Code-First System Dynamics: An LLM-Enabled Workflow for Validatable and Structurally Recoverable Modeling

Authors: Alton Alexander, Veydra

Abstract

Quantitative modeling in system dynamics traditionally follows a workflow where problem statements are manually translated into causal loop and stock–flow diagrams before being implemented as executable models. This diagram-first approach reinforces the foundational principle that structure determines behavior. While this has fostered the widespread use of specialized graphical modeling software, the manual construction of these diagrams often creates a bottleneck between conceptualization and simulation. Recent advances in large language models (LLMs) now offer a way to rethink this workflow, allowing for the direct translation of natural language into formal code while preserving the discipline's core structural outcomes.

This paper introduces a **code-first methodology** for system dynamics in which problem descriptions are translated directly into executable Python code through LLM-assisted generation. In this workflow, the code itself becomes the primary formal representation of the system's underlying structure, while diagrams can be derived automatically from the executable code.

To support this approach, we introduce a lightweight **Validatable Model Standard (VMS)** that constrains generated models to remain machine-parsable, auditable, and structurally recoverable. We compare traditional diagram-first workflows, intermediary AI approaches, and the proposed code-first paradigm, suggesting that code-first modeling can accelerate model development, improve reproducibility, and integrate system dynamics with modern computational ecosystems.

Keywords:

Code-First Modeling, Python, Model Transparency, Reproducible Simulation, AI-assisted model development, Model Standard

Code-First System Dynamics: An LLM-Enabled Workflow for Validatable and Structurally Recoverable Modeling

Authors: Alton Alexander, Veydra

Introduction

System dynamics (SD) modeling has long been grounded in the principle that **the internal structure of the system determines behavior**. The system dynamics approach to analyzing complex decisions, articulated by Forrester (1961) and later codified by Sterman (2000), emphasizes identifying feedback structures that generate observed system dynamics. In practice, this process typically involves translating a problem statement into causal loop diagrams and stock–flow diagrams that represent accumulations, flows, and feedback relationships.

Traditional quantitative modeling workflows therefore follow a sequence:

Problem → Manual Structure → Code → Behavior

In this process, modelers first articulate the dynamic problem and then use software tools to help construct visual diagrams representing system structure. When implemented in drag-and-drop computational platforms such as Vensim, Stella, or AnyLogic, these diagrammatic structures are linked to user-specified equations that combined are converted into simulation code and executed at run time.

While this diagram-centric workflow remains intuitive and widely adopted, it introduces several practical limitations. Model construction can be time-consuming for novice users, and implementation often requires manual translation between conceptual diagrams and their executable representations. In addition, many modeling environments remain tightly coupled to proprietary graphical tools, which can make integration with modern data-science ecosystems difficult. Despite these limitations, drag-and-drop interfaces remain an accessible and intuitive gateway to the robust computational capabilities of system dynamics, enabling users to simulate the behavioral evolution of complex systems over time.

Recent advances in **large language models (LLMs)** create new possibilities for restructuring this workflow in order to arrive at the same ability to simulate the behavioral evolution of complex systems over time. LLMs are increasingly capable of generating syntactically correct and semantically coherent code directly from natural language descriptions. This capability suggests an alternative modeling sequence:

Problem → LLM-Generated Code → Structure → Behavior

In this paradigm, executable code becomes the primary formal representation of system structure, while diagrams are derived automatically from the computational representation rather than constructed manually.

In this paper, we formalize this **code-first methodology for system dynamics modeling** and introduce a lightweight Python modeling framework designed to support it. We propose a **Validatable Model Standard (VMS)** that constrains generated models to remain structurally interpretable, auditable, and reproducible, while preserving a tight coupling between the feedback structure of the system and its executable representation.

Specifically, this paper makes five contributions:

1. **Methodological Paradigm:** We propose an LLM-enabled, code-first workflow that replaces the traditional manual-structuring phase with executable code as the primary formal modeling artifact.

2. **The Validatable Model Standard (VMS):** We introduce a Python-based framework that constrains LLM output to ensure generated models remain structurally interpretable, mathematically consistent, and auditable.

3. **Structural Recoverability:** We demonstrate, via canonical SD models, that complex feedback structures can be programmatically recovered from code, preserving the "glass-box" transparency essential for policy analysis.

Background and Theoretical Foundations

Improving the Modeling Workflow Efficiency

The traditional system dynamics methodology follows an ordered sequence: defining the dynamic problem, mapping feedback structure, formulating equations, and testing policy interventions. This process emphasizes causal clarity and is often summarized by the principle that “the internal structure of the system determines behavior.” In practice, however, it requires quantitative modelers to translate conceptual feedback structures into executable mathematical representations. Although graphical modeling tools were introduced to improve modeling workflow efficiency, translating conceptual feedback structures into executable implementations often remains the most time-consuming stage of the modeling process, motivating ongoing efforts to streamline the modeling workflow.

Recognizing the time and effort required for this translation, several researchers have sought to clarify and improve the modeling workflow. Ford (2010) provided a structured description of the

system dynamics model-building process, outlining stages from conceptualization through model formulation, testing, and policy analysis. Other efforts have focused on improving transparency and reproducibility in simulation-based research. Rahmandad and Sterman (2012) proposed reporting guidelines aimed at improving the documentation and communication of simulation models. Martinez-Moyano (2012) introduced SDM-Doc, a framework for standardized model documentation. Later work by Warren (2018) on Agile System Dynamics (AgileSD) extended these ideas by proposing iterative development cycles inspired by software engineering sprints. AgileSD emphasizes collaborative model evolution, rapid prototyping, and continuous validation—principles that align closely with contemporary efforts to further automate and accelerate the computational modeling process.

Using Technological Advances to Improve the Quantitative Modeling Workflow

The medium through which quantitative system dynamics models are expressed has evolved alongside advances in computing. In the early years of the field, models were implemented directly in simulation languages such as DYNAMO. In this environment, the executable code itself constituted the primary structural artifact of the model, requiring modelers to manually encode stocks, flows, and equations in textual form before simulation could be executed on mainframe digital computers.

With the spread of personal computing, system dynamics shifted toward graphical modeling environments built around drag-and-drop interfaces characteristic of WYSIWYG software applications. These tools established causal loop diagrams (CLDs) and stock-and-flow diagrams as the dominant representations of system structure. Graphical modeling environments enabled practitioners to visually construct models while automatically executing the corresponding mathematical equations. To facilitate interoperability among proprietary tools, the community later introduced standards such as XMILE to represent model structures in a portable format (Eberlein & Chichakly, 2013).

Modern computing environments have also dramatically expanded the computational capacity available for simulation. High-performance processors, cloud computing, and distributed systems now make it possible to execute large-scale numerical simulations for scientific computing. As a result, the primary constraint in many modeling workflows is no longer computational execution but the time and effort required for modelers to translate conceptual system structures into executable representations. This shift has encouraged renewed interest in modeling approaches that represent system structure directly in executable code.

The Emergence of Code-Native Modeling Environments

Alongside graphical modeling environments, the broader scientific computing community has increasingly adopted programming-based workflows for modeling and simulation. In particular, the Python ecosystem has become widely used for scientific computing, providing libraries for

numerical integration, optimization, data analysis, and machine learning (Virtanen et al., 2020). These tools enable models to be implemented directly as executable code while integrating with modern analytical workflows.

Within system dynamics, several projects have explored implementing models directly in general-purpose programming languages. For example, Duggan (2016) demonstrates how system dynamics models can be implemented in the R programming language, while projects such as PySD translate models from traditional system dynamics software into Python modules that can be executed within the broader scientific computing ecosystem (Houghton & Siegel, 2015). More broadly, Python has also become a common platform for other simulation paradigms, including agent-based modeling frameworks such as Mesa (Kazil et al., 2020).

Representing models as code also aligns with modern software development practices, allowing models to be version controlled, tested, and integrated into computational pipelines. In this sense, contemporary code-native modeling environments represent a partial return to the field's early computational roots in languages such as DYNAMO, while extending those ideas into modern scientific computing ecosystems and, increasingly, AI-assisted model generation.

The First Wave of AI Integration: Assistants and Intermediaries

The emergence of generative AI has introduced new possibilities for accelerating the modeling process. Early efforts to integrate large language models (LLMs) into system dynamics have largely focused on using AI as an assistant within existing modeling environments.

The System Dynamics Society's "AI in SD Education" initiative highlights the potential role of LLMs as conversational modeling assistants (System Dynamics Society, 2026). Similarly, several software platforms have introduced chat-based interfaces that assist users in constructing models. For example, analytics platforms have introduced conversational AI assistants such as Copilot in Microsoft Power BI and Einstein GPT in Tableau, enabling users to generate analyses and visualizations directly through natural-language interaction (Microsoft, 2023; Salesforce, 2023). Similar trends are emerging in simulation environments, where platforms are beginning to expose APIs and libraries that allow conversational agents to interact with modeling software.

Within the system dynamics community, this intermediary approach is exemplified by the open-source sd-ai project developed through the Benchmarking and Evaluating AI for Modeling and Simulation (BEAMS) initiative at the University at Buffalo (BEAMS, 2025; sd-ai, 2026), along with related tools such as CoModel and the AI assistant in Stella that support LLM-mediated model construction. In these systems, the LLM parses a natural-language problem description and generates a **structured intermediary representation** of the model in a machine-readable format such as JSON. This representation typically specifies variables, equations, arrays, relationships, and modules in a standardized schema designed to be consumed by an external modeling environment.

The resulting configuration is then transmitted through an API to a graphical modeling environment—such as Stella—which constructs the corresponding stock-and-flow representation and executes the simulation interactively (isee systems, 2025). These systems can assist with drafting, validating, or modifying models, but the structured configuration primarily serves as an **intermediate translation layer** between the LLM and the simulation environment. The executable model ultimately resides within the graphical tool rather than in the generated representation itself.

Consequently, iterative refinement of the model often requires repeated translation between the structured configuration and the graphical representation. In this architecture, the machine-readable schema functions as a bridge between LLM-generated output and the execution environment rather than serving as the primary modeling artifact. Figure 1 (top) illustrates this intermediary AI workflow.

The AI-Code-First Methodology

Code-First Modeling in Scientific Computing

This approach parallels “code-first” movements in other areas of software engineering, such as Model-Driven Engineering (MDE), which explore deriving models from executable artifacts rather than generating code from models (Boronat, 2019). In the paradigm proposed here, the executable model becomes the primary formal representation of the system and its complete structure.

Representing models as executable code provides a transparent and auditable record of feedback structure that can be stored in version control systems, validated automatically, and executed within modern computational environments. Visual stock-and-flow diagrams are then generated automatically from the executable representation as derived visualizations rather than manually constructed artifacts.

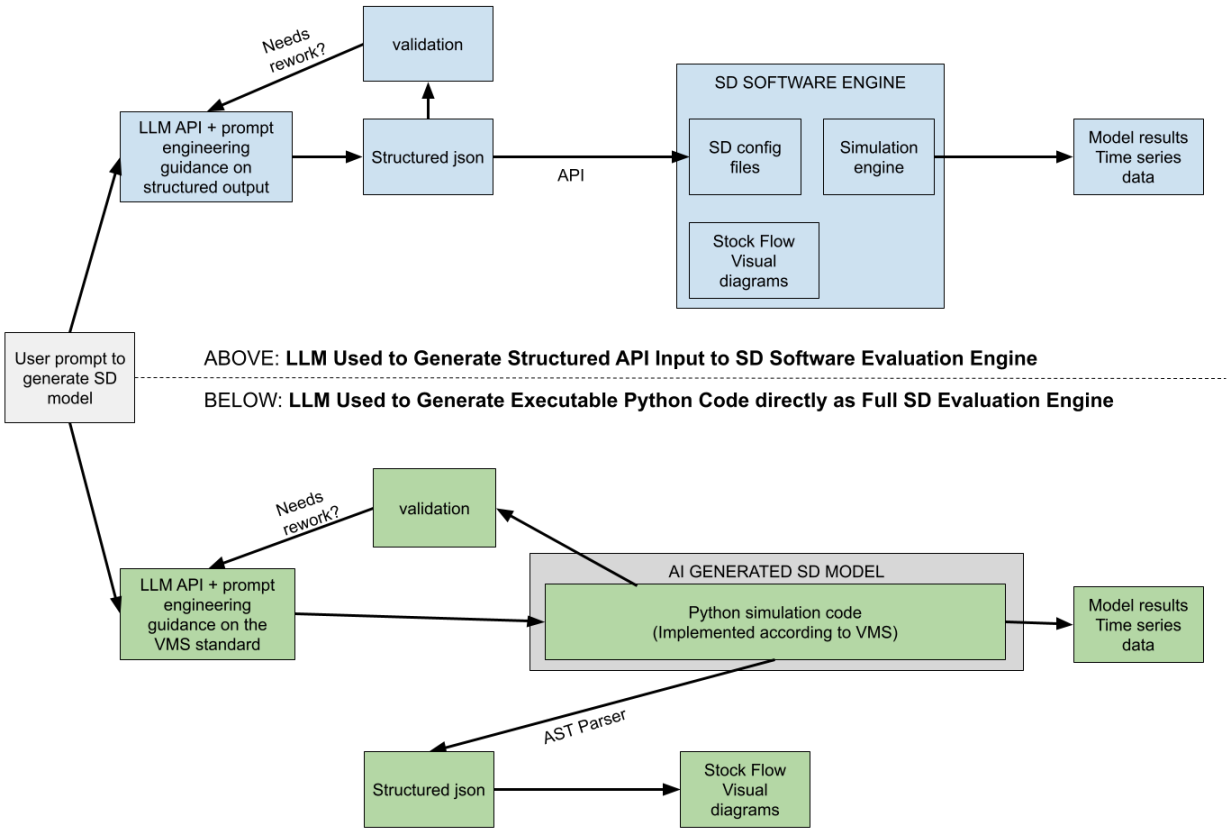

Figure 1. Comparison of AI-assisted system dynamics model-generation workflows.
Top: Intermediary AI workflow in which an LLM produces a structured representation (e.g., JSON) that is transmitted to a graphical system dynamics engine to construct diagrams and execute the model.
Bottom: Proposed AI code-first workflow in which the LLM generates executable Python simulation code conforming to the Validatable Model Standard (VMS), enabling direct execution while deriving diagrams and visual representations from the code.

AI-Assisted Code Generation

Building on the emergence of code-native modeling environments, we introduce a native AI code-first paradigm for system dynamics modeling. These capabilities provide an important foundation for AI-assisted model generation, where executable models can be generated as code and refined directly from natural-language problem descriptions using large language models (Chen et al., 2021). Rather than generating intermediate representations that must be translated into graphical tools, this approach allows large language models (LLMs) to generate executable model code directly.

In this workflow, modelers iteratively refine natural-language descriptions of the dynamic problem. The LLM interprets these descriptions and produces executable Python code that conforms to a standardized modeling grammar. The resulting code represents the structural relationships of the model directly, including stocks, flows, parameters, and auxiliary variables.

Challenges of Code-First System Dynamics

For this paradigm to remain consistent with established system dynamics practice, the derived diagrams must faithfully reproduce the structure encoded in the executable model while preserving the visual conventions familiar to practitioners. Achieving this consistency requires a modeling standard capable of preserving structural semantics within the code representation. For this reason, the methodology introduced in this paper relies on the Validatable Model Standard (VMS) to ensure that generated models remain structurally interpretable and that visualizations derived from code accurately reflect the underlying model structure.

The Validatable Model Standard (VMS)

To address this challenge, the methodology introduced in this paper relies on the Validatable Model Standard (VMS) to ensure that generated models remain structurally interpretable and that visualizations derived from code accurately reflect the underlying model structure.

The AI-Code-First methodology extends the agile philosophy of rapid iteration and transparent documentation into a code-first modeling paradigm. Instead of constructing diagrams and subsequently translating them into equations, modelers collaborate with a large language model (LLM) to generate executable code directly from the problem statement. The workflow unfolds through several interlinked stages.

Iterative Problem Definition

The process begins with conversational problem framing. The modeler uses an LLM to iteratively refine the dynamic hypothesis using natural language. Through successive prompts, the LLM identifies candidate feedback mechanisms, boundary assumptions, and expected reference modes. This mirrors the short iterative development cycles proposed in Agile System Dynamics approaches (Warren, 2018), but replaces diagramming sprints with **language sprints**—short cycles in which hypotheses are articulated, challenged, and refined through natural-language interaction.

Lightweight Modeling Grammar and The Validatable Model Standard (VMS) in Python

To maintain consistency across AI-generated models, the methodology employs a lightweight modeling grammar implemented through the Validatable Model Standard (VMS). VMS is realized as a Python framework that imposes minimal but explicit structure on generated code through standardized base classes, metadata conventions, and parameter-handling rules. In this sense, VMS functions as a lightweight modeling grammar for code-first system dynamics models, much like a coding standard in software engineering, providing a stable interface between human reasoning, LLM code generation, simulation execution, and automated parsing tools.

Figure 2 summarizes this architecture. A standardized input schema is first passed into a master orchestration layer, which resolves parameters, applies validation rules, and manages execution. The orchestrator then coordinates one or more VMS-compliant submodels implemented as Python subclasses. Each submodel encapsulates a specific portion of the system's logic while adhering to a common interface for variable declaration, parameter access, and derivative and flow calculation. This architecture allows multiple submodels to be instantiated and executed within a unified simulation pipeline while still producing standardized outputs suitable for downstream visualization, validation, and machine parsing.

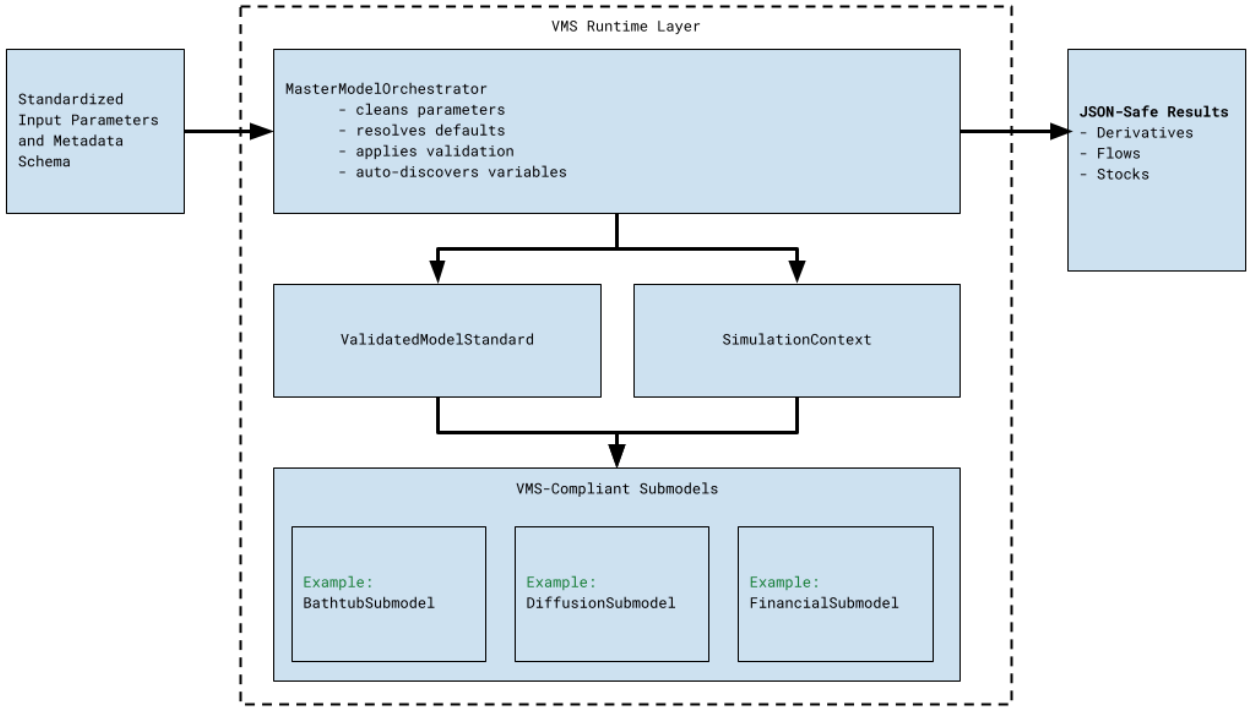

Figure 2. Conceptual architecture of the Validatable Model Standard (VMS). *Standardized input parameters and metadata are passed into a master orchestration layer that resolves*

defaults, validates values, and manages execution. The orchestrator coordinates one or more VMS-compliant Python submodels through shared base classes and a common simulation interface, producing standardized outputs that are suitable for simulation, visualization, and automated parsing.

At the implementation level, this architecture is realized through a small set of reusable base classes that define the modeling grammar while leaving domain logic to concrete subclasses. These base classes handle boilerplate functionality such as variable discovery, parameter validation, and dynamic execution. Concrete submodels generated by the LLM then focus primarily on expressing the structural logic of the system. A simple example of such a VMS-compliant submodel is shown in Code Snippet 1.

```python
class BathtubSubmodel(Submodel):

    def calculate_derivatives_and_flows(self, sim_context):
        water_level = sim_context.get_stock('bathtub.s_water_level', 0.0)
        tap_flow_rate = sim_context.get_param('bathtub.tap_flow_rate', 0.0)
        drain_flow_rate = sim_context.get_param('bathtub.drain_flow_rate', 0.0)
        drain_sensitivity =
sim_context.get_param('bathtub.drain_sensitivity_to_level', 0.0)

        # Calculate the actual drain flow, which depends on base rate and water
level
        # Ensure drain flow does not exceed water level, though in this simplified
model it can conceptually be negative if demand exceeds supply.
        # For a physical bathtub, drain flow is also limited by water_level, but
we keep it simple here.
        actual_drain_flow = drain_flow_rate + drain_sensitivity * water_level
        actual_drain_flow = max(0.0, actual_drain_flow) # Ensure drain flow is not
negative

        # Calculate derivatives
        # d(water_level)/dt = tap flow - drain flow
        d_water_level_dt = tap_flow_rate - actual_drain_flow

        # Calculate flows
        tap_dt = tap_flow_rate
        drain_dt = actual_drain_flow

        derivatives = [d_water_level_dt]
```

```
        flows = {
            'bathtub.tap_dt': tap_dt,
            'bathtub.drain_dt': drain_dt
        }

        return derivatives, flows
```

Code Snippet 1: *Example VMS-compliant submodel implementing the classic bathtub system dynamics structure. The complete implementation of this example is provided in* ***Appendix A****.*

Modular Orchestration and Decomposition of Complex Systems

In robust policy analysis, models are viewed not as monolithic entities, but as an assembly of conceptually compartmentalized sub-components. The Validatable Model Standard (VMS) therefore introduces a **MasterModelOrchestrator** pattern that aggregates multiple submodels (e.g., a `BathtubSubmodel`) into a unified simulation system. The orchestrator uses an `auto_discover_variables()` method to collect variable definitions and metadata from all included submodels, thereby creating a single managed parameter state for the model.

Modular decomposition, enforced by VMS, is not merely an architectural convenience but also a methodological discipline. For example, when modeling the adoption of a new product, the system can be decomposed into several interacting components, such as a `DiffusionSubmodel` implementing Bass diffusion dynamics, an `InventorySubmodel` representing supply and stock levels, and a `FinancialSubmodel` tracking revenues and costs. By structuring the model as distinct Python classes—each adhering to the VMS interface and interacting only through the **MasterModelOrchestrator**—model complexity becomes easier to manage. This structure improves readability, supports independent validation of components, and enables scalable policy simulations.

This modular organization is also consistent with long-standing practices in system dynamics modeling environments such as Stella and Vensim, where models are commonly organized into modules or diagram views that decompose complex systems into manageable conceptual components.

Modular decomposition also proved useful when working with earlier frontier LLMs, whose limited context windows and reliability constraints made it advantageous to generate and validate individual submodels separately. Although newer models with substantially larger context windows and improved reasoning capabilities can operate across multi-file repositories, the modular structure remains beneficial for interpretability, validation, and maintainability (Vaswani et al., 2017; Wei et al., 2022; OpenAI, 2023; Ding et al., 2023).

The submodel therefore remains a flexible organizational unit that allows the LLM to structure the implementation in a manner consistent with the conceptual mental model used by the analyst, while preserving flexibility in how the underlying computational representation is realized.

Validating that a Model Conforms to the Standard

Generated models can undergo automated validation to ensure that they conform to the structural requirements of the Validatable Model Standard (VMS). Structural checks verify that flows connect valid stocks and that feedback loops are properly defined. Dimensional tests ensure unit consistency across model equations, while behavioral tests evaluate baseline scenarios to confirm expected system dynamics behavior.

If validation tests fail, the AI-assisted generation process can issue corrective prompts to revise the model structure until the validation criteria are satisfied. The generation and validation workflow is illustrated in the lower portion of **Figure 1**.

Although the VMS framework is intentionally flexible, it enforces strict formatting requirements for key structural elements—including stocks, flows, parameters, and auxiliary variables. These conventions ensure that the executable model both produces the intended simulation behavior and remains structurally interpretable by automated tools. Model definitions therefore remain machine-readable and can be reliably parsed using an Abstract Syntax Tree (AST) representation. AST parsing analyzes the syntactic structure of the Python source code without executing it, allowing variables, function calls, and dependency relationships to be programmatically identified. Within the VMS framework, this enables automated extraction of stocks, flows, and parameter dependencies directly from the executable model, supporting structural validation and diagram reconstruction.

Execution, Flow Capturing, and Interpretation

Once a model passes validation, it can be executed directly using standard numerical solvers. In the current implementation, simulations are performed using `scipy.integrate.solve_ivp` with the RK45 integration method. Because the Validatable Model Standard (VMS) is intentionally lightweight, however, it does not prescribe a specific integration technique. Implementations may instead use alternative numerical solvers, including explicit Euler stepping in a time-loop or other numerical integration libraries, depending on the requirements of the modeling environment.

During execution, VMS separates state integration from flow recording. While the numerical solver integrates stock values over time, the orchestration layer simultaneously evaluates and records the associated flows at each time step. The variables to be captured are specified through a centralized `outputs.json` configuration.

Because models are implemented as executable Python code, simulation outputs can be returned through programmatic interfaces. Wrapper functions (e.g., `run_interactive_simulation()`) allow parameters to be modified and simulations executed without reinitializing the underlying model architecture. These functions return structured, JSON-compatible result objects containing synchronized time-series data for stocks, flows, and model parameters.

Visualization services can then analyze the dependency structure of the executable model to generate causal-loop and stock–flow diagrams automatically. Because these diagrams are derived directly from the executable representation, the graphical visualization remains consistent with the underlying mathematical model.

Illustrative Evaluation of Structural Recoverability

To assess the practical viability of the proposed methodology, we conducted an illustrative evaluation of structural recoverability across three canonical system dynamics models: the bathtub model, a SIR epidemiological model, and a Bass diffusion model. The objective of this evaluation is not to claim comprehensive benchmark performance, but to demonstrate that models generated in a code-first workflow can preserve the structural semantics required for system dynamics interpretation when they conform to the Validatable Model Standard (VMS).

In this paper, **structural recoverability** refers to the extent to which a model's essential system dynamics structure can be reconstructed from its executable Python representation. Recoverability was evaluated by determining whether stocks, flows, dependencies, and variable metadata could be programmatically identified and whether the recovered structure was sufficient to derive a stock–flow representation automatically. For each model, a natural-language problem description was used to generate VMS-compliant Python code. The resulting artifact was validated against the standard, and structural elements were extracted and compared with the intended canonical formulation.

Across all three cases, the VMS-compliant executable models preserved the structural elements required for automated reconstruction of the intended system dynamics representation. Stocks were recoverable because they were declared explicitly through the variable schema and referenced consistently in derivative calculations. Flows were recoverable because rate variables were named and recorded separately during execution. Dependencies among stocks, auxiliaries, and parameters were identifiable by parsing the expressions used in flow and derivative logic, including nested mathematical expressions or user-defined functions commonly used in system dynamics modeling environments. Metadata fields further supported machine-readable interpretation of variable roles, units, and descriptions.

Table 1. Illustrative evaluation of structural recoverability across canonical models.

Python Model	Stocks Recovered	Flows Recovered	Dependencies Recovered	Metadata Present	Diagram Derivable	Validation Outcome
Bathtub	Yes	Yes	Yes	Yes	Yes	Passed
SIR	Yes	Yes	Yes	Yes	Yes	Passed
Bass Diffusion	Yes	Yes	Yes	Yes	Yes	Passed

In each case, all structural elements present in the canonical model formulation were successfully recovered from the generated VMS-compliant code. These results provide initial evidence that a code-first workflow can preserve the formal structure of system dynamics models when the generated code follows a constrained, machine-readable standard.

However, structural recoverability should not be confused with correctness. A structurally recoverable model may still contain incorrect assumptions, weak boundary choices, or unrealistic parameterization. The contribution of VMS is therefore not to guarantee scientific validity, but to ensure that executable Python models remain inspectable, auditable, and structurally derivable from code. Scientific validity instead depends on the assumptions embedded in the generated model, which are shaped by both the capabilities of the language model and the contextual information supplied in the LLM prompt.

This evaluation is intentionally limited in scope. The three models examined here are canonical and compact, making them suitable for proof-of-concept demonstration but insufficient to establish general performance across larger policy models or more complex domains. Future work should extend this analysis to larger benchmark suites, alternative prompting strategies, and more formal comparisons between recovered diagrams and expert-authored model structures.

Comparison Overview of Model Generation Methodologies

As illustrated by the architectural evolution of simulation tools, there are several distinct ways in which artificial intelligence can be integrated into system dynamics modeling workflows. Table 2 summarizes key differences between traditional diagram-driven modeling approaches, intermediary AI-assisted workflows that rely on structured interchange formats, and the

proposed validated AI code-first methodology. Figure 1 further illustrates the model-generation workflows for intermediary AI systems and the proposed AI code-first approach.

***Table 2**. Comparison of system dynamics model-generation methodologies.*

Aspect	Traditional SD Methodology	API / Intermediary AI Workflows (e.g., SD-AI)	VMS AI Code-First Methodology
How AI is Used	No direct AI involvement in model generation.	AI generates structured model descriptions (e.g., SD-JSON).	AI generates executable Python code conforming to VMS.
Model Conceptualization	Modelers design causal-loop and stock–flow diagrams manually.	AI produces an intermediate structured representation of model elements.	VMS defines a modeling grammar for representing models directly in Python.
Model Construction	Equations implemented manually within modeling environments (e.g., Stella, Vensim).	Structured payload passed into simulation environments for assembly.	LLM generates runnable Python code following the VMS standard.
Primary Structural Artifact	Diagram and equations within graphical modeling tools.	Structured configuration plus GUI model representation.	Executable, machine-readable model code
Execution Environment	Built-in solvers within system dynamics modeling environments.	Simulation engines within modeling environments.	Python numerical ecosystem (e.g., NumPy, SciPy).
Structural Validation	Manual inspection + built-in model consistency checks	Structured payload checks + modeling-environment validation	Automated validation via VMS schema and AST parsing

Additional Benefits for Code-first Modeling

Beyond enabling AI-assisted model generation, the code-first representation introduced by VMS also provides several practical advantages for model development, experimentation, and integration with modern analytical workflows.

Version Control and Scenario Management

Representing a system dynamics model entirely as source code allows researchers to manage complexity using **Git** which is the industry-standard version control system for software development**.** Rather than relying on a linear series of "Save As" files, analysts can treat different policy assumptions as distinct, non-linear branches of a single codebase. In this workflow, a "scenario" is no longer just a set of changed constants; it can be an entirely different structural configuration allowing for direct, side-by-side execution and comparison of divergent system architectures.

This approach marks a significant departure from traditional modeling, where simulations are often distributed as monolithic, proprietary binaries or stored without a formal structural history. By adopting a code-first framework, the model inherits a transparent, auditable record of every modification to both its parameters and its underlying logic. This transition to code level version control fundamentally enhances reproducibility and enables multi-user collaboration, ensuring that the evolution of a policy design is systematically documented and easily reversible.

Simulation Context and Time-Varying Dynamics

A key challenge in code-based system dynamics models is managing time-varying signals and contextual state during simulation. VMS addresses this through a `SimulationContext` object that is passed to submodels during execution. This object maintains global state, simulation time, and cached values required for dynamic evaluation.

The context enables the evaluation of dynamic mathematical expressions (e.g., `sin(t * 0.1) * 50 + 100`) and interpolated lookup tables at each simulation step. As a result, traditional system dynamics variables can incorporate time-dependent functions or external signals without disrupting the numerical integration process.

Integration with Scientific Computing Ecosystems

Because VMS models are implemented in Python, model equations can directly leverage the broader scientific computing ecosystem, including mathematical functions from the standard library and numerical capabilities provided by libraries such as NumPy. This environment also enables integration with external data sources, APIs, and analytical libraries, allowing system dynamics models to interact with modern data pipelines, real-time data streams, and machine-learning workflows.

Moreover, representing system dynamics models as executable code allows them to interoperate with a wide range of Python-based analytical tools, including libraries for optimization, scheduling, and statistical analysis. This capability supports model calibration using empirical data, real-time parameter updates, and automated publication of simulation outputs within analytical pipelines.

Finally, the use of a general-purpose programming environment enables natural integration with other simulation paradigms, allowing hybrid modeling approaches that combine system dynamics with agent-based models or discrete-event simulations within a unified computational framework.

Reference Library

Establishing a curated library of validated reference models can help define best practices for a new modeling standard. Such a library may include classical system dynamics formulations such as the **Bass diffusion model**, **SIR epidemiological model**, and the **bathtub model**. While these examples are relatively simple, they provide well-understood structural patterns that can be used to validate the correctness of AI-generated models.

Representing these canonical models within the Validatable Model Standard provides reliable examples that can be included as contextual references when prompting large language models to generate new models. These reference implementations therefore serve both as validation benchmarks and as structured examples that guide the generation of more complex models. An example implementation of such a reference model is provided in **Appendix A**.

Discussion of Our Contributions

The core distinction of the AI code-first methodology is not merely the representation of models as code, but the shift from linguistic reasoning about system behavior toward computational reasoning through executable simulation. Rather than using large language models only to discuss dynamic hypotheses, the approach uses them to construct formal models that can be simulated and analyzed before their results are interpreted again in natural language.

This distinction reflects a long-standing insight in system dynamics. Reflecting on the development of the field, Jay Forrester emphasized that meaningful understanding of complex systems ultimately depends on explicit computational models rather than informal reasoning, noting that “only by going the full road to extensive computer simulations is one prepared for the depth of understanding required in real-world situations” (Forrester, 2007, p. 6).

The AI-generated code-first methodology operationalizes the rapid iteration principles associated with Agile System Dynamics while introducing a structured framework for documentation and validation. This approach offers several advantages:

- First, **speed and iteration** are improved. Modelers can progress from a narrative problem statement to an executable simulation within minutes, enabling rapid exploration of alternative hypotheses and policy scenarios.

- Second, **transparency and auditability** are enhanced because model structure is represented directly in source code, providing a version-controlled record of equations, assumptions, and structural changes.

- Third, **interoperability** is expanded: because models are implemented in Python, they can integrate directly with data pipelines, machine-learning libraries, and modern computational workflows.

Despite these advantages, the code-first paradigm introduces several challenges. Quality control remains a central concern. Although large language models can generate syntactically valid code, they may still introduce incorrect causal relationships, inconsistent feedback structures, or poorly justified parameter assumptions. Ensuring that generated models reflect meaningful causal reasoning rather than superficial pattern matching therefore remains an important area for future research, particularly as AI tools become integrated into the process of building and learning with dynamic models (Schoenberg, 2026). Robust validation frameworks, benchmark model libraries, and automated structural tests will be necessary to support reliable AI-assisted modeling.

A second challenge concerns **user cognition and modeling practice**. Traditional system dynamics workflows emphasize visual diagramming as a means of supporting conceptual reasoning about feedback structure. A code-first workflow risks weakening this visual intuition if diagrams are not immediately accessible. The Validatable Model Standard (VMS) addresses this concern by enforcing explicit structural metadata that allows causal-loop and stock–flow diagrams to be generated automatically from the executable model. In this way, graphical representations become synchronized views of the underlying model structure rather than manually constructed artifacts. This preserves the visual reasoning practices central to system dynamics while enabling a more reproducible, code-based modeling workflow.

Future work will focus on expanding automated validation techniques for AI-generated models, developing larger libraries of reference models expressed in the Validatable Model Standard, and exploring interactive environments in which human modelers and AI systems collaboratively construct and refine dynamic hypotheses. These efforts may help establish code-first modeling as a complementary methodology within the broader system dynamics toolkit and may also broaden access to quantitative system dynamics modeling for a wider community of analysts, educators, and decision-makers.

Future Directions: Agentic AI Workflows and Integration

The methodology presented in this paper demonstrates how system dynamics models can be generated and executed through an AI-assisted code-first workflow. By representing model structure directly in executable Python code, the approach improves transparency, reproducibility, and interoperability relative to traditional diagram-centric modeling environments.

Looking forward, advances in AI tooling may further expand the capabilities of such modeling environments. Recent developments in agentic AI systems suggest a shift from passive prompt-based interactions toward autonomous agents capable of planning, reasoning, and executing multi-step analytical workflows (Yao et al., 2023). In this context, large language models may evolve from tools that assist in model generation into agents that iteratively construct, test, and refine simulation models in collaboration with human analysts.

Emerging interoperability protocols such as the **Model Context Protocol (MCP)** aim to standardize how AI systems interact with external tools and data sources (Hou et al., 2026). Within a code-first modeling environment, such protocols could allow AI agents to retrieve data, execute simulations, and evaluate policy scenarios across multiple analytical systems as part of a continuous modeling workflow.

Future research may therefore focus on designing collaborative environments in which human modelers and AI systems jointly develop and test dynamic hypotheses. These developments may help extend system dynamics modeling toward a more transparent, reproducible, and computationally integrated practice in which executable model representations serve as the primary artifact for both analysis and communication.

In conclusion, the code-first paradigm does not replace the foundational principle of system dynamics—that structure determines behavior—but instead reintroduces code-based modeling within modern computational environments to represent, validate, and explore the dynamics of complex systems.

References

BEAMS. (2025). Benchmarking and Evaluating AI for Modeling and Simulation. Institute for Artificial Intelligence and Data Science, University at Buffalo. https://www.buffalo.edu/ai-data-science/research/beams.html

Boronat, A., "Code-First Model-Driven Engineering: On the Agile Adoption of MDE Tooling," 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 2019, pp. 874-886, doi: 10.1109/ASE.2019.00086.

Chen, M. et al., “Evaluating Large Language Models Trained on Code,” arXiv preprint arXiv:2107.03374, 2021. https://arxiv.org/abs/2107.03374

Ding, X., et al. (2023). LongNet: Scaling Transformers to 1,000,000,000 Tokens. arXiv preprint. https://arxiv.org/abs/2307.02486

Duggan, J. (2016). *System Dynamics Modeling with R*. Springer. https://doi.org/10.1007/978-3-319-34043-2

Eberlein, R. L., & Chichakly, K. J. (2013). XMILE: A new standard for system dynamics. System Dynamics Review, 29(3), 188–195.

Ford, A. (2010). *Modeling the environment* (2nd ed.). Island Press.

Forrester, J. W. (1961). Industrial Dynamics. MIT Press.

Forrester, J. W. (2007), System dynamics—the next fifty years. Syst. Dyn. Rev., 23: 359-370. https://doi.org/10.1002/sdr.381

Hou, X., et al. (2026). Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. ACM Transactions on Software Engineering and Methodology. Just Accepted (February 2026). https://doi.org/10.1145/3796519

Houghton, J., & Siegel, M. (2015). PySD: A Python library for system dynamics modeling. System Dynamics Society.

isee systems. (2025). Stella system dynamics modeling software and the AI Assistant. isee systems, inc. https://iseesystems.com/resources/help/v4/default.htm#08-Reference/02-AnalysisWindows/AIAssistant.htm

Kazil, J., Masad, D., & Crooks, A. (2020). Utilizing Python for agent-based modeling: The Mesa framework. In R. Thomson, H. Bisgin, C. Dancy, A. Hyder, & M. Hussain (Eds.), Social, Cultural, and Behavioral Modeling (pp. 308–317). Springer. https://doi.org/10.1007/978-3-030-61255-9_30

Martinez-Moyano, I. J. (2012). Documentation for model transparency. System Dynamics Review, 28(2), 199–208.

Microsoft. (2023). Copilot in Power BI. Microsoft Corporation. https://learn.microsoft.com/en-us/power-bi/create-reports/copilot-power-bi

OpenAI. (2023). GPT-4 Technical Report. arXiv preprint. https://arxiv.org/abs/2303.08774

Rahmandad, H., & Sterman, J. D. (2012). Reporting guidelines for simulation-based research in social sciences. System Dynamics Review, 28(4), 396–411.

Salesforce. (2023). Einstein GPT for Tableau: Generative AI for analytics. Salesforce Inc. https://www.salesforce.com/products/einstein/

Schoenberg, W. (2026). Building and learning with models using AI. System Dynamics Review, 42(1), e70019. https://doi.org/10.1002/sdr.70019

sd-ai. (2026). sd-ai: Natural language system dynamics modeling. GitHub repository. Retrieved March 3, 2026 from https://github.com/isee-systems/sd-ai

Sterman, J. D. (2000). *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Irwin McGraw-Hill.

System Dynamics Society. (2026). AI in SD Education. Retrieved February 27, 2026, from https://systemdynamics.org/ai-in-sd-education/

Vaswani, A., et al. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems (NeurIPS). https://arxiv.org/abs/1706.03762

Virtanen, P. et al., (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. Nature Methods, 17(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Warren, K. (2018). Fast and effective living business models with system dynamics: A tutorial on business cases. 2018 Winter Simulation Conference (WSC), Gothenburg, Sweden, 291–305. https://doi.org/10.1109/WSC.2018.8632228

Wei, J., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. Advances in Neural Information Processing Systems (NeurIPS). https://arxiv.org/abs/2201.11903

Yao, S., et al. (2023). ReAct: Synergizing reasoning and acting in language models. Proceedings of the International Conference on Learning Representations (ICLR). https://arxiv.org/abs/2210.03629

Appendix A: The Complete VMS Bathtub Model Example

This appendix provides the complete set of files illustrating the implementation of the classic Bathtub Model using the Validatable Model Standard (VMS). This simple, yet foundational, system dynamics structure serves as a canonical example for training and validating LLM-generated models. The VMS enforces modularity, explicit parameter resolution, and clear separation of concerns, ensuring the model is easily machine-parsable for automated visualization and validation.

The following files are included in this example:

- **`/src/validatable_model_standard.py`**: Defines the base classes and simulation context used by VMS-compliant models.
- **`/src/model.py`**: Implements the MasterModelOrchestrator for the Bathtub example and executes the simulation.
- **`/src/bathtub_submodel.py`**: Contains the VMS-compliant `BathtubSubmodel` class, defining the stock and flow logic, including the calculation of derivatives.
- **`/ast_dependency_extractor_demo.py`**: Illustrative AST-based dependency extractor with sample output.
- **`/config/outputs.json`**: The configuration file specifying which variables (stocks and flows) should be explicitly captured and returned in the final JSON-safe result payload.
- **`test_model.py`**: Sample standalone script
- Example output format and chart of data
- **`LICENSE`**: The MIT License text, ensuring open and permissive use of the VMS code as provided.

validatable_model_standard.py:

```python
"""Validatable Model Standard - Base classes for VMS-compliant models."""

import numpy as np
from scipy.integrate import solve_ivp
import json
from typing import Dict, Any, Optional, List, Tuple
from datetime import datetime

class Submodel:
    """
```

```
Base class for VMS-compliant submodels.

Handles all the boilerplate for variable management, leaving submodels
to focus only on their specific physics/logic implementation.

Usage:
    1. Define VARIABLES dict at module level with complete info
    2. Inherit from this class
    3. Implement calculate_derivatives_and_flows(self, sim_context) method
    4. Add your physics methods - everything else is handled automatically!

The base class provides:
    - get_variables() - auto-detects VARIABLES from module
    - get_defaults() - extracts default values
    - get_validation() - extracts min/max/step/type rules
    - get_definitions() - extracts name/units/description/category
    - get_value() - parameter access with caching and defaults

Subclasses only need to implement:
    - calculate_derivatives_and_flows() - required by VMS standard
    - your physics methods - the actual model logic

That's it! No boilerplate needed.
"""

def __init__(self, params: Dict[str, Any]):
    """Initialize with parameter dictionary."""
    self.params = params
    self._variable_cache = {}

def get_value(self, key: str) -> Any:
    """Get parameter value with default fallback and caching."""
    if key not in self._variable_cache:
        variables = self.__class__.get_variables()
        default = variables.get(key, {}).get('default', None)
        if default is None:
            raise KeyError(f"Unknown parameter: {key}")
        self._variable_cache[key] = self.params.get(key, default)
    return self._variable_cache[key]

@classmethod
def get_variables(cls) -> Dict[str, Any]:
    """Return this submodel's variables.

    Default implementation tries to find VARIABLES in the module.
```

```
        Subclasses can override this method if they use a different pattern.
        """
        # Try to auto-detect VARIABLES from the module
        import sys

        # First try the normal module resolution
        if cls.__module__ in sys.modules and cls.__module__ != 'builtins':
            module = sys.modules[cls.__module__]
            if hasattr(module, 'VARIABLES'):
                return getattr(module, 'VARIABLES').copy()

        # Fallback for Pyodide/browser environments where __module__ might be
'builtins'
        # In this case, look for VARIABLES in the same namespace as the class
        try:
            # Get the globals from the frame where this method is being called
            import inspect
            frame = inspect.currentframe().f_back
            caller_globals = frame.f_globals
            if 'VARIABLES' in caller_globals:
                return caller_globals['VARIABLES'].copy()
        except:
            pass

        # Another fallback: check __main__ module
        try:
            import __main__
            if hasattr(__main__, 'VARIABLES'):
                return getattr(__main__, 'VARIABLES').copy()
        except:
            pass

        raise NotImplementedError(
            f"Subclass {cls.__name__} must implement get_variables() "
            f"or define VARIABLES dict in module {cls.__module__}. "
            f"Module info: {cls.__module__} in {list(sys.modules.keys())[-3:] if
sys.modules else 'no modules'}"
        )

    @classmethod
    def get_defaults(cls) -> Dict[str, Any]:
        """Extract just the default values."""
        return {key: var.get('default') for key, var in
cls.get_variables().items()}
```

```
    @classmethod
    def get_validation(cls) -> Dict[str, Any]:
        """Extract just the validation rules."""
        validation = {}
        for key, var in cls.get_variables().items():
            val_rules = {rule: var[rule] for rule in ['min', 'max', 'step',
'type'] if rule in var}
            if val_rules:
                validation[key] = val_rules
        return validation

    @classmethod
    def get_definitions(cls) -> Dict[str, Any]:
        """Extract just the human-readable definitions."""
        definitions = {}
        for key, var in cls.get_variables().items():
            def_data = {field: var[field] for field in ['name', 'units',
'description', 'category'] if field in var}
            if def_data:
                definitions[key] = def_data
        return definitions

    def calculate_derivatives_and_flows(self, sim_context) -> Tuple[List[float],
Dict[str, float]]:
        """Calculate derivatives and flows for this submodel.

           Args:
               sim_context: SimulationContext containing current time, parameters,
       and stock values.

           Returns:
               A tuple containing:
               - List of derivative values for this submodel's stocks
               - Dictionary of named flow values
           """
        raise NotImplementedError("Submodels must implement
calculate_derivatives_and_flows()")


class ValidatableModelStandard:
    """
    Lightweight base class for validatable system dynamics models.

    Provides minimal structure - models should implement their own explicit logic.
    """
```

```
    def __init__(self, params: Dict[str, Any]):
        """Initialize with parameter dictionary."""
        self.params = params

    def _clean_params(self, params: Dict[str, Any]) -> Dict[str, Any]:
        """Clean parameter dictionary by extracting 'value' if needed."""
        cleaned = {}
        for key, val in params.items():
            if isinstance(val, dict) and 'value' in val:
                cleaned[key] = val['value']
            else:
                cleaned[key] = val
        return cleaned

    def _resolve_parameters(self, user_params: Dict[str, Any], all_variables:
Dict[str, Any]) -> Dict[str, Any]:
        """Resolve parameters with defaults and validation - explicit and
clear."""
        resolved = {}

        # Start with defaults explicitly
        for key, var_def in all_variables.items():
            resolved[key] = var_def.get('default', 0.0)

        # Override with user parameters, applying validation explicitly
        for key, value in user_params.items():
            if key in all_variables:
                var_def = all_variables[key]

                # Apply type-specific validation explicitly
                if var_def.get('type') == 'slider':
                    # Apply min/max constraints explicitly
                    min_val = var_def.get('min', value)
                    max_val = var_def.get('max', value)
                    value = max(min_val, min(max_val, float(value)))
                elif var_def.get('type') == 'boolean':
                    value = bool(value)

                resolved[key] = value
            else:
                # Allow unknown parameters for backward compatibility
                resolved[key] = value

        return resolved
```

```
    @classmethod
    def auto_discover_variables(cls):
        """
        Simple method for models to define their variables.
        Override this in each model to specify variables explicitly.
        """
        return {}



class SimulationContext:
    """Container for all simulation context information passed to submodels."""

    def __init__(self,
                 current_time: float,
                 current_datetime: datetime,
                 all_params: Dict[str, Any],
                 all_stocks: Dict[str, float],
                 intervention_active: bool = False):
        self.current_time = current_time  # Elapsed time in days
        self.current_datetime = current_datetime  # Actual datetime
        self.all_params = all_params  # All model parameters
        self.all_stocks = all_stocks  # All current stock values
        self.intervention_active = intervention_active  # Whether interventions
are active

    def get_param(self, key: str, default=None):
        """Get a parameter value with optional default."""
        return self.all_params.get(key, default)

    def get_stock(self, key: str, default=0.0):
        """Get a stock value with optional default."""
        return self.all_stocks.get(key, default)
```

model.py

```
Python
"""Master orchestrator for the Bathtub Inventory Dynamics model."""

from validatable_model_standard import ValidatableModelStandard, SimulationContext
from scipy.integrate import solve_ivp
import numpy as np
```

```python
from bathtub_submodel import BathtubSubmodel

SIMULATION_VARIABLES = {
    'simulation.duration': {
        'default': 100.0,
        'min': 10.0,
        'max': 500.0,
        'step': 1.0,
        'type': 'slider',
        'name': 'Simulation Duration',
        'units': 'day',
        'description': 'Total duration of the simulation',
        'category': 'parameter'
    },
    'simulation.time_step': {
        'default': 1.0,
        'min': 0.01,
        'max': 5.0,
        'step': 0.01,
        'type': 'slider',
        'name': 'Simulation Time Step',
        'units': 'day',
        'description': 'Time step for numerical integration',
        'category': 'parameter'
    }
}

class MasterModelOrchestrator(ValidatableModelStandard):
    """Master orchestrator for the Bathtub Inventory Dynamics model."""

    @classmethod
    def auto_discover_variables(cls):
        """Discover all variables from submodels - explicit and clear."""
        variables = {}
        variables.update(SIMULATION_VARIABLES)
        variables.update(BathtubSubmodel.get_variables())
        return variables

    def __init__(self, initial_params: dict = None):
        """Initialize model once - expensive operations here."""
        if initial_params is None:
            initial_params = {}
        initial_params = self._clean_params(initial_params)

        self.all_variables = self.auto_discover_variables()
```

```
        self.bathtub = BathtubSubmodel(self)

        self.output_config = self._load_output_config()
        self.stock_map = {
            'bathtub.s_water_level': 0
        }
        self.update_parameters(initial_params)
        self.captured_flows = {}

    def _load_output_config(self):
        """Load output configuration from config/outputs.json."""
        import json
        from pathlib import Path
        config_path = Path(__file__).parent.parent / "config" / "outputs.json"
        with open(config_path, 'r', encoding='utf-8') as f:
            config = json.load(f)
            return {
                'stocks': set(config.get('stocks', [])),
                'flows': set(config.get('flows', []))
            }

    def update_parameters(self, new_params: dict):
        """Update parameters without re-initializing the model."""
        new_params = self._clean_params(new_params)
        self.params = self._resolve_parameters(new_params, self.all_variables)

    def get_initial_conditions(self):
        """Get initial stock values using explicit parameter names."""
        initial_values = np.zeros(len(self.stock_map))
        for stock_param_name, index in self.stock_map.items():
            initial_values[index] = self.params.get(stock_param_name, 0.0)
        return initial_values

    def derivatives(self, t, stocks):
        """Calculate derivatives using submodels and capture flows during
calculation."""
        all_stocks = {}
        for stock_param_name, index in self.stock_map.items():
            all_stocks[stock_param_name] = stocks[index]

        sim_context = SimulationContext(
            current_time=t,
            current_datetime=None, # Not used in this simple model
            all_params=self.params,
            all_stocks=all_stocks
```

```
        )

        submodel_derivatives, submodel_flows =
self.bathtub.calculate_derivatives_and_flows(sim_context)

        combined_derivatives = np.zeros_like(stocks)
        for i, deriv in enumerate(submodel_derivatives):
            combined_derivatives[i] = deriv

        for flow_name, flow_value in submodel_flows.items():
            if flow_name not in self.captured_flows:
                self.captured_flows[flow_name] = []
            self.captured_flows[flow_name].append(flow_value)

        return combined_derivatives

    def run_simulation(self, params: dict = None):
        """Run simulation with optional parameter updates."""
        if params is not None:
            self.update_parameters(params)

        duration = self.params.get('simulation.duration', 100.0)
        time_step = self.params.get('simulation.time_step', 1.0)

        self.captured_flows = {}
        t_span = [0, duration]
        t_eval = np.linspace(t_span[0], t_span[1], int(duration /
time_step)).tolist()
        y0 = self.get_initial_conditions()

        # Initialize captured flows with empty lists aligned with time points
        for flow_name in self.output_config['flows']:
            self.captured_flows[flow_name] = [0.0] * len(t_eval)
        for stock_name in self.output_config['stocks']:
            if stock_name not in self.captured_flows:
                self.captured_flows[stock_name] = [0.0] * len(t_eval)

        solution = solve_ivp(self.derivatives, t_span, y0, t_eval=t_eval,
method='RK45')

        if solution.success:
            flow_results = {}
            stock_results = {}

            for stock_param_name, index in self.stock_map.items():
```

```
                if stock_param_name in self.output_config['stocks']:
                    stock_results[stock_param_name] = solution.y[index].tolist()

            for flow_param_name in self.output_config['flows']:
                if flow_param_name in self.captured_flows:
                    current_flows = self.captured_flows[flow_param_name]
                    if len(current_flows) > len(t_eval):
                        flow_results[flow_param_name] =
current_flows[:len(t_eval)]
                    elif len(current_flows) < len(t_eval):
                        flow_results[flow_param_name] = current_flows +
[current_flows[-1]] * (len(t_eval) - len(current_flows))
                    else:
                        flow_results[flow_param_name] = current_flows
                else:
                    flow_results[flow_param_name] = [0.0] * len(t_eval)

            json_safe_params = {}
            for key, value in self.params.items():
                if isinstance(value, np.ndarray):
                    json_safe_params[key] = value.tolist()
                elif isinstance(value, (np.integer, np.floating)):
                    json_safe_params[key] = float(value)
                else:
                    json_safe_params[key] = value

            return {
                'success': True,
                'time': solution.t.tolist(),
                'stocks': stock_results,
                'flows': flow_results,
                'parameters': json_safe_params
            }
        else:
            return {'success': False, 'error': f'Integration failed:
{solution.message}'}


# Interactive model functions
def initialize_model(initial_params=None):
    """Create a reusable model instance and return it with default results."""
    if initial_params is None:
        initial_params = {}
    model = MasterModelOrchestrator(initial_params)
    default_results = model.run_simulation()
```

```
    return model, default_results

def run_interactive_simulation(model, params=None, return_json=True):
    """Run simulation on existing model instance with new parameters."""
    results = model.run_simulation(params)
    if return_json:
        import json
        return json.dumps(results, indent=2)
    else:
        return results


def run_model(params=None, return_json=True):
    """Run a complete simulation with optional parameters."""
    model = MasterModelOrchestrator(params)
    results = model.run_simulation()
    if return_json:
        import json
        return json.dumps(results, indent=2)
    else:
        return results
```

bathtub_submodel.py

```
Python
from validatable_model_standard import Submodel
import numpy as np

VARIABLES = {
    'bathtub.s_water_level': {
        'default': 50.0,
        'min': 0.0,
        'max': 100.0,
        'step': 1.0,
        'type': 'slider',
        'name': 'Water Level',
        'units': 'units',
        'description': 'Current amount of water in the bathtub',
        'category': 'stock'
    },
    'bathtub.tap_flow_rate': {
        'default': 10.0,
```

```
        'min': 0.0,
        'max': 50.0,
        'step': 1.0,
        'type': 'slider',
        'name': 'Tap Flow Rate',
        'units': 'units per day',
        'description': 'Rate at which water flows from the tap',
        'category': 'parameter'
    },
    'bathtub.drain_flow_rate': {
        'default': 8.0,
        'min': 0.0,
        'max': 50.0,
        'step': 1.0,
        'type': 'slider',
        'name': 'Drain Flow Rate',
        'units': 'units per day',
        'description': 'Base rate at which water drains',
        'category': 'parameter'
    },
    'bathtub.drain_sensitivity_to_level': {
        'default': 0.1,
        'min': 0.0,
        'max': 1.0,
        'step': 0.05,
        'type': 'slider',
        'name': 'Drain Sensitivity',
        'units': 'units per day per unit',
        'description': 'How much drain flow increases with water level',
        'category': 'parameter'
    },
    'bathtub.tap_dt': {
        'default': 10.0,
        'min': 0.0,
        'max': 50.0,
        'step': 1.0,
        'type': 'rate',
        'name': 'Tap Flow',
        'units': 'units per day',
        'description': 'Rate of water entering the bathtub',
        'category': 'rate'
    },
    'bathtub.drain_dt': {
        'default': 8.0,
        'min': 0.0,
```

```
        'max': 50.0,
        'step': 1.0,
        'type': 'rate',
        'name': 'Drain Flow',
        'units': 'units per day',
        'description': 'Rate of water leaving the bathtub',
        'category': 'rate'
    }
}

class BathtubSubmodel(Submodel):

    def calculate_derivatives_and_flows(self, sim_context):
        water_level = sim_context.get_stock('bathtub.s_water_level', 0.0)
        tap_flow_rate = sim_context.get_param('bathtub.tap_flow_rate', 0.0)
        drain_flow_rate = sim_context.get_param('bathtub.drain_flow_rate', 0.0)
        drain_sensitivity =
sim_context.get_param('bathtub.drain_sensitivity_to_level', 0.0)

        # Calculate the actual drain flow, which depends on base rate and water
level
        # Ensure drain flow does not exceed water level, though in this simplified
model it can conceptually be negative if demand exceeds supply.
        # For a physical bathtub, drain flow is also limited by water_level, but
we keep it simple here.
        actual_drain_flow = drain_flow_rate + drain_sensitivity * water_level
        actual_drain_flow = max(0.0, actual_drain_flow) # Ensure drain flow is not
negative

        # Calculate derivatives
        # d(water_level)/dt = tap flow - drain flow
        d_water_level_dt = tap_flow_rate - actual_drain_flow

        # Calculate flows
        tap_dt = tap_flow_rate
        drain_dt = actual_drain_flow

        derivatives = [d_water_level_dt]
        flows = {
            'bathtub.tap_dt': tap_dt,
            'bathtub.drain_dt': drain_dt
        }

        return derivatives, flows
```

```
    def get_current_flows(self, sim_context):
        water_level = sim_context.get_stock('bathtub.s_water_level', 0.0)
        tap_flow_rate = sim_context.get_param('bathtub.tap_flow_rate', 0.0)
        drain_flow_rate = sim_context.get_param('bathtub.drain_flow_rate', 0.0)
        drain_sensitivity =
sim_context.get_param('bathtub.drain_sensitivity_to_level', 0.0)

        actual_drain_flow = drain_flow_rate + drain_sensitivity * water_level
        actual_drain_flow = max(0.0, actual_drain_flow)

        return {
            'bathtub.tap_dt': tap_flow_rate,
            'bathtub.drain_dt': actual_drain_flow
        }
```

ast_dependency_extractor_demo.py:

Python

```python
#!/usr/bin/env python3
"""Illustrative AST-based dependency extraction for VMS models."""

import ast


class DependencyExtractor(ast.NodeVisitor):
    def __init__(self):
        self.dependencies = []
        self.assigned_names = set()

    def visit_Assign(self, node):
        # Track local variable assignments.
        for target in node.targets:
            if isinstance(target, ast.Name):
                self.assigned_names.add(target.id)
        self.generic_visit(node)

    def visit_Call(self, node):
        # Extract sim_context.get_stock("...") and sim_context.get_param("...").
        if isinstance(node.func, ast.Attribute):
            method_name = node.func.attr
            if method_name in {"get_stock", "get_param"} and node.args:
                arg = node.args[0]
```

```
                if isinstance(arg, ast.Constant) and isinstance(arg.value, str):
                    self.dependencies.append((method_name, arg.value))
        self.generic_visit(node)


source_code = '''
class BathtubSubmodel(Submodel):
    def calculate_derivatives_and_flows(self, sim_context):
        water_level = sim_context.get_stock('bathtub.s_water_level', 0.0)
        tap_flow_rate = sim_context.get_param('bathtub.tap_flow_rate', 0.0)
        drain_flow_rate = sim_context.get_param('bathtub.drain_flow_rate', 0.0)
        drain_sensitivity =
sim_context.get_param('bathtub.drain_sensitivity_to_level', 0.0)

        actual_drain_flow = drain_flow_rate + drain_sensitivity * water_level
        actual_drain_flow = max(0.0, actual_drain_flow)

        d_water_level_dt = tap_flow_rate - actual_drain_flow

        flows = {
            'bathtub.tap_dt': tap_flow_rate,
            'bathtub.drain_dt': actual_drain_flow
        }

        return [d_water_level_dt], flows
'''


def main():
    tree = ast.parse(source_code)
    extractor = DependencyExtractor()
    extractor.visit(tree)

    for dep_type, var_name in extractor.dependencies:
        print(f"{dep_type}: {var_name}")


if __name__ == "__main__":
    main()
```

Sample output from ast parser:

```shell
> python.exe ast_dependency_extractor_demo.py
get_stock: bathtub.s_water_level
get_param: bathtub.tap_flow_rate
get_param: bathtub.drain_flow_rate
get_param: bathtub.drain_sensitivity_to_level
```

outputs.json

```json
{
  "_readme": {
    "purpose": "Configuration for which stocks and flows to return in simulation results",
    "usage": "All stocks/flows are calculated internally, but only those listed here are returned to frontend",
    "custom": "Modify these lists to control what data is returned without changing code"
  },
  "stocks": [
    "bathtub.s_water_level"
  ],
  "flows": [
    "bathtub.tap_dt",
    "bathtub.drain_dt"
  ]
}
```

test_model.py

```python
#!/usr/bin/env python3
"""
Test script for Generated Model model.
Generated automatically to validate model functionality.
"""

import sys
import os
```

```
from pathlib import Path

# Add src directory to path
sys.path.insert(0, str(Path(__file__).parent / "src"))

def test_model_import():
    """Test that the model can be imported successfully."""
    try:
        from model import run_model, MasterModelOrchestrator
        print("✅ Model import successful")
        return True
    except Exception as e:
        print(f"❌ Model import failed: {e}")
        return False

def test_model_initialization():
    """Test that the model can be initialized."""
    try:
        from model import MasterModelOrchestrator
        model = MasterModelOrchestrator()
        print("✅ Model initialization successful")
        return True
    except Exception as e:
        print(f"❌ Model initialization failed: {e}")
        return False

def test_model_run():
    """Test that the model can run a basic simulation."""
    try:
        from model import run_model
        result = run_model(return_json=False)

        if result.get('success', False):
            print("✅ Model run successful")
            print(f"   Stocks returned: {list(result.get('stocks', {}).keys())}")
            print(f"   Flows returned: {list(result.get('flows', {}).keys())}")
            return True
        else:
            print(f"❌ Model run failed: {result.get('error', 'Unknown error')}")
            return False
    except Exception as e:
        print(f"❌ Model run failed with exception: {e}")
        return False

def main():
```

```
    """Run all tests."""
    print(f"🧪 Testing Generated Model model...")

    tests = [
        ("Import Test", test_model_import),
        ("Initialization Test", test_model_initialization),
        ("Run Test", test_model_run)
    ]

    passed = 0
    total = len(tests)

    for test_name, test_func in tests:
        print(f"\n🔬 Running {test_name}...")
        if test_func():
            passed += 1

    print(f"\n🏁 Test Results: {passed}/{total} tests passed")

    if passed == total:
        print("🎉 All tests passed! Model is ready to use.")
        return True
    else:
        print("⚠️  Some tests failed. Please check the model implementation.")
        return False

if __name__ == "__main__":
    main()
```

Example output of running test_model.py:

```
None
🧪 Testing Generated Model model...

🔬 Running Import Test...
✅ Model import successful

🔬 Running Initialization Test...
✅ Model initialization successful

🔬 Running Run Test...
✅ Model run successful
```

```
   Stocks returned: ['bathtub.s_water_level']
   Flows returned: ['bathtub.drain_dt', 'bathtub.tap_dt']

🏁 Test Results: 3/3 tests passed
```

Example result data from `result = run_model(return_json=False)` (abbreviated for illustration purposes):

JSON


```
{'success': True, 'time': [0, 1, 2, 3, ..., 18, ...], 'stocks':
{'bathtub.s_water_level': [...]}, 'flows': {'bathtub.drain_dt': [...],
'bathtub.tap_dt': [...]}, 'parameters': {'simulation.duration': 100.0,
'bathtub.s_water_level': 50.0, 'bathtub.tap_flow_rate': 10.0,
'bathtub.drain_flow_rate': 8.0, 'bathtub.drain_sensitivity_to_level': 0.1,
'bathtub.tap_dt': 10.0, 'bathtub.drain_dt': 8.0, ...}}
```


Plot of results['stocks']['bathtub.s_water_level'] over results['time']:

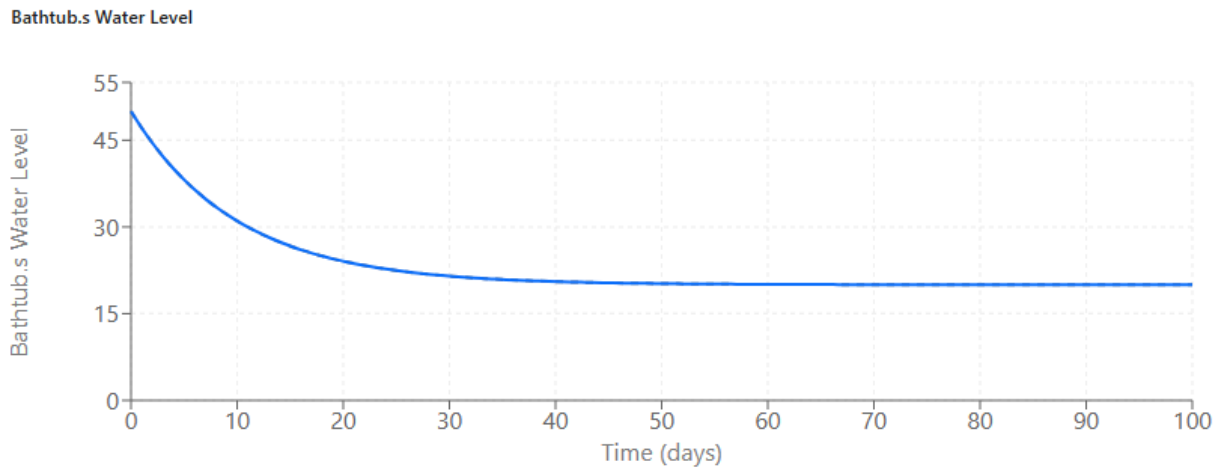

Figure A1: Output for AI-Code-First Bathtub example using default values as implemented in the code provided in Appendix A.

LICENSE

None

MIT License

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```