized (`NumPy` array) variables, the algorithm calculates the difference between the perturbed output arrays and sums the total change. If the absolute magnitude of this total change falls below a pre-defined numerical threshold, the link polarity is considered neutral (0); otherwise, the sign of the total change determines the polarity. This numerical differentiation approach allows the library to automatically and reliably determine link polarities for both scalar and vectorized user-defined functions, regardless of their mathematical complexity.

Unit Management

The unit management system guarantees dimensional consistency across all model components. It is implemented through the `UnitManager` class, which uses the `Pint` library and relies on a predefined unit registry. Whenever an undefined unit is encountered, it is either registered as a new base unit or derived from an existing one using the appropriate SI prefix.

A single global instance of `UnitManager` is created at library import and made available through the global `units` interface. This design allows all classes, including `Stock`, `Flow`, `Auxiliary`, and `Parameter` to rely on the same registry when defining, checking, or converting quantities. Users can extend the registry by placing a `units.txt` file in the working directory (loaded automatically at import); if no local file is present, OpenCLD falls back to the packaged `my_units.txt` definitions.

```
model = Model(
    stocks=[prey, predator],
    flows=[prey_birth, prey_death, predator_birth, predator_death],
    auxiliaries=[prey_eaten],
    parameters=[birth_rate, predation_rate, conversion_rate, death_rate],
    timestep=0.01,
    timestep_unit="day"
)

# Run structural analysis and print loops
model.print_relationships()
model.print_loops()

# Simulate for 100 days
model.run(duration=100)
results = model.get_results()
```

Listing 3.5: Creating and running the Predator-Prey Model

3.1.7 Plotting Class

In OpenCLD, visualization is handled by a dedicated `Plotting` class, architected as a modular and extensible toolkit. This design decouples the visualization logic from the core simulation engine (`Model` class), adhering to the principle of separation of concerns. By implementing its methods as static (`@staticmethod`), the `Plotting` class functions as a namespace

for a suite of visualization utilities that operate on standardized `pandas.DataFrame` inputs, primarily the "tidy" long-format data produced by the `Model.to_df()` method and the `run_simulation_multiple` experiment runner.

The primary methods provided by the `Plotting` class address the key visualization needs in System Dynamics: analyzing time-series data from a single run and interpreting the results of stochastic, multi-run (Monte Carlo) analyses.

Time-Series Visualization

The `plot_timeseries` method serves as a general-purpose tool for creating standard line graphs from simulation data. It is designed with flexibility in mind, accepting data as a `pandas DataFrame` (its primary input format) or other iterable data structures. The method allows users to specify which `columns` to plot, set custom labels and a `title`, and control aesthetic properties such as figure size and grid visibility. Its primary function is to render the dynamic behavior of one or more model variables over time, which is the most fundamental form of output analysis in System Dynamics.

Visualization of Uncertainty

A key feature of the OpenCLD library is its ability to support stochastic modeling and uncertainty analysis, and the `Plotting` class provides specialized methods to visualize the results of these multi-run experiments.

`plot_alpha_density_lines`: This method is designed to visualize the behavior of a single variable across numerous simulation runs. It takes a tidy-format DataFrame and pivots it to align all runs for a specified `variable_name` against the time axis. It then plots each individual run as a highly transparent line (by controlling the `alpha` parameter). The resulting overlap of lines creates a visual density map, often referred to as a "spaghetti plot," which intuitively communicates the range and likelihood of different behavioral trajectories. To provide a measure of central tendency, the method also calculates and overlays the median trajectory across all runs, offering a clear summary of the most probable system behavior.

`plot_variable_facets`: For a comprehensive overview of a multi-run experiment, the `plot_variable_facets` method generates a grid of subplots, with each subplot

dedicated to a different variable in the model. This is achieved by leveraging the `FacetGrid` functionality of the `seaborn` library. The method automatically filters for dynamic variables (stocks, flows, and auxiliaries) and arranges them into a grid. Within each facet, it renders a density line plot identical to that produced by `plot_alpha_density_lines`, including the median overlay. This approach is exceptionally powerful as it allows a modeler to assess the propagation of uncertainty through the entire system at a single glance, comparing the behavioral envelopes of interconnected variables side-by-side. The method provides extensive customization options, including control over the facet layout (`col_wrap`), axis sharing, and figure aesthetics.

Chapter 4

Case Studies: Hybrid SD-AI Applications

4.1 Introduction

To validate the capabilities of the `OpenCLD` library, this chapter presents three comprehensive case studies applied to distinct domains: water resources management, supply chain engineering, and microgrid energy optimization. These examples were selected to demonstrate the library's versatility in handling complex dynamic systems characterized by non-linearity, delays, and competing objectives. By integrating physics-based System Dynamics models (the "White Box") with Artificial Intelligence components (the "Black Box"), we enable workflows that are difficult or impossible to implement with traditional, closed-source SD software.

Before presenting the complex case studies, Section 4.2 briefly demonstrates how `OpenCLD`'s design addresses the core limitations identified in Chapter 2, namely, the "Black-Box" model gap, the validation gap, and the extensibility gap. Using classic System Dynamics examples, we illustrate the library's three foundational pillars: declarative syntax for transparent model structure (Lotka-Volterra), strict dimensional consistency to ensure physically meaningful simulations (SIR), and structural modularity that allows dynamic feedback logic to be fully exposed to the Python interpreter (Bass Diffusion).

4.2 Library Capabilities Demonstration

4.2.1 Declarative Syntax and Transparency

`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 4.1, is completely transparent. Furthermore, `OpenCLD` provides built-in automated structural analysis. As demonstrated in Listing 4.1 and Listing 4.2, 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 4.1: 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 4.2: Automated detection of Reinforcing and Balancing feedback loops for the Lotka-Volterra model.

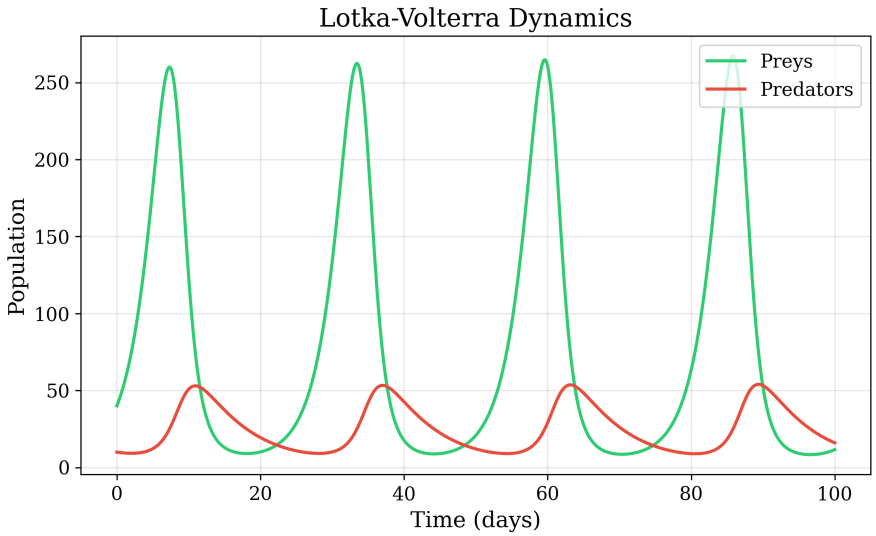

Figure 4.1: Lotka-Volterra Population Dynamics: Oscillations between Preys and Predators.

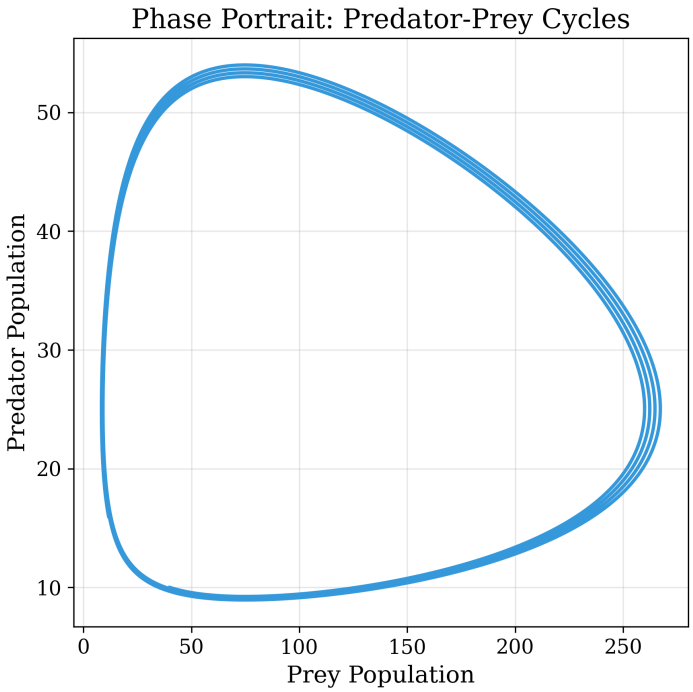

Figure 4.2: Lotka-Volterra Phase Portrait: The closed loop confirms the system's structural stability and periodic behavior.

4.2.2 Dimensional Consistency

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 = sd.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='animal/day',
    inputs=['S', 'I', 'beta']
)
```

Listing 4.3: Unit-aware definition of the SIR infection flow

This feature ensures that the simulated behavior, such as the infection peak shown in Figure 4.3, is not just numerically convergent but physically meaningful.

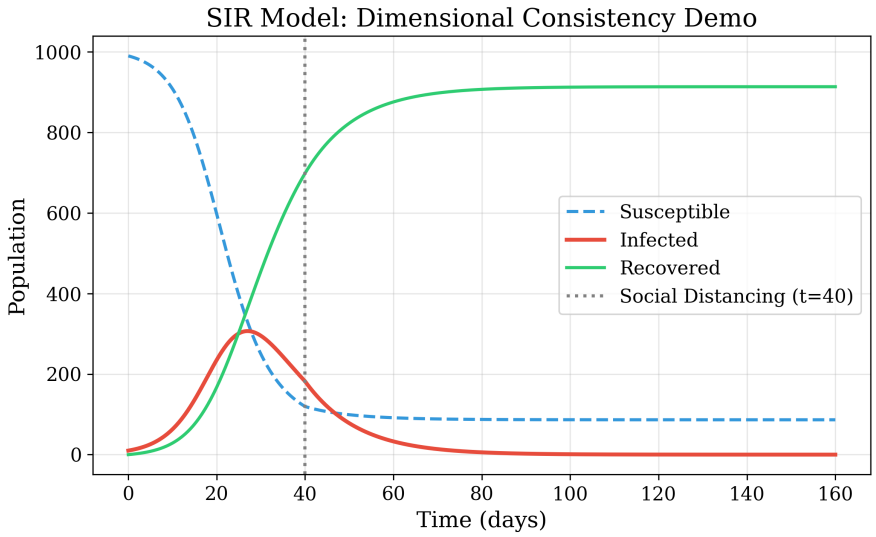

Figure 4.3: SIR Model simulation. The library ensures that the flows moving population between stocks (S $\rightarrow$ I $\rightarrow$ R) conserve mass and maintain dimensional consistency throughout the epidemic wave.

4.2.3 Structural Modularity

`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.

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

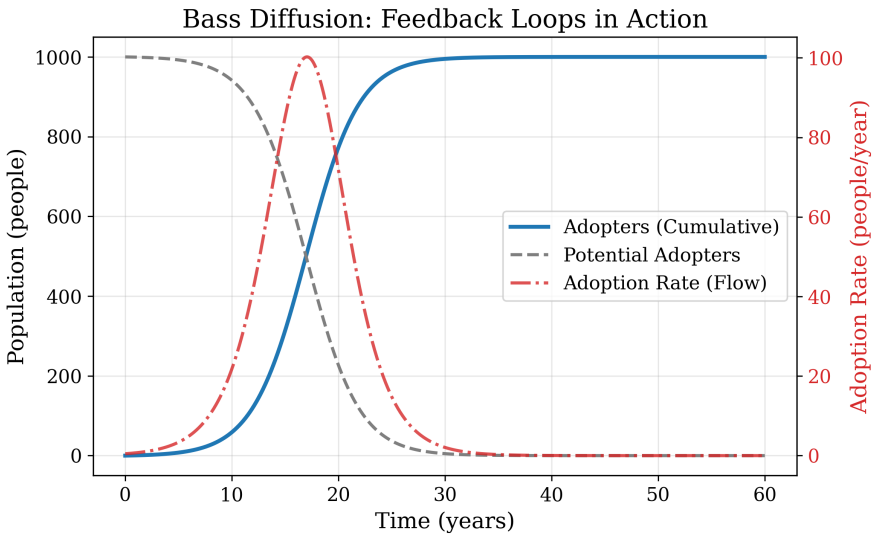

Figure 4.4: Bass Diffusion Model. The S-shaped growth (purple) is driven by the interaction of Innovation and Imitation feedback loops, captured natively in the model structure.

4.3 Case Study 1: Hybrid Flood Control in the Zola Reservoir

4.3.1 System Context and Data Generation

The first system under study is the Zola Reservoir, located in northwest Iran. The reservoir serves a critical dual purpose: supplying water for 15,000 hectares of downstream agricultural land and protecting the region from seasonal flash floods. This section replicates and extends the work of Rahmani & Zarghami (2014).

To ensure a robust experiment, a synthetic dataset was generated to replicate the hydrological characteristics of the Zola basin. A custom Python script (`data_generator.py`) produced 20 years of daily data, including:

- **Inflow:** Modeled with a log-normal distribution to capture the "flashy" nature of river flows, with a mean annual runoff of approximately 150 Million Cubic Meters (MCM) and a seasonal peak in May.
- **Climate:** Daily rainfall and temperature time series were generated with seasonal seasonality (sine waves) and stochastic noise to drive the evaporation and runoff processes.

4.3.2 Implementation with OpenCLD

The reservoir model was implemented as a reusable Python class, `ZolaReservoir`, leveraging `OpenCLD`'s "Model-as-Code" philosophy. This implementation defines the physical structure, Stocks, Flows, and Parameters, directly as Python objects, ensuring both dimensional consistency and structural transparency.

Model Structure and Physics

The system state is represented by a single Stock, `Volume` (MCM), which tracks the water stored in the reservoir. The dynamics are driven by four key Flows, each encapsulating specific physical logic and constraints:

- **Inflow (MCM/day):** Adds mass to the stock based on the external `Inflow_Input` parameter.
- **Controlled Release (MCM/day):** Removes water based on a decision variable. The flow logic includes a safety constraint: it automatically stops releases if the volume drops below the `Min_Volume` (10 MCM), preventing physical violations regardless of the requested release target.
- **Spillway Discharge (MCM/day):** Represents the uncontrolled overflow. Using Python's conditional logic within the flow function, this rate is automatically triggered only when the volume exceeds the `Max_Capacity` (250 MCM), adhering to mass conservation laws without requiring complex "IF-THEN-ELSE" graphical structures.
- **Evaporation (MCM/day):** Modeled as a non-linear function of both volume (as a proxy for surface area) and temperature. The rate equation is defined as a standard Python function:

$$E(V, T) = k_{area} \cdot V^{2/3} \cdot (0.5 + 0.05 \cdot T) \tag{4.1}$$

  This demonstrates `OpenCLD`'s ability to handle custom nonlinearities transparently.

Parameters and Control Interface

The model exposes key variables as Parameters, including static properties like `Max_Capacity` and dynamic inputs like `Temperature` and `Target_Release`.

Crucially, the class exposes a stepping interface (`ZolaReservoir.step()`). At each simulated day, this method updates the exogenous parameters (inflow, temperature) and the control decision (release target) before advancing the `OpenCLD` integrator. This “single-step” execution pattern is the technical enabler for the "Loop-in-the-Loop" architecture, allowing an external AI agent to intervene and set the release policy at every time step.

4.3.3 Hybrid Integration: The "Loop-in-the-Loop" Architecture

The core innovation of this case study is the integration of the SD physical model with an Artificial Intelligence agent. The goal was to replace the traditional "Reactive" control policy (Standard Operating Policy - SOP) with a "Proactive" policy informed by deep learning.

The AI Forecaster

An Long Short-Term Memory (LSTM) neural network was trained using `TensorFlow/Keras` on the first 15 years of synthetic data. The model takes a 7-day sliding window of past rainfall and inflow as input and predicts the river inflow for the next 5 days. This component acts as the "Eyes" of the system, detecting patterns that precede flood events.

The Hybrid Simulation Loop

Unlike traditional SD software that runs the simulation as a monolithic block, `OpenCLD` allows the user to step through time programmatically. This enabled a "Loop-in-the-Loop" architecture:

1. **Observe:** At each time step t, the Python controller extracts the past 7 days of data.
2. **Forecast:** The LSTM model predicts the cumulative inflow for the next 5 days.
3. **Decide:** The controller calculates a "Projected Volume":

$$V_{proj} = V_t + \sum_{i=1}^{5} \text{Predicted\_Inflow}_{t+i} \tag{4.2}$$

   If V_{proj} exceeds the safe operating limit (200 MCM), the controller triggers a "Preemptive Release" command.

4. **Act:** The `OpenCLD` model is advanced by one step (`model.step()`) using the calculated release command. The physical model enforces constraints (e.g., minimum volume, maximum valve capacity) and returns the new state.

This tight coupling, where an external ML model interrupts and influences the SD simulation at every time step, demonstrates the extensibility of the `OpenCLD` architecture.

4.3.4 Results and Discussion

The hybrid system was evaluated on a hold-out test year (2019) containing two distinct stress scenarios: a predictable storm event (Scenario 1) and an unforecastable "Black Swan" shock (Scenario 2).

Scenario 1: The Predictable Storm

In the first scenario, a significant storm event (peak inflow 100 cms) consistent with historical rainfall patterns was introduced. Figure 4.5 illustrates the performance of the LSTM forecaster during this period.

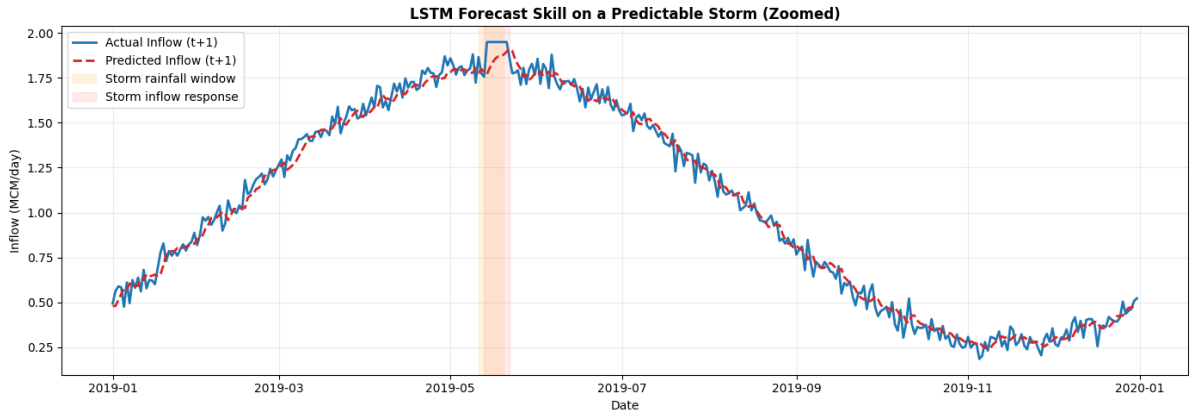

Figure 4.5: LSTM Forecast Skill on a Predictable Storm. The red dashed line (Predicted Inflow) closely tracks the blue solid line (Actual Inflow), demonstrating the model's ability to learn rainfall-runoff patterns.

As shown in Figure 4.5, the LSTM successfully recognized the rainfall precursor patterns, predicting the inflow surge with high accuracy ($RMSE \approx 0.0566$ MCM/day). The forecast (red line) anticipates the rise in actual inflow (blue line), providing the controller with crucial lead time.

The system response (Figure 4.6) demonstrates the efficacy of the hybrid control logic. The AI-informed policy monitored the projected volume and, coupled with standard P-control, managed the inflow event. In this simulation, the reservoir absorbed the 100 cms flood peak without needing emergency releases, as the volume peaked at 161.6 MCM, well below the 250 MCM capacity. The system effectively utilized the available storage, demonstrating that proactive management does not always require aggressive dumping if the forecast indicates the reservoir can safely contain the event.

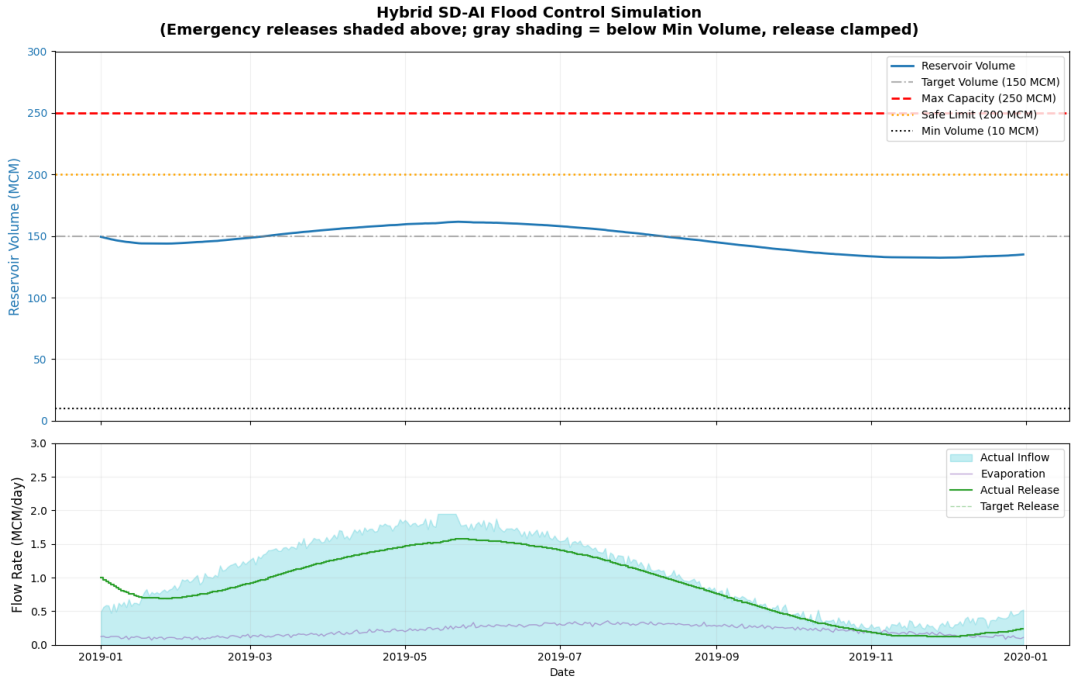

Figure 4.6: Hybrid System Response to Predictable Storm. Top: Reservoir Volume (Blue) rises but stays within safe limits. Bottom: Flow Rates. The system manages the flood with standard operations, avoiding panic releases.

Scenario 2: The Black Swan (Robustness Test)

To test the system's resilience, a "Black Swan" event (a sudden 180 cms inflow spike, significantly larger than training maximums) was injected without any rainfall precursor. To further stress the system, the initial reservoir volume was set to a high level (195 MCM), leaving little buffer space.

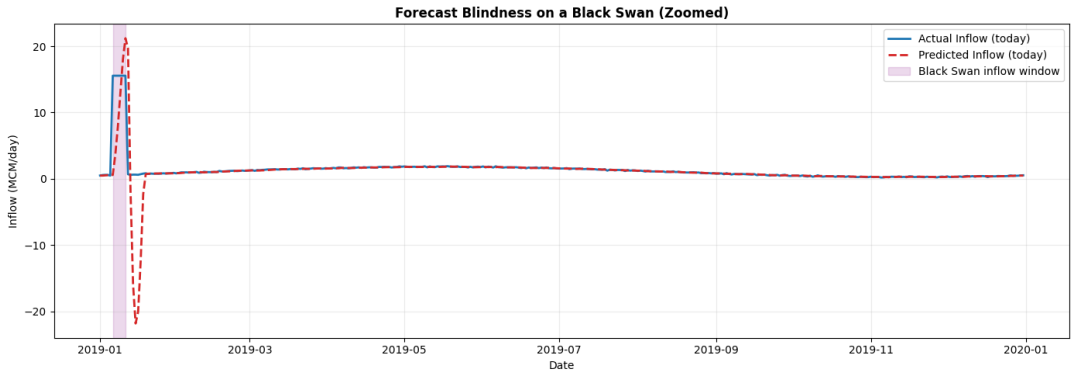

Figure 4.7: Forecast Blindness on a Black Swan Event. The red prediction line remains flat while the actual inflow (blue) spikes vertically, illustrating the "blindness" of the AI to out-of-distribution events.

- **Forecast Blindness:** As expected, the LSTM completely failed to predict the event. Figure 4.7 visualizes this blindness: the forecast (red line) remains near zero while the actual inflow (blue line) surges vertically. The AI treats the event as a complete surprise.
- **Reactive Robustness:** Despite this failure of anticipation, Figure 4.8 demonstrates the robustness of the underlying physical controller. The reactive P-controller detected the rapidly rising water level (Top Panel) and automatically ramped up releases (Green Line, Bottom Panel) once the event began.
- **Result:** Even starting from a high initial volume (195 MCM) and facing a massive 180 cms shock without warning, the system did not spill. The reservoir volume peaked at approximately 197 MCM, staying just below the 200 MCM Safe Limit and well below the 250 MCM capacity. This result highlights a critical safety feature of the hybrid architecture: the AI optimizes performance when possible, but the physical laws and reactive constraints ensure safety when the AI fails.

Validation against Literature

To further validate the library's accuracy, a separate experiment was conducted to reproduce the specific results of Rahmani & Zarghami (2014). The monthly volume-elevation curves for "Existing" and "Development" demand scenarios were simulated.

Figure 4.9 compares the simulated water levels under the "Non-Development" scenario

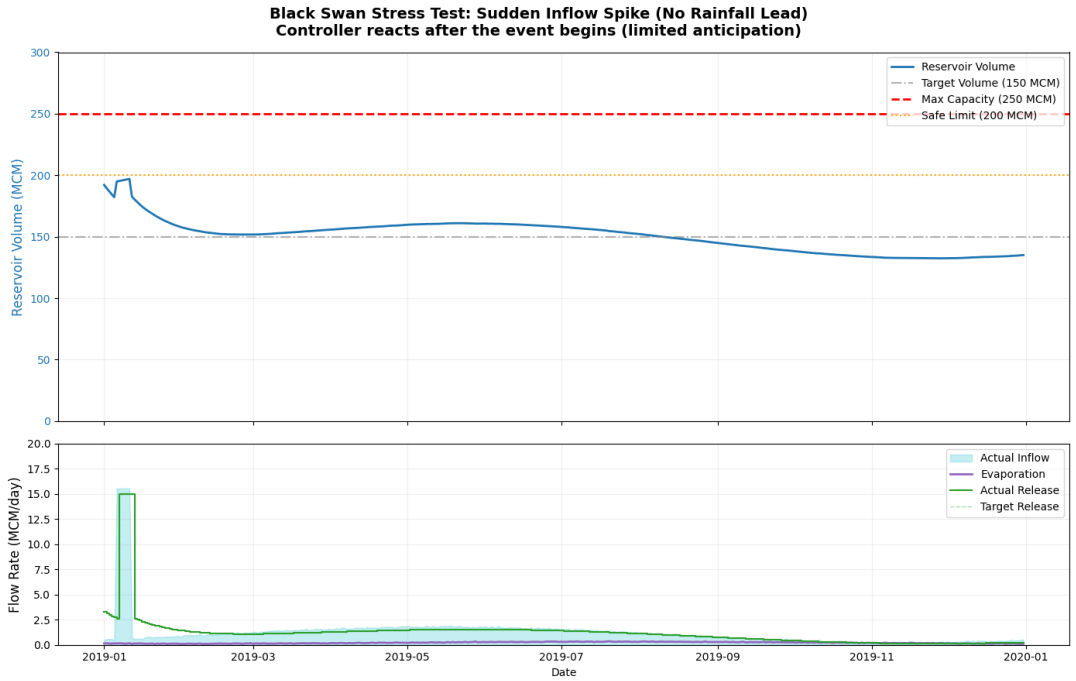

Figure 4.8: Hybrid System Response to Black Swan (Stress Test). Top: Despite the lack of AI warning, the reservoir (Blue) is contained safely below the limit. Bottom: The controller reacts after the event begins, demonstrating robust fallback behavior.

against the historical baseline. The model (colored lines) accurately captures the seasonal drawdown and refill cycles reported in the original study (black line).

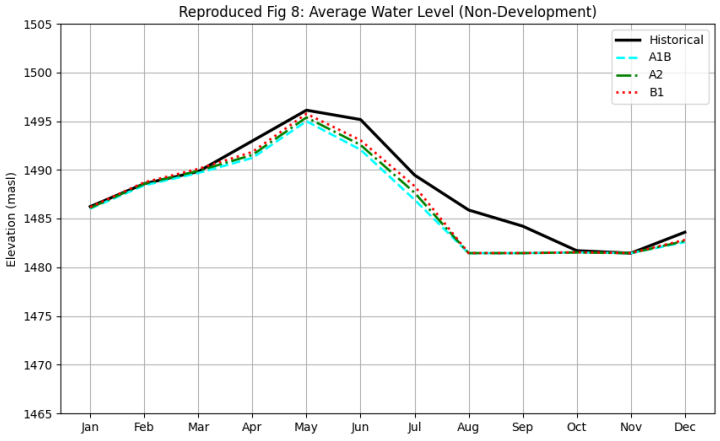

Figure 4.9: Reproduction of Rahmani & Zarghami (2014) Figure 8: Average Water Level (Non-Development Scenarios). The OpenCLD simulation (colored lines) matches the seasonal phase and amplitude of the historical data.

Similarly, Figure 4.10 illustrates the "Development" scenario, where increased downstream

demand leads to more aggressive reservoir drawdown in the summer months. The `OpenCLD` simulation successfully replicates these steeper withdrawal curves, confirming that the library's physical engine produces scientifically valid results comparable to established commercial tools.

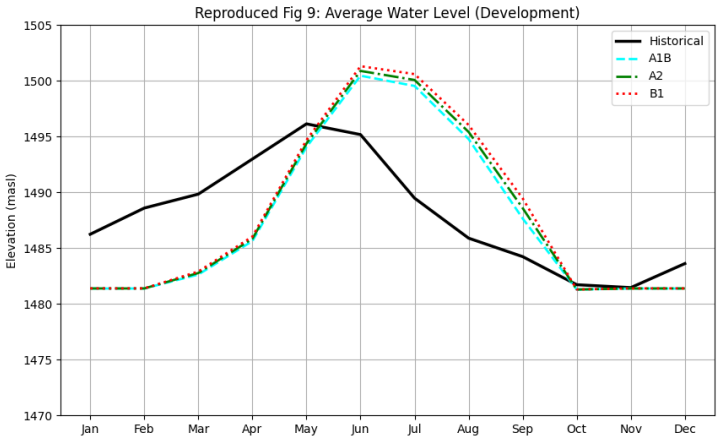

Figure 4.10: Reproduction of Rahmani & Zarghami (2014) Figure 9: Average Water Level (Development Scenarios). The model correctly captures the intensified drawdown caused by increased agricultural demand.

4.4 Case Study 2: Surrogate Modeling for Supply Chain Optimization

4.4.1 Introduction and Motivation: The Cost of Optimization

The second case study focuses on the "Bullwhip Effect" in multi-echelon supply chains, replicating the complex nonlinear system presented by Zhang et al. (2020) [21]. While the previous case study demonstrated "Online" control, this study addresses a different challenge: the computational cost of "Offline" optimization.

Supply chains are notoriously difficult to optimize due to delays (production and transportation lags) and nonlinear constraints (capacity limits, non-negative orders). Traditional analysis often relies on trial-and-error simulation. However, exploring a high-dimensional parameter space with a complex System Dynamics model is computationally prohibitive. If a single simulation takes 1 second, evaluating 1 million parameter combinations would take over 11 days.

This section demonstrates how `OpenCLD` can be used as a high-throughput "Data Factory" to train a fast AI Surrogate Model. The core value proposition here is time: by paying a one-time cost to generate training data, we create a lightweight AI model that can evaluate millions of scenarios in seconds, enabling near-instantaneous optimization.

4.4.2 Methodology: From Physics to Surrogate

Massive Parallel Data Generation (The "Data Factory")

Using `OpenCLD`, a detailed 3-echelon model (Supplier $\rightarrow$ Warehouse $\rightarrow$ Retailer) was implemented. To capture the system's behavior, we conducted a massive factorial experiment:

- **Grid Design:** A 6-dimensional grid was defined, varying smoothing parameters (α_S, α_{SL}), safety stock coefficients (G_d, G_{sys}), demand volatility (σ_D), and decision perspectives (Local vs. Global).
- **Parallel Execution:** Leveraging Python's `multiprocessing` and the efficient design of `OpenCLD`, we evaluated 320,000 unique parameter points. With two stochastic replicates per point, this corresponds to 640,000 simulation runs in total.
- **Performance:** The full factorial dataset was completed in approximately 12 hours on a multi-core CPU. While this initial cost is significant, it is a one-time investment.

Supply Chain Model Implementation with OpenCLD

The Zhang et al. (2020) supply chain was implemented as a first-class SD model using `OpenCLD`'s primitives. All variables were defined with explicit units (e.g., `items`, `items/week`, `week`), ensuring dimensional consistency. The implementation maps the mathematical formulation directly to software objects:

- **Parameters:** The model configuration includes decision weights (α_S, α_{SL}), safety stock factors (G_r, G_d, G_{sys}), and physical constants like mean demand (μ), forecast smoothing factor (θ), production delay ($T_c = 1$), and transport delay ($T_p = 2$).
- **Stocks (State Variables):** The system state is captured by inventory stocks (I_r, I_d, I_w), the supplier work-in-process (Y), and the demand forecast (F). Crucially, the transport delay is not a black box but explicitly modeled as a chain of pipeline stocks (`Trans_Pipe_1`, `Trans_Pipe_2`) to replicate the discrete delay dynamics.

- **Auxiliaries (Policies):** The ordering logic is encapsulated in auxiliary functions (O_r, O_d, $O_{supplier}$). The core experimental variable, the decision perspective, is implemented via a conditional switch in the supplier's order function, choosing between Local Inventory ($I_1 = I_w + I_{transit} + I_d$) and Global Inventory ($I_2 = I_1 + I_r$).

- **Flows (Rates):** The physical movement of goods is governed by flows that enforce conservation and capacity constraints:

  - **Production & Forecast:** `Prod_Start` initiates production based on supplier orders, while `update_F` adapts the forecast based on the smoothing parameter θ.

  - **Shipments (S_w, S_d):** These outflows are constrained by available inventory (e.g., $S_d = \min(Order_r, I_d + Arrivals)$), ensuring the model cannot ship what it does not have.

  - **Arrivals (R_w, R_d):** These inflows represent the completion of production or transport delays.

  - **Consumption (S_r):** The final outflow represents sales to the customer, limited by retailer inventory.

Surrogate Model Training: The Speed Multiplier

The generated dataset was used to train a Random Forest Regressor (a "Surrogate Model") to predict the Bullwhip Effect (BE) based on system parameters.

Log-Transformation of the Target Variable Because the simulated BE spans a very wide range (from stable regimes $BE < 1$ up to highly unstable regimes $BE \gg 1$), we trained the model on a log-transformed target to improve accuracy in the low-BE region where optimization is performed:

$$y = \log(1 + BE) \tag{4.3}$$

During inference, predictions are mapped back using $BE = \exp(y) - 1$.

Accuracy assessment Figure 4.11 shows parity plots for the surrogate. The model achieved an R^2 score of 0.9927 (evaluated on the original BE scale), indicating that the surrogate closely matches the `OpenCLD` simulation across both stable and unstable regimes.

It must be noted that this R^2 metric measures fidelity to the SD simulation, not to physical ground truth. The surrogate model is validated against the synthetic data generated by `OpenCLD`, demonstrating its capability to act as a fast approximation of the simulation engine. Full empirical validation against real-world supply chain data remains a direction for future work.

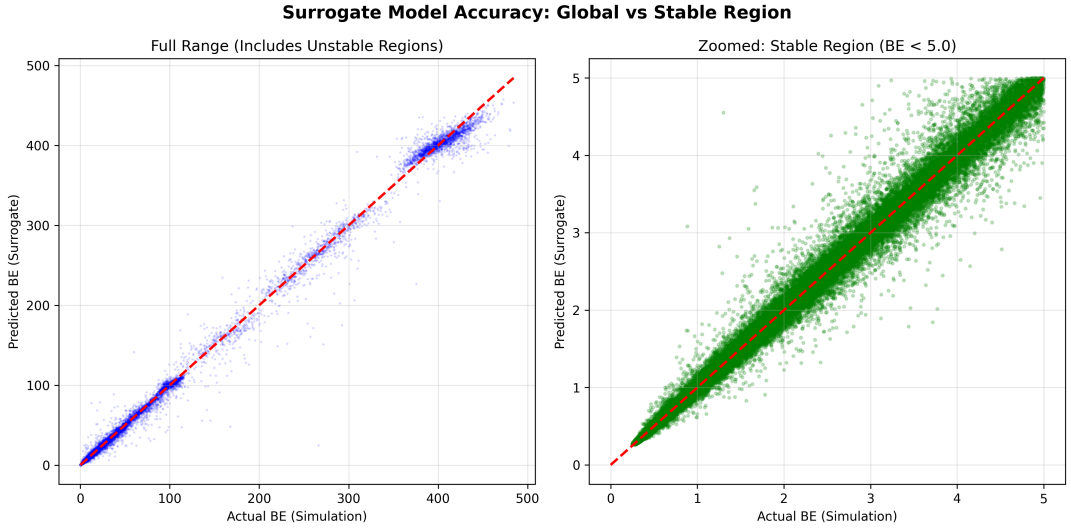

Figure 4.11: Surrogate model accuracy (Random Forest). Left: full range of BE values (including unstable regimes). Right: zoomed stable region ($BE < 5$), showing the model fidelity where optimization and policy selection occur.

4.4.3 Phase 1: Single-Objective Optimization (Min-BE)

To validate the utility of this approach, we first used the surrogate landscape to identify parameter combinations (α_S, α_{SL}) that strictly minimize the Bullwhip Effect.

The Power of Speed The true power of the surrogate approach becomes evident during this optimization phase. Mapping the entire response surface (as shown in Figure 4.14 and Figure 4.15) requires evaluating thousands of points to visualize the "valley" of stability.

- **Using Simulation:** Generating a high-resolution heatmap (e.g., 50×50 grid) would require running 2,500 full simulations. Even at 0.1s per run, this takes minutes for a single slice.
- **Using Surrogate:** The Random Forest can predict the outcome for the same 2,500 points in milliseconds.

This speed allows us to interactively explore the landscape, perform "What-If" analysis, and run global optimization algorithms (like Differential Evolution) that require thousands of function evaluations, all in real-time.

Landscape visualization The AI identifies a deep "valley" of stability where the Bullwhip Effect is minimized ($BE \approx 0.22$). However, this aggressive optimization comes at a cost.

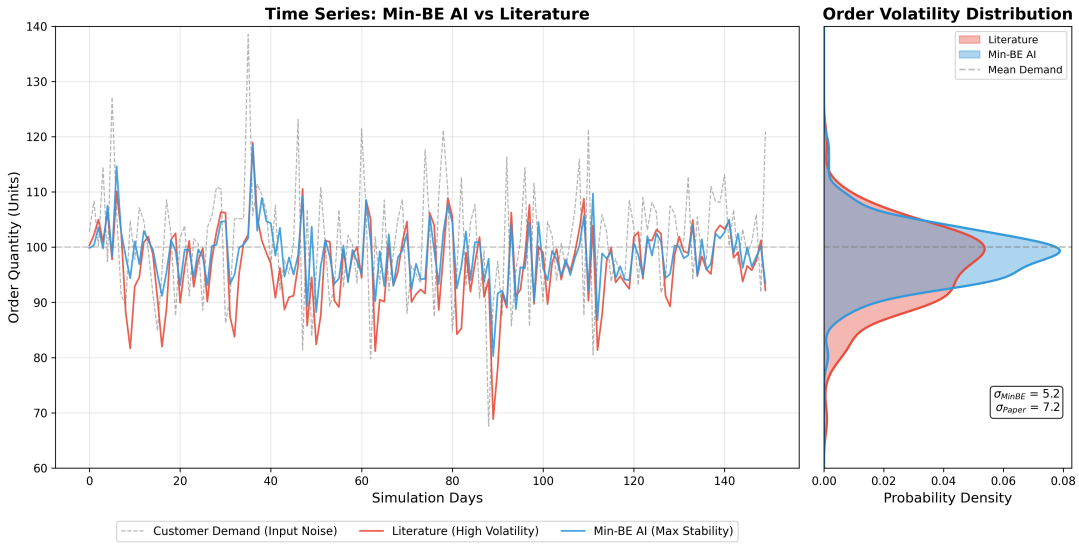

Figure 4.12: Time-series comparison (Min-BE AI vs Literature). The AI solution (Blue) significantly dampens order volatility compared to the Literature baseline (Red), achieving a much lower Bullwhip Effect.

The Problem: Inventory Drift

While the "Min-BE" solution ($\alpha_S \approx 0.01, \alpha_{SL} \approx 0.46$) achieves an impressive 54% reduction in Bullwhip Effect compared to the literature, closer inspection reveals a critical flaw. As shown in Figure 4.16, the aggressive smoothing required to stabilize orders causes the inventory levels to "drift" significantly away from their targets. This "Inventory Instability" is a classic control theory trade-off: you can stabilize the flow (orders) or the stock (inventory), but rarely both perfectly.

4.4.4 Phase 2: Multi-Objective Optimization (Balanced AI)

To address the inventory drift problem, we advanced the methodology to a Multi-Objective Optimization framework. The goal was to find a "Balanced" solution that minimizes Bullwhip Effect while simultaneously minimizing Inventory Instability (RMSE from target).

Pareto Front Generation

We employed a scalarization approach, sweeping a weight parameter w from 0 to 1 to minimize a combined cost function:

$$J = (1 - w)\frac{BE}{BE_{ref}} + w\frac{RMSE_{inv}}{RMSE_{ref}} \tag{4.4}$$

where BE_{ref} and $RMSE_{ref}$ are the baseline values from the literature, ensuring both objectives are normalized.

Figure 4.13 shows the resulting Pareto Front. This curve represents the set of optimal trade-offs. We identified a "Knee Point" (Balanced AI) where a massive improvement in inventory stability is achieved with only a marginal increase in Bullwhip Effect. Generating this Pareto Front requires running the optimizer repeatedly for many different weights. With the surrogate model, generating the entire curve (Figure 4.13) takes less than a second. Doing this with the full simulation model would have been a slow, iterative process.

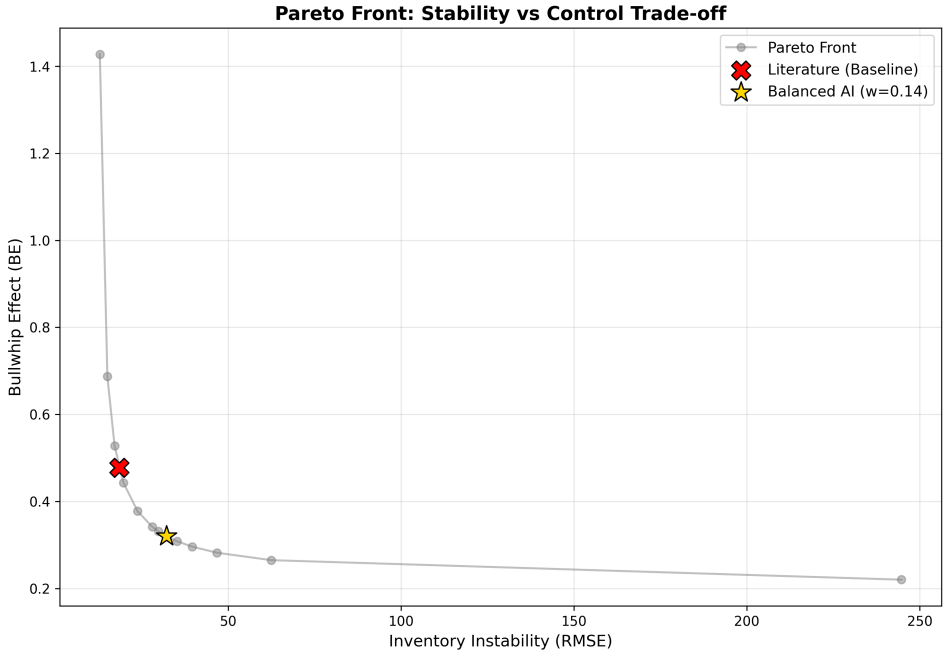

Figure 4.13: Pareto Front of Stability vs. Control. The "Balanced AI" (Gold Star) represents the optimal trade-off, achieving high inventory stability (low RMSE) while maintaining a low Bullwhip Effect.

4.4.5 Final Results: Balanced AI vs. Literature

The "Balanced AI" solution ($\alpha_S = 0.195, \alpha_{SL} = 0.321$) represents a robust, deployable policy. We compared this solution against the Literature baseline and the unstable Min-BE solution across multiple safety stock scenarios.

Landscape visualization Figure 4.14 and Figure 4.15 visualize the complete solution landscape for a representative global scenario ($G_d = 1, G_{sys} = 4$). The Min-BE optimum (Green Star) lies in the deepest part of the valley ($BE \approx 0.22$), while the Balanced AI (Gold Diamond) sits slightly higher up the slope but in a region of greater inventory control.

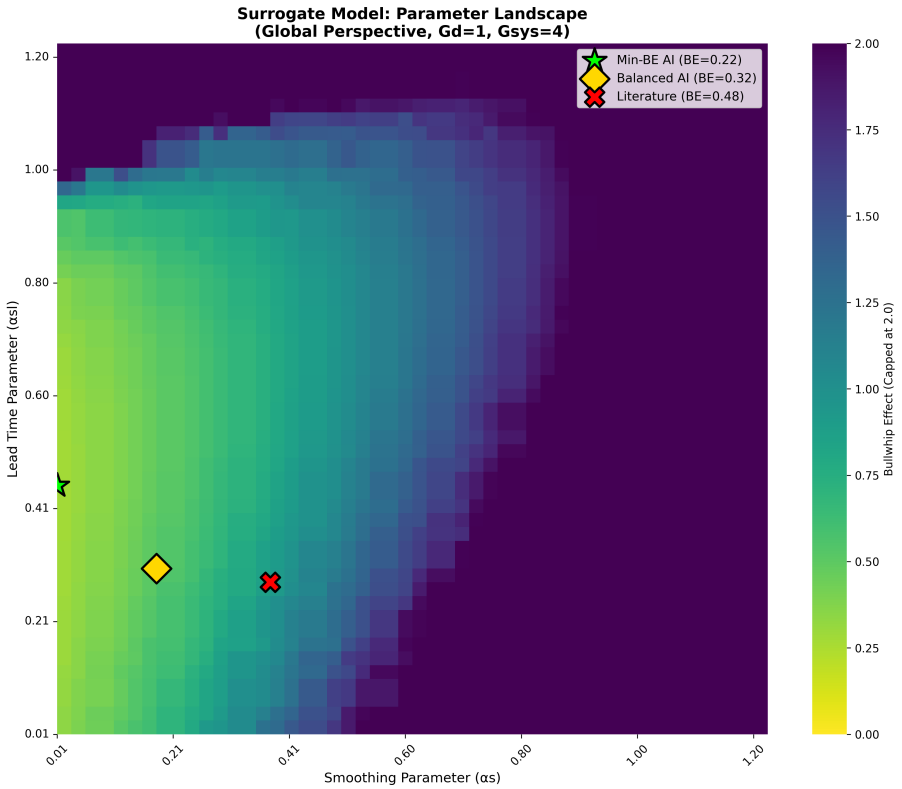

Figure 4.14: Surrogate parameter landscape (global perspective, $G_d = 1, G_{sys} = 4$). The map shows the positions of the three solutions: Literature (Red X), Min-BE AI (Green Star), and Balanced AI (Gold Diamond). Color scale is capped at 2.0 to emphasize the stable region.

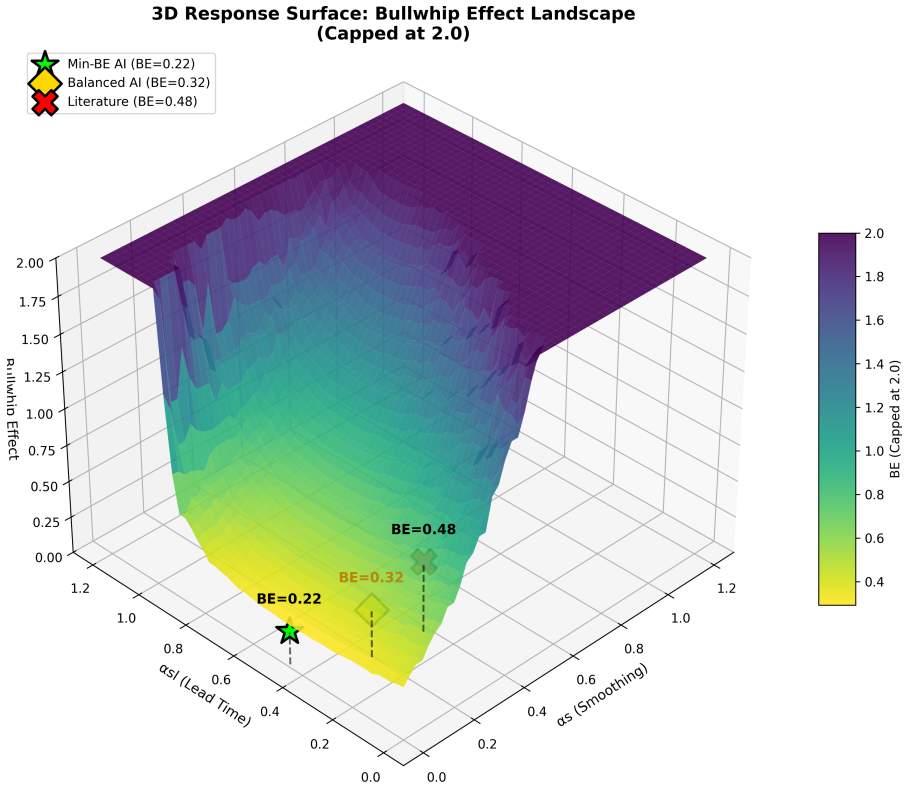

Figure 4.15: 3D response surface (capped at 2.0). The landscape shows the trade-off valley, with the Balanced AI solution positioned to optimize both objectives.

Performance Comparison Figure 4.16 illustrates the dynamic behavior of the three policies. The Balanced AI (Orange) successfully maintains the target inventory level, avoiding the drift seen in the Min-BE solution (Green), while still smoothing orders significantly better than the Literature baseline (Red).

Figure 4.17 summarizes the BE reduction across the three global scenarios. Figure 4.18 provides a qualitative view: the Balanced AI policy produces noticeably tighter order fluctuations, and the distribution panel quantifies the reduction in standard deviation.

Discussion of Findings

The multi-objective optimization process yielded several key insights:

1. **Superior Performance:** The Balanced AI solution reduces the Bullwhip Effect by approximately **33%** compared to the literature baseline ($BE \approx 0.32$ vs 0.48) while maintaining equivalent inventory control.

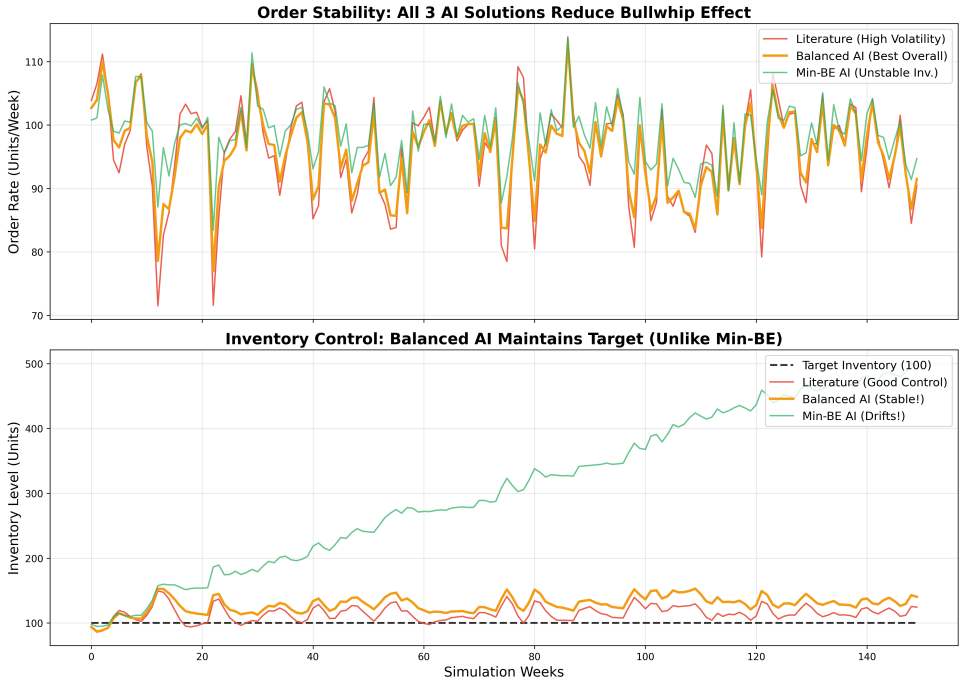

Figure 4.16: The Stability-Control Trade-off. Top: All AI solutions stabilize orders better than Literature. Bottom: The "Min-BE AI" (Green) allows inventory to drift uncontrollably. The "Balanced AI" (Orange) maintains inventory stability similar to Literature while still reducing order volatility.

2. **Safety Awareness:** Unlike naive single-objective optimization, which proposed an unstable policy ($BE \approx 0.22$), the multi-objective approach successfully navigated the trade-off between flow stability and stock control.

3. **Computational Efficiency:** The most significant finding is the transformative speed of the surrogate approach. While generating the training dataset required approximately 12 hours of CPU time, a single surrogate inference completes in approximately 0.1ms, compared to roughly 100ms for a full OpenCLD simulation run. This represents a speedup of two to three orders of magnitude (10^2–$10^3\times$), effectively compressing the physics of the System Dynamics model into a lightweight function. When performing a full optimization sweep requiring thousands of evaluations, this compresses what would require hours of simulation time into seconds. This efficiency unlocks capabilities that are computationally infeasible with raw simulation, such as interactive “What-If” analysis, real-time control, and global sensitivity analysis over high-dimensional parameter spaces.

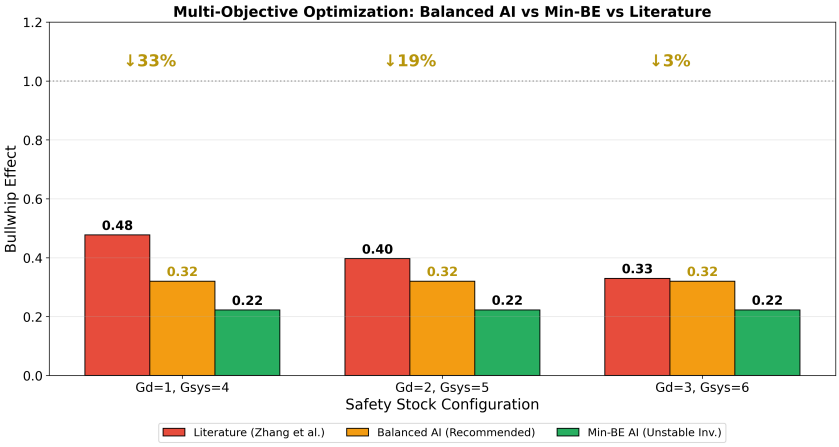

Figure 4.17: Summary comparison across global scenarios. The "Balanced AI" (Orange) consistently reduces the Bullwhip Effect by 30-40% compared to Literature (Red), without the instability risks of the Min-BE solution (Green).

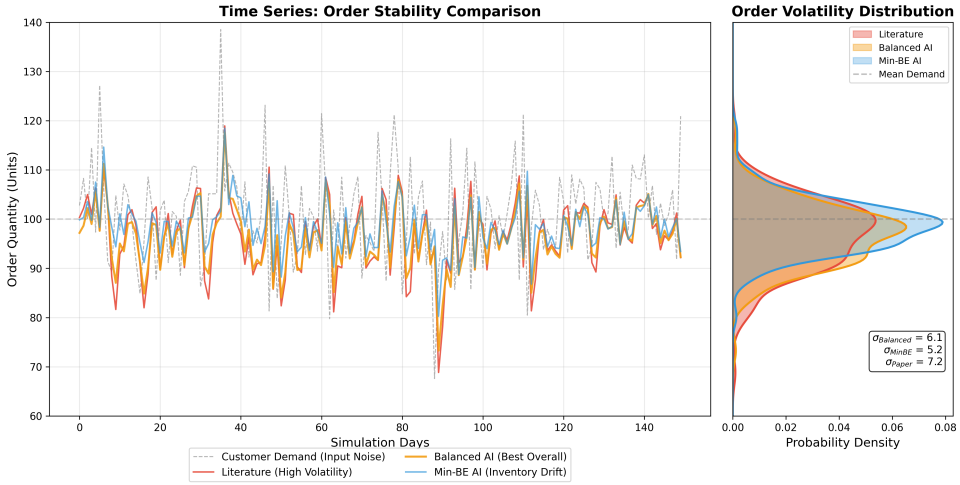

Figure 4.18: Time-series comparison (global scenario $G_d = 1, G_{sys} = 4$). The Balanced AI policy (Orange) produces significantly tighter order fluctuations than the Literature baseline (Red), reducing the standard deviation of orders (σ) while keeping inventory on target.

4.5 Case Study 3: Smart Microgrid Optimization with OpenCLD

To demonstrate the capabilities of the `OpenCLD` library in a realistic setting, we implemented a Simulation-Based Inference (SBI) environment for a residential microgrid. This case study showcases how System Dynamics models can serve as robust Gym environments for training Deep Reinforcement Learning (DRL) agents to solve complex control problems.

4.5.1 Model Implementation with OpenCLD

The microgrid system was implemented as a `MicrogridModel` class, wrapping an `OpenCLD` core. This design separates the physical dynamics from the reinforcement learning interface. The model structure is defined as follows:

- **Parameters:** The system is configured with static physical constants, `capacity` (13.5 kWh), `max_power` (5.0 kW), and `efficiency` (0.95 one-way), and dynamic exogenous inputs updated at every step: `solar` generation, household `load`, grid `price` (buy), `sell_price`, and the agent's control `action`.
- **Stocks:** A single stock, `battery` (kWh), represents the energy reservoir.
- **Flows:** Two flows, `charge` and `discharge`, govern the state changes. These flows utilize a shared physics kernel (`_actual_battery_power`) that ensures physical realism:
  - **Action Translation:** The normalized action $[-1, 1]$ is scaled by `max_power`.
  - **Efficiency Losses:** Charging draws more from the grid than enters the battery ($P_{grid} = P_{batt}/\eta$), while discharging delivers less to the grid than leaves the battery ($P_{grid} = P_{batt} \cdot \eta$).
  - **Physical Clamping:** Rates are dynamically clamped based on available "headroom" (Capacity - SOC) or available energy, preventing overcharging or discharging below zero within the discrete time step.
- **Auxiliaries:**
  - `net_grid`: Calculates the net interaction with the main grid: $P_{net} = P_{load} - P_{solar} + P_{grid\_interaction}$.
  - `cost_rate`: Implements asymmetric pricing logic. If $P_{net} > 0$, cost is calculated using the high retail `price`; if $P_{net} < 0$, revenue is calculated using the lower `sell_price`.
  - `soc`: Computes the normalized State of Charge (E_{batt}/E_{cap}).

Crucially, the class exposes a `step()` method that updates the dynamic parameters (action, weather, prices) before advancing the simulation. This allows the `OpenCLD` model to serve as the state transition function within a standard Gymnasium environment (`MicrogridEnv`).

4.5.2 Realistic Data Integration

A key contribution of this study is the integration of high-fidelity, synchronized datasets for the Italian context (Milan, 2025 scenario), moving beyond synthetic data:

- **Solar Generation**: Hourly irradiance data from PVGIS (JRC) for Milan, processed to account for panel efficiency and system losses.
- **Load Profile**: Real household consumption data from the GREEND dataset (Austrian/Italian households) [56], capturing realistic evening peaks and stochastic usage patterns.
- **Electricity Prices**:
  - **Buy Price (PUN + Spread)**: Based on the Italian *Prezzo Unico Nazionale* (PUN), the national wholesale electricity price, combined with a Time-of-Use (TOU) tariff structure. This structure features a low night-time and weekend tariff (F3 band) and a high daytime peak tariff (F1 band), reflecting real-world consumer energy contracts.
  - **Sell Price (Zonal Price)**: Hourly averaged Prezzo Zonale (NORD) data from GME (Gestore Mercati Energetici, the Italian energy market operator), extracted from 15-minute market settlements. This captures the dynamic, fluctuating value of energy exported back to the regional grid.

4.5.3 Optimization Strategy and Reward Function

We trained a Proximal Policy Optimization (PPO) agent to minimize electricity costs. The reward function design proved critical. A naive "cost minimization" reward led to a degenerate policy where the agent sat idle to avoid efficiency losses.

To guide learning, we implemented a sophisticated shaped reward:

1. **Real Savings**: The primary signal is the difference between the baseline cost (no battery) and the agent's cost.
2. **Physics Constraints**: An "Invalid Action Penalty" was added to punish the agent for attempting to discharge an empty battery or charge a full one, forcing it to internalize the system's physical limits.

3. **Strategic Bonuses**: Small "nudge" bonuses were introduced for:

    - **Solar Self-Consumption**: Incentivizing the absorption of excess solar instead of selling it at low prices.
    - **TOU Arbitrage**: Encouraging charging during low-price hours (night) and discharging during peaks.

4. **State Observation and Forecasting:** The agent's observation space includes the current state of charge, current prices, and a 24-hour look-ahead forecast. The day-ahead electricity prices are provided exactly as they are published in real energy markets (24 hours in advance). However, for solar generation and household load, the agent receives predictions generated by a simple moving average of the previous days, ensuring no future data leakage occurs during training.

4.5.4 Results and Discussion

The trained agent was evaluated on three representative weeks: Best (Summer), Average (Spring /Autumn), and Worst (Winter).

Best Case (Summer - High Solar, High Spread)

The agent demonstrated near-optimal behavior (Figure 4.19). It consistently charged the battery during midday solar peaks (yellow), effectively capturing free energy that would otherwise be sold cheaply. It then discharged this energy precisely during the evening load peaks (purple) when grid prices were highest (€0.35/kWh). This strategy yielded significant savings compared to the baseline, reducing the weekly energy cost from €8.68 to €4.73, a saving of €3.96 (45.6%).

Worst Case (Winter - Low Solar)

In the absence of solar energy, the agent successfully switched to a pure arbitrage strategy (Figure 4.20). It charged from the grid during the cheap night window (€0.15/kWh) and discharged during the morning and evening peaks. This confirms the agent learned to exploit price differentials independent of solar generation. While the absolute savings are modest (€0.24, from €30.67 to €30.43), the agent correctly identifies that arbitrage opportunities are limited by the battery's round-trip efficiency.

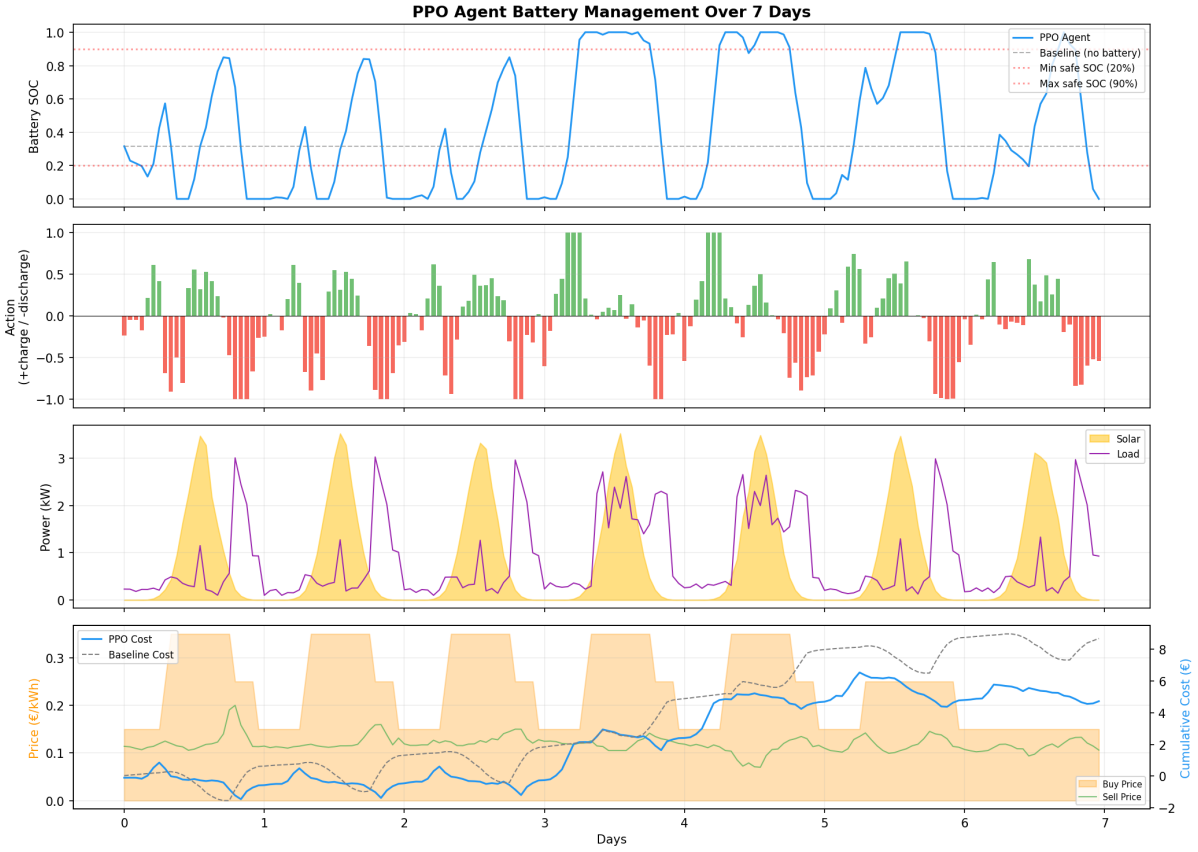

Figure 4.19: Best Case Week (Summer). The agent (blue line, top panel) charges during solar peaks (yellow, third panel) and discharges during evening load peaks (purple), maximizing self-consumption and savings.

The Efficiency Trade-off (Average Week)

An interesting phenomenon emerged in the "Average" week (Figure 4.21). We initially tested an "Active" agent with high behavioral bonuses, forcing it to cycle the battery daily. However, this resulted in a net financial loss compared to the baseline. The reason is the round-trip efficiency (90%). If the price spread between day and night is less than ≈10%, storing energy destroys value.

The final "Profitable" agent correctly learned to idle on days with thin price spreads, avoiding the "churning" trap. It only engaged the battery when the day-ahead forecast predicted a spread large enough to cover the efficiency loss. This selective behavior highlights the strength of the DRL approach: it doesn't just follow a rule-based heuristic ("always charge at noon"); it dynamically evaluates the economic viability of every action based on the 24-hour forecast. In this scenario, the agent incurred a slight cost increase of €1.23 compared to the baseline (€20.82 vs €19.59), which is a significant improvement over earlier "active" iterations that lost

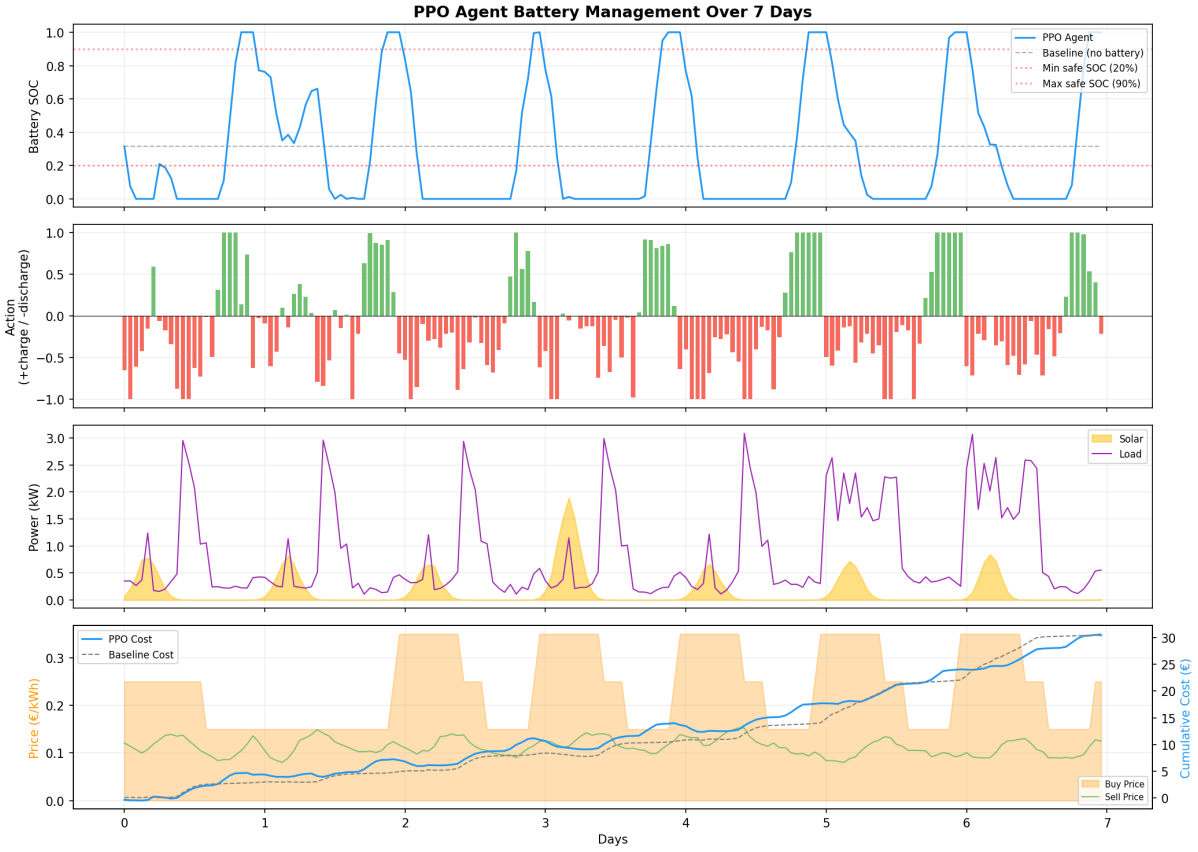

Figure 4.20: Worst Case Week (Winter). With minimal solar, the agent shifts to pure arbitrage, charging at night (cheap prices) and discharging during morning/evening price peaks.

much more, demonstrating the agent's ability to minimize losses in unfavorable conditions.

4.5.5 Summary

This application demonstrates that `OpenCLD` can support complex, data-driven optimization tasks. By coupling a rigorous physical model with real-world economic data, we trained an AI agent that not only learned the physics of energy storage but also discovered sophisticated economic strategies like arbitrage and selective idling. The failure of the forced "Active" model underscores the importance of physics-aware reward shaping: a sustainable energy transition requires solutions that are not just technically feasible, but economically viable.

4.6 Conclusion

These three case studies demonstrate the transformative potential of the `OpenCLD` library. By treating System Dynamics models as standard Python code, we were able to:

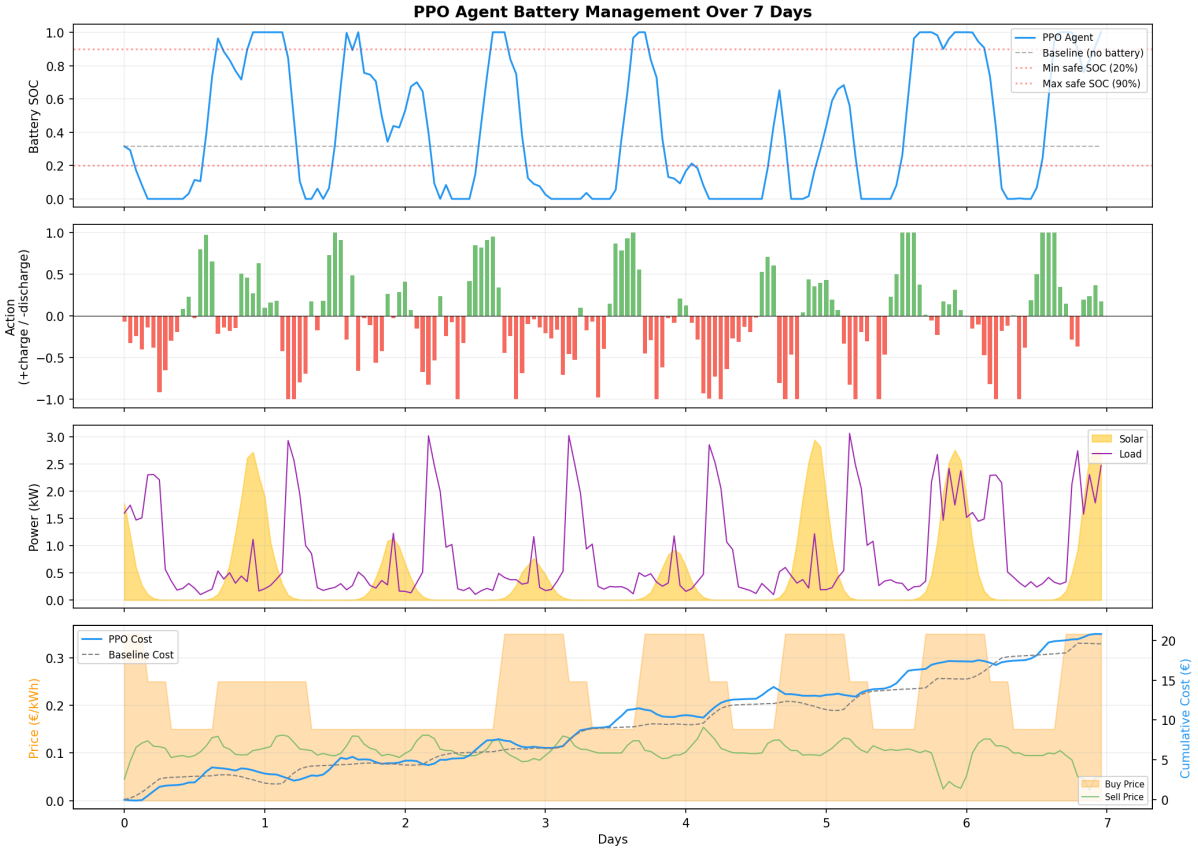

Figure 4.21: Average Week (Efficiency Trade-off). The agent selectively idles on days with insufficient price spreads to cover round-trip efficiency losses, avoiding the "churning" trap that plagued earlier iterations.

1. **Hybrid Control:** Embed a Deep Learning forecaster directly into the simulation loop for proactive flood control.

2. **Surrogate Optimization:** Generate massive datasets to train AI surrogates, discovering optimal supply chain policies that outperform established literature with orders of magnitude faster optimization time.

3. **Policy Learning:** Use the model as a training environment for Deep Reinforcement Learning, enabling an agent to autonomously discover complex, profit-maximizing strategies for microgrid energy management under physical and economic constraints.

These results validate the thesis proposition: that a Python-native, "Model-as-Code" approach effectively bridges the gap between System Dynamics and Data Science.

Chapter 5

Conclusions and Future Work

5.1 Summary of Contributions

The primary contribution of this thesis 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 transpiling 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 particularly significant contribution, and one underappreciated in prior SD tooling, is the library's native interoperability with the broader Python data science ecosystem. Because OpenCLD models are standard Python objects, simulation state can be read, written, and exchanged with any Python-compatible tool at any point during execution, with no file I/O, no serialization overhead, and no custom adapters. A simulation's internal variables can be streamed directly into a `pandas` DataFrame for real-time analysis, passed to a `scikit-learn` model for inference, or consumed by a `PyTorch` training loop in the same process. Conversely, data from external sources, whether live sensor feeds, database queries, or ML model outputs, can

be injected directly into the simulation parameters at each time step. This bidirectional, in-memory data flow eliminates what prior approaches imposed as a "translation tax," and is what makes the three embedded hybrid workflows in Chapter 4 possible without any glue code.

The utility and flexibility of this architecture were empirically validated through three distinct case studies, each targeting a specific mode of hybrid integration:

- **Hybrid Control:** In the Zola Reservoir case study, the library demonstrated the capacity for "Loop-in-the-Loop" control. By embedding a deep learning forecaster (LSTM) directly into the simulation time-step, the system achieved proactive flood management. Crucially, the model demonstrated that hybrid architectures can handle "Black Swan" events safely, relying on physical constraints when AI predictions failed.

- **Surrogate Optimization:** In the Supply Chain case study, OpenCLD functioned as a high-throughput "Data Factory," executing over 640,000 parallel simulations to map the Bullwhip Effect landscape. This data enabled the training of a Random Forest surrogate model, reducing optimization time from days to milliseconds. The resulting "Balanced AI" policy achieved a substantial reduction (approximately 33%) in the Bullwhip Effect compared to the literature baseline, without compromising inventory stability.

- **Policy Learning:** In the Smart Microgrid case study, the library served as a Gymnasium-compliant environment for Deep Reinforcement Learning. Grounded in real-world, high-fidelity data (e.g., PVGIS solar irradiance, GREEND load profiles, and dynamic market prices), a Proximal Policy Optimization (PPO) agent successfully learned complex, non-linear economic behaviors, such as arbitrage and self-consumption, demonstrating that OpenCLD can support the rigorous training loops required for autonomous agent discovery.

Collectively, these results satisfy the specific research objectives outlined in Chapter 1: (i) demonstrating the feasibility of a Python-native SD architecture, (ii) enabling seamless, bidirectional data flow between the simulation engine and the broader Python ecosystem without external glue code, and (iii) empirically validating the approach across control, optimization, and learning tasks.

5.2 Implications: The Hard Convergence

This research argues that the field of simulation modeling is undergoing a critical transition from "Soft Convergence" to "Hard Convergence" [15].

In the taxonomy introduced in Chapter 2, "Hard Convergence" corresponds to tightly coupled, embedded hybrid architectures, where ML components operate within the simulation loop rather than as pre- or post-processing tools. The implication of this convergence is the creation of "Gray Box" systems that leverage the complementary strengths of both paradigms. Pure "White Box" models (traditional SD) offer interpretability and adherence to physical laws but struggle with high-dimensional pattern recognition. Pure "Black Box" models (Deep Learning) offer superior predictive power but lack causal reasoning and safety guarantees.

The Zola Reservoir experiment provides the strongest evidence for the necessity of this hybrid approach. When the AI component was blinded by an out-of-distribution "Black Swan" event, the system did not fail catastrophically. Instead, the "White Box" physical constraints (conservation of mass, maximum valve capacity) provided a safety layer that a pure end-to-end neural network would lack. This implies that for infrastructure, from dams to power grids, hybrid architectures emerge not merely as an optimization, but as a necessary design pattern for safety-critical systems. OpenCLD proves that such architectures are technically feasible within a standard open-source workflow.

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 capable of inference two to three orders of magnitude faster than the original simulator (10^2–$10^3\times$), 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.

A legitimate counterargument to this hybrid philosophy must be acknowledged: if end-to-end deep learning architectures, such as Neural Ordinary Differential Equations, are capable of learning system dynamics directly from data, why retain the explicit SD structure at all? The evidence from this thesis suggests three concrete reasons. First, *safety*: the Zola Reservoir Black Swan experiment demonstrated that when the ML component encountered an out-of-distribution event, it was the explicit physical constraints of the SD model (conservation of mass, valve capacity limits) that prevented catastrophic failure, a guarantee that a purely learned model cannot provide. Second, *data efficiency*: the Supply Chain surrogate was effective precisely because the SD model generated physically valid training data across 640,000 scenarios; a Neural ODE trained on the same problem would require comparable volumes of real-world observations that rarely exist for socio-economic systems. Third, *interpretability*: the SD structure exposes every feedback loop and causal link to human inspection, enabling stakeholders to understand, trust, and challenge model assumptions, a requirement in policy-relevant domains where accountability matters. End-to-end learning may eventually match these properties, but in the current state of the art, the explicit causal scaffold of System Dynamics remains indispensable for safety-critical, data-scarce, and decision-transparent applications.

5.3 Cross-Case Synthesis: What the Three Studies Reveal Together

Considered individually, each case study validates a specific hybrid architecture. Considered together, they reveal three broader design principles that neither case alone could establish.

5.3.1 Principle 1: Physical Constraints Are a Safety Layer, Not a Limitation

A recurring theme across all three studies is the role of the System Dynamics structure as a hard constraint on AI behavior. In the Zola Reservoir, when the LSTM failed completely on the Black Swan event, the physical conservation laws encoded in the SD modelprevented catas-

trophic overflow. In the Supply Chain study, the non-negativity constraints on inventory flows prevented the surrogate optimizer from proposing physically impossible policies, something a purely data-driven optimizer acting on an unconstrained parameter space would not guarantee. In the microgrid, the physical clamping of charge and discharge rates meant the PPO agent could never learn a policy that violated battery capacity limits, even during early random exploration.

This convergence suggests a general principle: in hybrid SD-ML architectures, the SD structure should be deliberately designed to encode inviolable physical and logical constraints, while the ML component is free to optimize within that feasible space. The SD layer is not merely a simulator, it is a constraint satisfaction engine that makes the overall system safer than either component alone. This has direct implications for safety-critical deployment: a pure neural network controller for a reservoir or a battery requires extensive adversarial testing to ensure it never proposes physically dangerous actions. A hybrid architecture with a well-formulated SD layer provides this guarantee structurally.

5.3.2 Principle 2: The Appropriate Integration Architecture Depends on the Time Horizon of the Decision

The three case studies span three fundamentally different decision timescales, and the appropriate hybrid architecture differs in each case. The reservoir requires online control at daily resolution, where the ML model must intervene within the simulation time-step, necessitating the Loop-in-the-Loop architecture. The supply chain requires offline policy design, where optimization happens before deployment and the surrogate must compress weeks of simulation time into milliseconds to make the search tractable. The microgrid operates on an episodic timescale, where a reinforcement learning agent must accumulate experience across thousands of simulated weeks to learn a general policy rather than a single optimal parameter set.

This progression from online to offline to episodic is not merely a technical distinction, it implies fundamentally different requirements on the SD engine. Online control requires low-latency single stepping. Offline optimization requires high-throughput batch execution and Python multiprocessing compatibility. Episodic learning requires a Gymnasium-compliant reset and step interface. That a single library architecture was able to support all three without modification to its core engine provides empirical evidence that the Model-as-Code paradigm is not optimized for one mode of hybrid integration but is genuinely general-purpose.

5.3.3 Principle 3: AI Failure Modes Are Predictable and Can Be Designed Around

Each case study also exposed a characteristic failure mode of the AI component, and in each case the failure was predictable from first principles. The LSTM failed on out-of-distribution inputs, as any supervised learning model will when the test distribution diverges from training. The Random Forest surrogate degraded in accuracy at the edges of the parameter space, as expected from a non-parametric model with sparse training data in high-variance regions. The PPO agent initially over-cycled the battery because the reward function did not account for round-trip efficiency losses, a known failure mode of naive reward shaping in continuous control.

What is significant is that in every case, the SD structure provided either a detection mechanism (the rising reservoir volume signaled that the LSTM forecast had failed) or a recovery mechanism (physical clamping contained the battery agent's exploration). This suggests that hybrid system designers should treat AI failure not as an edge case to be minimized, but as a design input: for any given ML component, its characteristic failure modes can be anticipated, and the surrounding SD structure can be explicitly designed to detect and compensate for them. This represents a more mature engineering philosophy than the common approach of simply improving ML accuracy until failure seems unlikely.

Taken together, these three principles constitute a nascent design grammar for hybrid SD-ML systems: encode physical constraints in the SD layer, match the integration architecture to the decision timescale, and design the SD structure to handle known AI failure modes gracefully. Future work should test whether these principles generalize beyond the domains studied here.

5.4 Limitations

While the cross-case synthesis above identifies general design principles, the current implementation of OpenCLD carries 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 synthetic hydrological data, and Case Study 2 uti-

lizes 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.

5.5 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.

5.6 Final Remarks

For decades, System Dynamics has been constrained by a reliance on proprietary, siloed software that isolated it from the broader exponential growth of the Data Science community. This thesis demonstrates that this isolation is not an inherent property of the methodology, but a consequence of outdated tooling.

By refactoring System Dynamics into open, extensible, and Python-native code, OpenCLD bridges the divide. It enables a new generation of researchers to build systems that are interpretable by design, optimized by data, and robust by nature. OpenCLD demonstrates that when System Dynamics is treated as executable theory rather than static diagrams, hybrid modeling becomes not an exception, but a natural extension of the methodology.

References

[1] John D. Sterman. *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Boston, MA: Irwin/McGraw-Hill, 2000. ISBN: 978-0072389159.

[2] Hazhir Rahmandad & John Sterman. "Quantifying the COVID-19 endgame: Is a new normal within reach?" *System Dynamics Review*, 38. 10.1002/sdr.1715.

[3] Justin Hoffmann et al. "A Systematic Literature Review on System Dynamics". *IEEE Access*, 13, 88871–88887. 10.1109/ACCESS.2025.3571620.

[4] Charles R. Harris et al. "Array programming with NumPy". *Nature*, 585:7825, 357–362. ISSN: 1476-4687. 10.1038/s41586-020-2649-2.

[5] Wes McKinney. "Data Structures for Statistical Computing in Python". Jan. 2010, 56–61. 10.25080/Majora-92bf1922-00a.

[6] Peter Jul-Rasmussen. "Hybrid Modeling for Process Systems Simulation: Integrating Process Understanding and Machine Learning". English. PhD thesis. Technical University of Denmark, 2025.

[7] Riccardo D'Elia. *Interpretable Neural System Dynamics: Combining Deep Learning with System Dynamics Modeling to Support Critical Applications*. 2025. arXiv: 2505.14428 [cs.LG].

[8] George Karniadakis et al. "Physics-informed machine learning". *Nature Reviews Physics*, 1–19. 10.1038/s42254-021-00314-5.

[9] Rick Stevens & et al. *AI for Science, Energy, and Security*. Tech. rep. Argonne National Laboratory, 2023.

[10] Emmanuel Klu et al. *SDGym: Low-Code Reinforcement Learning Environments using System Dynamics Models*. 2024. arXiv: 2310.12494 [cs.LG].

[11] Marek Zanker, Vladimír Bureš & Petr Tucnik. “Environment, Business, and Health Care Prevail: A Comprehensive, Systematic Review of System Dynamics Application Domains”. *Systems*, 9, 28. 10.3390/systems9020028.

[12] Vladimír Bureš. “Comparative Analysis of System Dynamics Software Packages”. *International Review on Modelling and Simulations (IREMOS)*, 8, 245. 10.15866/iremos.v8i2.5401.

[13] Eneko Martin-Martinez et al. “PySD: System Dynamics Modeling in Python”. *Journal of Open Source Software*, 7, 4329. 10.21105/joss.04329.

[14] Transentis Labs. *BPTK: Business Prototyping Toolkit*. `https://bptk.transentis.com/`. 2025.

[15] Stefano Armenia et al. “Zooming in and out the landscape: Artificial intelligence and system dynamics in business and management”. *Technological Forecasting and Social Change*, 200, 123131. ISSN: 0040-1625. https://doi.org/10.1016/j.techfore.2023.123131.

[16] Gerd Wagner & Frederic Mckenzie. “Process modeling for simulation: Observations and open issues”. Dec. 2016, 1072–1083. 10.1109/WSC.2016.7822166.

[17] K. Soetaert, Thomas Petzoldt & Rhyne Setzer. “Solving Differential Equations inR: PackagedeSolve”. *Journal of Statistical Software*, 33. 10.18637/jss.v033.i09.

[18] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. *Nature Methods*, 17, 1–12. 10.1038/s41592-019-0686-2.

[19] Mustafa Nesanır et al. “A Literature Review on the Integration of System Dynamics and Machine Learning/Artificial Intelligence in Addressing Environmental Matters”. Aug. 2025, 370–384. ISBN: 978-3-032-03514-1. 10.1007/978-3-032-03515-8_26.

[20] M. Amir Rahmani & Mahdi 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. ISSN: 1726-2135. 10.3808/jei.201400285.

[21] Zusheng Zhang et al. “Simulation and Analysis of the Complex Dynamic Behavior of Supply Chain Inventory System from Different Decision Perspectives”. *Complexity*, 2020. ISSN: 1076-2787. 10.1155/2020/7393848.

[22] James Doyle & et al. “System Dynamics and Endogenous Mental Models”. *Proceedings of the 31st International Conference of the System Dynamics Society*. 2013.

[23] System Dynamics Society. *What is System Dynamics?* `https://systemdynamics.org/what-is-system-dynamics-old/`. 2025.

[24] Ignacio Martinez-Moyano & George Richardson. "Best practices in system dynamics modeling". *System Dynamics Review*, 29. 10.1002/sdr.1495.

[25] Yaman Barlas. "Model validation in system dynamics". Jan. 1994, 1–10.

[26] Robert Sargent. "Verification and validation of simulation models". Vol. 37. Jan. 2011, 166–183. 10.1109/WSC.2010.5679166.

[27] Osman Balci. "Verification, validation, and accreditation". *Proceedings of the 30th Conference on Winter Simulation*. WSC '98. Washington, D.C., USA: IEEE Computer Society Press, 1998, 41–4. ISBN: 0780351347.

[28] Yaman Barlas. "Formal aspects of model validity and validation in system dynamics". *System Dynamics Review*, 12:3, 183–210. https://doi.org/10.1002/(SICI)1099-1727(199623)12:3<183::AID-SDR103>3.0.CO;2-4.

[29] Jay W. Forrester. *Industrial Dynamics*. Cambridge, MA: The MIT Press, 1961. ISBN: 978-0262060035.

[30] Jay W. Forrester. *Urban Dynamics*. Cambridge, MA: The MIT Press, 1969. ISBN: 978-0262060264.

[31] Donella H. Meadows et al. *The Limits to Growth: A Report for the Club of Rome's Project on the Predicament of Mankind*. New York: Universe Books, 1972. ISBN: 978-0876631652.

[32] Barry Richmond. "STELLA: Software for Bringing System Dynamics to the Other 98%". *Proceedings of the 1985 International Conference of the System Dynamics Society*. Keystone, CO: System Dynamics Society, 1985, 706–718.

[33] John D. Sterman. "Modeling managerial behavior: Misperceptions of feedback in a dynamic decision making experiment". *Management Science*, 35:3, 321–339. 10.1287/mnsc.35.3.321.

[34] Jack Homer & Gary Hirsch. "System dynamics modeling for public health: background and opportunities". *American Journal of Public Health*, 96, 452–458. 10.2105/AJPH.2005.

[35] John Sterman et al. "Climate interactive: the C-ROADS climate policy model". *System Dynamics Review*, 28. 10.1002/sdr.1474.

[36] John D.W. Morecroft. “System dynamics and microworlds for policymakers”. *European Journal of Operational Research*, 35:3, 301–320. ISSN: 0377-2217. https://doi.org/10.1016/0377-2217(88)90221-4.

[37] Rod Walker & Wayne Wakeland. “Calibration of Complex System Dynamics Models: A Practitioner’s Report”. *Proceedings of the 29th International Conference of the System Dynamics Society*. Washington, DC, 2011.

[38] Amin Ghadami & Bogdan Epureanu. “Data-driven prediction in dynamical systems: recent developments”. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 380. 10.1098/rsta.2021.0213.

[39] Riccardo D’Elia, Alberto Termine & Francesco Flammini. “Towards explainable decision support using hybrid neural models for logistic terminal automation”. 10.48550/arXiv.2509.07577.

[40] Dongheon Lee, Arul Jayaraman & Joseph Kwon. “Development of a hybrid model for a partially known intracellular signaling pathway through correction term estimation and neural network modeling”. *PLOS Computational Biology*, 16, e1008472. 10.1371/journal.pcbi.1008472.

[41] Rose Yu & Rui Wang. “Learning dynamical systems from data: An introduction to physics-guided deep learning”. *Proceedings of the National Academy of Sciences of the United States of America*, 121, e2311808121. 10.1073/pnas.2311808121.

[42] Dimitris Kastoris, Kostas Giotopoulos & Dimitris Papadopoulos. “Neural Network-Based Parameter Estimation in Dynamical Systems”. *Information*, 15, 809. 10.3390/info15120809.

[43] Hazhir Rahmandad, Ali Akhavan & Mohammad S. Jalali. “Incorporating Deep Learning Into System Dynamics: Amortized Bayesian Inference for Scalable Likelihood-Free Parameter Estimation”. *System Dynamics Review*, 41. 10.1002/sdr.1798.

[44] Víctor Madrigal et al. “Dynamical System Modeling for Disruption in Supply Chain and Its Detection Using a Data-Driven Deep Learning-Based Architecture”. *Logistics*, 9, 51. 10.3390/logistics9020051.

[45] Mark Alber et al. “Integrating Machine Learning and Multiscale Modeling: Perspectives, Challenges, and Opportunities in the Biological, Biomedical, and Behavioral Sciences”. 10.48550/arXiv.1910.01258.

[46] Yue Shi et al. “Deep Learning Meets Process-Based Models: A Hybrid Approach to Agricultural Challenges”. 10.48550/arXiv.2504.16141.

[47] Maede Maftouni, Navid Ghaffarzadegan & Zhenyu Kong. “A Deep Learning Technique for Parameter Estimation in System Dynamics Models”. *SSRN Electronic Journal*. 10.2139/ssrn.4440489.

[48] Akhil Ahmed, Ehecatl del Rio-Chanona & Mehmet Mercangöz. “Comparative Study of Machine Learning and System Identification for Process Systems Engineering Dynamics”. *Industrial Engineering Chemistry Research*, 64. 10.1021/acs.iecr.4c03264.

[49] Colby Fronk & Linda Petzold. “Interpretable polynomial neural ordinary differential equations”. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 33:4. ISSN: 1089-7682. 10.1063/5.0130803.

[50] Fabian Bussieweke, Josefa Mula & Francisco Campuzano-Bolarin. “Optimisation of recovery policies in the era of supply chain disruptions: a system dynamics and reinforcement learning approach”. *International Journal of Production Research*, 63:5, 1649–1673. 10.1080/00207543.2024.2383293.

[51] Ventana Systems. *Vensim Software*. `https://vensim.com/`. 2025.

[52] isee systems. *Stella Software*. `https://www.iseesystems.com/`. 2025.

[53] The AnyLogic Company. *AnyLogic Simulation Software*. `https://www.anylogic.com/`. 2025.

[54] Wikipedia contributors. *Comparison of system dynamics software*. `https://en.wikipedia.org/wiki/Comparison_of_system_dynamics_software`. 2025.

[55] Mark Wilkinson et al. “The FAIR Guiding Principles for scientific data management and stewardship”. *Scientific Data*, 3. 10.1038/sdata.2016.18.

[56] Andrea Monacchi et al. “GREEND: An Energy Consumption Dataset of Households in Italy and Austria”. *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. 2014.