SPECIFICATION-DRIVEN GENERATION AND EVALUATION OF DISCRETE-EVENT WORLD MODELS VIA THE DEVS FORMALISM

Anonymous authors
Paper under double-blind review

ABSTRACT

World models are central to LLM agents that must evaluate actions over long horizons. Yet much existing work focuses on environments governed by physical dynamics or spatial structure, whereas many high-impact domains, including supply chains, procurement networks, and business processes, evolve through discrete events, timing constraints, and causal dependencies. These settings call for discrete-event world models. Hand-engineered simulators provide consistency but are costly to build and adapt, while neural models can accumulate inconsistencies over long rollouts. We introduce DEVS-GEN, an LLM pipeline that generates discrete-event simulators from natural-language specifications using the Discrete Event System Specification (DEVS) formalism. It first plans the component hierarchy and interfaces, then synthesizes component behavior under these interface constraints. For evaluation, we develop a benchmark based on completed specifications that state the required interface and trace-observable behavior. We validate simulator traces against these operational and behavioral requirements, enabling automated scoring and rule-level diagnostics. When averaged across four backbones, DEVS-GEN achieves the highest operational and behavioral scores among the evaluated methods, although its relative performance varies by backbone. Its behavioral gains over both non-executing baselines are statistically significant, whereas those over the execution-enabled baselines are not. It also requires less generation time and fewer tokens to reach higher operational and behavioral performance than the baselines.

1 INTRODUCTION

LLM agents increasingly need to learn from and plan within complex environments. This makes *world models*, which support hypothetical rollouts under candidate actions, increasingly important. They matter for two reasons: (i)agentic reinforcement learning relies on experience-driven scaling, where world models can generate simulated experience to improve learning efficiency (Ha & Schmidhuber, 2018). (ii)LLM agents are expected to operate over long horizons, where delayed consequences make reliable reasoning difficult; world models let agents simulate downstream effects before acting, supporting better decisions (Hao et al., 2023; Gu et al., 2025). In novel settings, however, these models cannot always be pre-built and must be constructed or adapted on demand.

A growing body of work has explored world models across domains (Parker-Holder & Fruchter, 2025; Feng et al., 2025), but much of it focuses on environments governed by physical dynamics or spatial structure, such as robotic control, visual navigation, and interactive visual worlds. Other consequential settings include supply chains, procurement networks, service operations, and business processes (de Jong et al., 2019; Chui et al., 2021; Next Generation Technologies Research Team, 2025). Such systems often involve discrete interactions among multiple entities, with outcomes shaped by the ordering, timing, and causal dependencies of events. Their structure motivates world models that represent system-level processes and the effects of interventions across interacting components.

This motivates the study of *discrete-event world models*: models in which state changes are triggered by discrete events with explicit timing and whose observable behavior is represented by event traces.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Existing methods for building world models tend to fall near one of two extremes. At one end, *hand-engineered simulators* explicitly encode domain rules and process logic, but require substantial effort to build and adapt(**?**). At the other end, *implicit neural approaches* predict future states directly (Hao et al., 2023; Li et al., 2026), but can accumulate inconsistencies over long rollouts. In process-driven domains, these inconsistencies can take the form of invalid event sequences, violated resource constraints, or conflicting entity states. Therefore, a valuable middle ground remains underdeveloped: world models that retain consistency over long rollouts, support behavioral checks through observable traces, and can be built on demand.

In this paper, we investigate whether LLMs can automate the generation of explicit discrete-event world models from natural-language specifications. The specification describes how the environment behaves, and the generated world model is a simulator that implements this behavior in executable code. During each rollout, the simulator follows explicit state-transition rules, preserving the long-horizon consistency of explicit simulation.

This raises two key challenges. First, realistic environments are structurally complex, and need to coordinate the behavior of components. As a result, single-pass generation often fails because one localized error can invalidate the full system. Iterative coding agents can identify and repair errors through execution feedback, but repeated debugging increases token use and generation time. This delay is particularly costly when planning must wait for a usable simulator. Second, evaluation is difficult because a specification may admit multiple valid simulators with different internal abstractions that satisfy the same stated behavior. Code-level equivalence is therefore ill-posed. Key performance indicators (KPIs) alone are insufficient: they can conceal violations of event-level requirements, and differences in KPIs need not imply incorrect behavior.

To address the generation challenge, we introduce DEVS-GEN, an LLM pipeline built on the Discrete Event System Specification (DEVS) formalism (Zeigler et al., 2000; Risco-Martín et al., 2023). DEVS separates local component behavior from interactions routed through declared ports. Leveraging this separation, DEVS-GEN first plans the system hierarchy and port interfaces, then synthesizes component logic under these interaction contracts. By defining interfaces before implementation, the pipeline lets components be synthesized separately and then assembled into a simulator. This separation limits each generation call to a component's responsibilities and interfaces, enables parallel generation, and reduces reliance on costly global debugging loops.

For evaluation, we introduce a benchmark based on *trace-based conformance*: whether event trace satisfies the temporal, causal, and semantic constraints stated in the specification. Drawn from curated DEVS examples, its tasks measure *operational validity* using $\mathrm{Score}_{\mathrm{ope}}$, covering the I/O contract and trace schema, and *behavioral conformance* using $\mathrm{Score}_{\mathrm{beh}}$, covering component transitions, cross-component dependencies, and aggregate behavior. This approach permits different implementations and is conceptually related to conformance testing (Tretmans, 2008) and runtime verification through execution traces (Leucker & Schallhart, 2009).

Together, we yield a framework for generating and evaluating discrete-event world models. Our experiments discover: (i) DEVS-GEN achieves the highest $\mathrm{Score}_{\mathrm{ope}}$ and $\mathrm{Score}_{\mathrm{beh}}$ among the execution-disabled baselines. However, its comparisons with the execution-enable baselines vary by backbone. (ii) At higher score levels, DEVS-GEN reaches the same scores as the baselines with less generation time and fewer tokens. (iii) ablations reveal an architectural trade-off: directly enforcing DEVS rules imposes a cognitive penalty on LLMs. DEVS-GEN successfully absorbs this burden through its planning, surpassing the performance of unconstrained single-pass baselines.

The generated hierarchy and traces also support reuse beyond simulator execution. We outline a proof-of-concept extension that couples the generated simulator to a physics engine, using DEVS for task logic and physical feedback to update task progress. We also include an exploratory supply-chain study to test whether DEVS-GEN can help refine policy. The resulting policies achieve lower costs than direct policy generation in all settings. They also outperform a coding agent using its own supply-chain approximations in all scenes with GPT-5.2, but not consistently with GLM-4.7.

2 DISCRETE-EVENT WORLD MODELS AND DEVS

In this section, we define the world model generation task and introduce DEVS, the formalism used by our method.

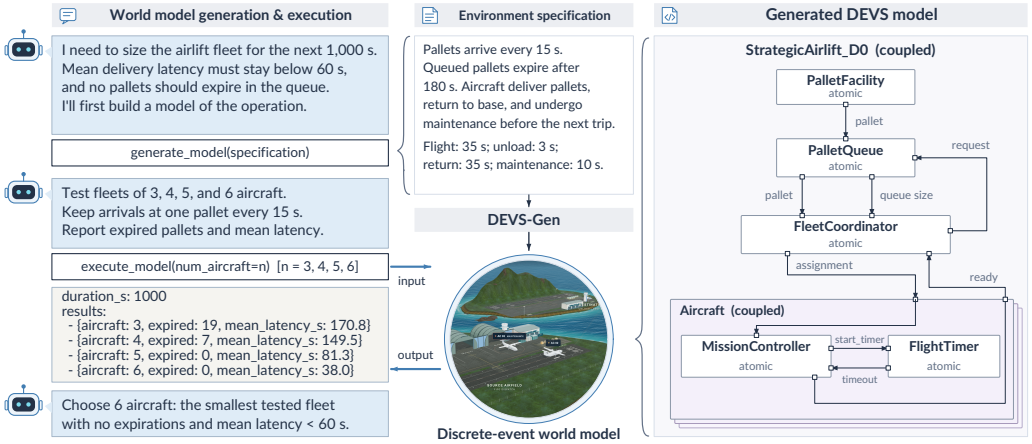

Figure 1: Generation and use of a discrete-event world model, illustrated with StratAirlift. DEVS-GEN translates a natural-language specification into a DEVS model. Executing the model produces event traces from which performance measures are derived for what-if analysis and planning. Here it uses the measures to select a fleet size that avoids expiration and meets the latency constraint.

2.1 PROBLEM SETUP

We study the generation of *executable discrete-event world models* from *natural-language specifications*. Figure 1 illustrates this pipeline with Strategic Airlift, where the input text describes pallets and aircraft, and events such as cargo assignment and delivery. The generated simulator can be used within an agent's reasoning loop for online planning or executed as an offline environment.

The *discrete-event world* abstraction targets environments whose decision-relevant behavior is driven by event timing, resource constraints, and multi-entity interaction rather than continuous physics. This includes settings such as service operations, supply chains, business processes, and distributed workflows, where agents must reason about queues, delays, synchronization, and shared resources. App. A compares it with MDP- and PDDL-based representations.

Discrete-Event World. A discrete-event world is an environment whose state changes at discrete event times and remains constant between them. In the airlift example, the state includes waiting pallets and each aircraft's transport or maintenance phase. Events trigger state changes.

Formally, we describe such a world by $\mathcal{W} = (\mathcal{E}, \mathcal{S}, \mathcal{U}, \delta)$. Here, $\mathcal{E}$ is a finite set of components, such as a pallet queue or an aircraft controller. The world state is $\mathcal{S} = \prod_{e \in \mathcal{E}} \mathcal{S}_e$, where $\mathcal{S}_e$ is the state space of component e. The event set $\mathcal{U}$ consisted of instances $u = (\ell, p) \in \mathcal{U} \subset \Omega \times \mathcal{P}$, where Ω is the set of event types and $\mathcal{P}$ is the set of payload data. For example, a delivery-completion event may carry aircraft and pallet IDs. Let $\mathcal{B}(\mathcal{U})$ denote the space of finite multisets of events. The transition function $\delta : \mathcal{S} \times \mathcal{B}(\mathcal{U}) \times \mathbb{R}_{\geq 0} \to \mathcal{S} \times \mathcal{B}(\mathcal{U}) \times \mathbb{R}_{\geq 0}$ takes the current state s, the incoming events B_{in}, and the elapsed time d, returns the updated state s', the events B_{out} emitted at the current time, and the waiting time τ until the next internal transition. After each update, the world waits until either an external event arrives or the scheduled internal transition becomes due.

World Models as Simulators. While $\mathcal{W}$ defines the abstract rules of the environment, an agent requires a concrete, interactable software artifact. We use $\mathcal{M}$ to denote the generated simulator: the executable discrete-event world model. The $\mathcal{M}$ implements a concrete interpretation of $\mathcal{W}$. Here we describe it as a black box with a fixed input–output interface specification. At launch, initialization values are drawn from a set $\mathcal{A}$ and provided as configuration arguments $\mathcal{I} = (a_1, a_2, \ldots, a_m), a_k \in \mathcal{A}$, such as the number of aircraft, pallet arrival interval, and simulation horizon. Technically, the simulator uses $\mathcal{I}$ to construct an initialization state s_{init} and the waiting time τ_0 until the first internal transition. Some tasks additionally provide an external input stream $\mathcal{J}$ during execution. We treat the overall external input to $\mathcal{M}$ as the pair $(\mathcal{I}, \mathcal{J})$. Executing $\mathcal{M}(\mathcal{I}, \mathcal{J})$ produces an event trace $\mathcal{T} = (r_1, r_2, \ldots)$. Each record $r = (t, e, \ell, p) \in \mathbb{R}_{\geq 0} \times \mathcal{E} \times \Omega \times \mathcal{P}$ is a time-stamped output, such as a pallet arrival or an aircraft completing maintenance at time t.

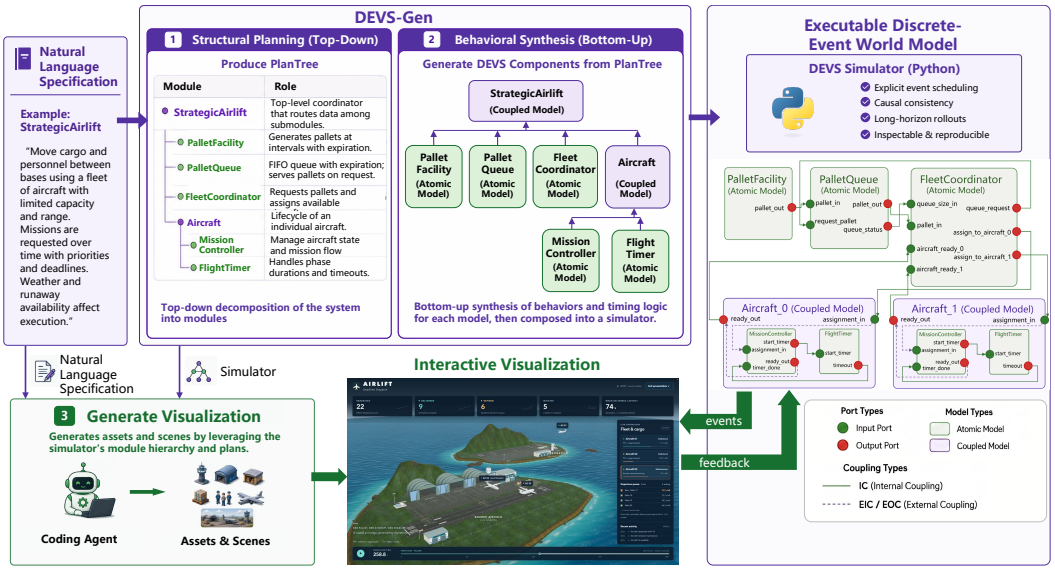

Figure 2: Illustration of DEVS-GEN. Structural planning specifies a model hierarchy, and behavioral synthesis implements its components, finally generating an executable discrete-event world model. The lower panels show a visualization extension and its interaction with the simulator.

World Model Generation Task. The task is to generate an $\mathcal{M}$ from a natural-language specification `Spec` of $\mathcal{W}$. The `Spec` describes the world's entities, events, causal and temporal dynamics, and the interface through which the model is executed and observed (see App. I.1 for example).

2.2 DEVS AS AN OPERATIONAL REPRESENTATION

The discrete-event world abstraction above does not commit us to an implementation formalism. In principle, such environments could be realized using other executable representations, including domain-specific simulators, Petri nets, queueing networks, or other explicit state-transition models. We choose Parallel DEVS (PDEVS) (Zeigler et al., 2000; Risco-Martín et al., 2023) because it supports hierarchical composition, defines event timing and concurrent interactions, and separates local component behavior from system-level routing. The formal definition of PDEVS is provided in App. C. For brevity, we refer to PDEVS simply as DEVS hereafter. All DEVS-targeting methods in our experiments generate Python simulators using the `xdevs.py` library. Concretely, DEVS realizes this modular structure through two component types:

Atomic Models (Behavioral Logic). Atomic models encapsulate local state, timing semantics, and event-handling rules, e.g., those of a pallet queue or aircraft controller. They autonomously update local state and emit events in response to elapsed simulated time or incoming events.

Coupled Models (Structural Routing). Coupled models define the system architecture by composing sub-components and specifying how events are routed among their input and output ports. They carry no local state or behavioral logic; instead, they make the communication structure explicit.

By recursively assembling these components, we obtain the hierarchical DEVS model. Complex system-level dynamics, e.g., a coordinator dispatching pallets across multiple aircraft, emerge from event passing across this topology. In the next section, we leverage this structure to separate structural planning of component interactions from behavioral synthesis of individual components.

3 DEVS-BASED MODEL GENERATION PIPELINE

Building on the DEVS formalization, we introduce DEVS-GEN, a pipeline for generating an executable discrete-event world model $\mathcal{M}$ from a specification `Spec`. The key idea is to use DEVS as a structural scaffold and decompose generation into a sequence of well-scoped sub-tasks. DEVS-

GEN constructs $\mathcal{M}$ in two stages: (i) *structural planning* derives a `PlanTree` from `Spec`, which assigns each component a local `ModelPlan`. (ii) *behavioral synthesis* implements each component according to its `ModelPlan` and assembles the resulting modules into $\mathcal{M}$. The upper half of Fig. 2 illustrates this procedure with the Strategic Airlift example. App. I further presents it in detail.

The intermediate representation `PlanTree` connects these two stages. It is a hierarchical blueprint that specifies the simulator topology, assigns each node a function, and defines port-interface contracts. Each node corresponds to a simulator component, such as a pallet queue or an aircraft controller, and stores a `ModelPlan` describing that component (data schema in App. F). The `ModelPlan` acts as a local contract: it specifies the component's model type, arguments, I/O port schemas, responsibilities for external input and output, and intended behavior. Atomic-model's plans also describe local state, events, and timing logic. Coupled-model's plans instead specify how events are routed between direct children and across the coupled model's boundary.

Detailed algorithms are in App. D, and the prompt templates used for LLM calls are in App. G.

3.1 STAGE 1: STRUCTURAL PLANNING

System Skeleton Drafting. The pipeline first extracts a requirement index $\mathcal{Q} = \{q_k\}_{k=1}^{K}$ from `Spec`. Each entry q_k records a self-contained requirement and links to the source sections from which it was derived. Given `Spec` and $\mathcal{Q}$, the planner generates a `Skeleton` whose nodes identify their children, state their responsibilities, and reference the relevant requirements.

Top-Down Component Planning. The planner then refines the `Skeleton` from the top down, generating a `ModelPlan` for each node (see Alg. 2). It generates the root plan in two steps: the first produces a draft, and the second revises it into the final plan. This procedure is designed for more stable generation. Each parent plan establishes the interface constraints for its children, whose plans can then be generated independently and in parallel. At each coupled node, the planner assigns port-interface contracts to the children and writes one routing rule per required coupling. At each atomic node, refinement specifies local state, events, timing, and any direct external I/O.

The `PlanTree` is the `Skeleton` after every node has been assigned a `ModelPlan`. It provides the component plans and hierarchy used in Stage 2. App. I.3 shows an example.

3.2 STAGE 2: BEHAVIORAL SYNTHESIS

Parallel Component Synthesis. Behavioral synthesis proceeds bottom-up through the `PlanTree` (Alg. 3). Atomic models are generated from their local plans, with prompts specifying transitions, timing, and trace emissions. A coupled model is generated only after its child modules are available, with its prompt specifying child instantiation and port routing. Because Stage 1 has already specified the ports and routing relationships, sibling subtrees can be synthesized concurrently. Each generation call handles one component and its interfaces, limiting the context length.

Bottom-Up Interface Reconciliation. Generated child modules may slightly differ from their `ModelPlan` in local identifiers or constructor signatures. Before generating a parent module, the pipeline parses each child module's Python abstract syntax tree to extract a *construction excerpt*: complete constructor and helper methods called during initialization. The parent-generation prompt receives this excerpt together with the child's `ModelPlan` as its *child context*. When the two conflict, the excerpt takes precedence because it reflects the generated implementation. The resulting parent module imports and instantiates its children using these interfaces and connects their ports according to its coupling rules.

Assembly and Execution Entry. Once the root model is complete, the pipeline generates a top-level script that instantiates the root model, applies the initialization $\mathcal{I}$, accepts the input $\mathcal{J}$, and launches the simulation. The generated modules and script together form the Python simulator $\mathcal{M}$.

3.3 COUPLING GENERATED WORLD MODELS WITH PHYSICS SIMULATION

We propose coupling the generated DEVS simulator with a physics engine such as MuJoCo for 3D visualization and embodied interaction (Figure 2). The component hierarchy and explicit interfaces provide attachment points for scene elements and controllers, while `PlanTree` records component

responsibilities and event-routing relationships to guide the coupling. A coding agent would use the specification, simulator project, `PlanTree`, and physical assets and scene parameters to generate the scene, controller, and coupling code. DEVS would govern task logic. The coupling code would translate component events into scene updates, and feed physical outcomes back to DEVS; visualization would render the resulting physics state.

4 TRACE-BASED EVALUATION OF WORLD MODELS

The world model generation task requires an agent to generate a valid simulator $\mathcal{M}$ from a `Spec`. To evaluate this capability, we design a benchmark grounded in *trace-based conformance*: whether the emitted event traces $\mathcal{T}$ satisfy the dynamic, temporal, and causal constraints derived from the `Spec`. This yields a fine-grained, implementation-agnostic measure of simulator correctness.

4.1 TRACE-BASED EVALUATION CRITERIA

Our benchmark evaluates a generated simulator $\mathcal{M}$ against `Spec`. We organize each specification into an operational contract $\texttt{Spec}_{\text{ope}}$ and behavioral requirements $\texttt{Spec}_{\text{beh}}$. Both parts are translated into executable testing rules.

Operational Validity. A test case is operationally valid when the simulator executes without runtime errors and emits a structured event trace $\mathcal{T}$ in the prescribed schema.

Behavioral Conformance. For operationally valid test cases, we further evaluate whether the observable behavior conforms to $\texttt{Spec}_{\text{beh}}$. We implement rules at two granularities: (i)*Micro-level consistency*: checks on the state evolution of individual components. For example, in a queueing system, a rule verifies whether service times and waiting intervals follow the timing logic in the behavioral description. (ii)*Macro-level causality*: checks on causal and temporal relationships across components. These rules detect failures such as events occurring without their required preceding events, or deviations from required statistical behavior in stochastic settings.

4.2 DATASET CONSTRUCTION AND COMPOSITION

We construct a benchmark to evaluate these criteria. The benchmark scenarios are derived from selected, publicly available DEVS examples. We exclude examples whose primary task is outside simulator generation (such as controller-training demos), and near-duplicate variants whose dynamics are already represented by another scenario. For each scenario, we write a natural-language specification and implement checker rules for its operational validity and behavioral requirements.

The benchmark uses completed specifications. Here, completed means the execution interface and behavioral requirements used for scoring are specified. Behavior left unspecified is not checked.

Data Assets. Each benchmark scenario is a self-contained evaluation package with three parts: (1) an input `Spec`, partitioned into $\texttt{Spec}_{\text{ope}}$ and $\texttt{Spec}_{\text{beh}}$, that is provided to the generation method; (2) a test suite containing multiple $(\mathcal{I}, \mathcal{J})$ input pairs that exercise the specified causal dependencies under varied conditions; and (3) a checker package that computes the operational validity and behavioral conformance. Section 4.3 defines the scores in detail. All together, the benchmark contains 66 behavioral requirements (50 micro and 16 macro). App. J describes checker construction, and maps representative requirements to trace checks.

Dataset Composition. The benchmark contains ten scenarios spanning service workflows, network protocols, transportation, access control, thermal control, and epidemic modeling. Their dynamics include stochastic queues, scheduled delays, state machines, resource contention, feedback control, and approximations of differential equations. App. H describes scenarios in detail.

4.3 METRICS AND EXPERIMENTAL PROTOCOL

To score a generated simulator $\mathcal{M}$, we execute a test suite $\mathcal{D} = \{d_1, d_2, \ldots, d_N\}$. Test case d_i supplies inputs $(\mathcal{I}_i, \mathcal{J}_i)$ and produces an execution trace $\mathcal{T}_i = \mathcal{M}(\mathcal{I}_i, \mathcal{J}_i)$.

Operational Score $\text{Score}_{\text{ope}}$**.** It checks whether it's operationally valid. Let $\mathcal{R}_{\text{ope}} = \{q_j\}$ denote the trace-schema checks, with $q_j(\mathcal{T}_i) \in \{0, 1\}$ indicating whether check q_j is passed. We further

define the operational gate $v_i = \mathbb{I}(\text{ExitCode}_i = 0)\mathbb{I}(\text{NoTimeout}_i) \prod_{q_j \in \mathcal{R}_{\text{ope}}} q_j(\mathcal{T}_i)$. The operational score is the valid-case fraction, $\text{Score}_{\text{ope}} = \frac{1}{N} \sum_{i=1}^{N} v_i$. Full operational success is $\mathbb{I}(v_i = 1 \; \forall i)$.

Behavioral Score $\text{Score}_{\text{beh}}$**.** It evaluates its behavioral conformance. Let $\mathcal{R}_{\text{micro}}$ and $\mathcal{R}_{\text{macro}}$ contain the two granularities of behavioral requirements. Each rule has a quality function s_r taking values in $[0, 1]$. For rules evaluated on single trace, we compute $Q_r = \frac{1}{N} \sum_{i:v_i=1} s_r(d_i, \mathcal{T}_i)$. For rules evaluated jointly across traces, we compute $Q_r = \text{Score}_{\text{ope}}\, s_r(\{(d_i, \mathcal{T}_i) : v_i = 1\})$. The overall score is: $\text{Score}_{\text{beh}} = \frac{1}{2} \left(\frac{1}{|\mathcal{R}_{\text{micro}}|} \sum_{r \in \mathcal{R}_{\text{micro}}} Q_r + \frac{1}{|\mathcal{R}_{\text{macro}}|} \sum_{r \in \mathcal{R}_{\text{macro}}} Q_r \right)$.

5 EXPERIMENTS

In this section, we evaluate DEVS-GEN and several LLM-based coding-agents. Our evaluation addresses three questions: (i) *Effectiveness:* How reliably does each method produce operationally valid and behaviorally conformant simulators? (ii) *Resource use:* How much generation time and token budget does each method require to reach a given level of performance? (iii) *Ablation:* Does requiring the DEVS framework improve generation quality, and does staged generation provide further gains over single-pass generation under the same requirement?

5.1 EXPERIMENTAL SETUP

We conduct experiments using the benchmark in Section 4. We evaluate four API-served backbones: OpenAI GPT-5.2 (OpenAI, 2025), Z.ai GLM-4.7 (Z.AI, 2025), Qwen3-Coder-30B-A3B-Instruct (Yang et al., 2025), and Meta Llama 4 Scout 17B-16E Instruct (Meta AI, 2025). For the main effectiveness evaluation, we compare DEVS-GEN against two open-source agentic frameworks: OpenHands (Wang et al., 2026) and SWE-Agent (Yang et al., 2024). We evaluate their execution-enabled Standard configurations and the execution-disabled Lite variants, to isolate the impact of execution feedback. Like the Lite variants, DEVS-GEN is also a non-execution method. We also evaluate two direct-generation controls: single generates a free-form simulator, whereas single-xdevs generates an `xdevs.py`-based simulator. For every combination of backbone and method, we schedule three repetitions on each of ten scenarios.

5.2 EFFECTIVENESS

Table 1 shows that DEVS-GEN achieves the highest operational and behavioral scores when averaged across the four backbones, with GPT-5.2 yielding its strongest results. The advantage is not uniform across backbones. Llama-4-Scout scores remain low across all evaluated frameworks, indicating that successful generation remains dependent on the backbone.

Two-sided permutation tests stratified by backbone and scenario show that DEVS-GEN achieves significantly higher behavioral scores than both Lite variants (OpenHands-Lite: $p = 0.0038$; SWE-Agent-Lite: $p = 0.0002$). Its higher mean scores relative to the Standard agents do not reach significance (OpenHands: $p = 0.408$; SWE-Agent: $p = 0.226$). App. E.3 details the test.

For further diagnosis, App. E.1 reports generation-completion and full-operational-success counts, together with the corresponding conditioned scores.

5.3 EFFICIENCY

Figure 3 compares generation completion and simulator quality under different time and token budgets. For method m and outcome q, let N_m be the number of scheduled generations, c_j the corresponding cost of generation j, and $y_{j,q}$ the outcome value (its completion indicator, $\text{Score}_{\text{ope}}$, or $\text{Score}_{\text{beh}}$). For completion, $y_{j,q} = 1$ only when generation ends normally and provides the expected entry point. We define the *cumulative outcome* at budget b as $C_{m,q}(b) = \frac{1}{N_m} \sum_{j:c_j \le b} y_{j,q}$. For completion, this is the fraction of scheduled generations completed within budget; for the two scores, it is the mean across all generations, with results outside the budget contributing zero.

Table 1: Operational validity and behavioral conformance on ten benchmark scenarios across four backbones. Backbone columns report mean ± sample standard deviation over three repetitions, each averaged over the ten scenarios. Avg. reports the mean across the four backbones. Bold indicates the highest mean in each column within each metric, determined before rounding.

Method	GPT-5.2	GLM-4.7	Qwen3-Coder	Llama 4 Scout	Avg.
Operational validity ($\mathrm{Score}_{\mathrm{ope}}$)					
OpenHands-Lite	0.930 ± 0.061	0.651 ± 0.104	0.507 ± 0.100	0.125 ± 0.108	0.553
SWE-Agent-Lite	0.830 ± 0.052	0.509 ± 0.110	0.544 ± 0.050	0.200 ± 0.095	0.521
OpenHands	0.961 ± 0.052	0.871 ± 0.115	0.623 ± 0.063	0.092 ± 0.007	0.636
SWE-Agent	0.897 ± 0.171	0.717 ± 0.126	$\mathbf{0.668} \pm 0.029$	0.158 ± 0.146	0.610
single	0.944 ± 0.019	0.584 ± 0.045	0.392 ± 0.033	0.070 ± 0.061	0.498
single-xdevs	0.650 ± 0.045	0.084 ± 0.074	0.003 ± 0.006	0.000 ± 0.000	0.184
DEVS-GEN	$\mathbf{0.985} \pm 0.027$	$\mathbf{0.882} \pm 0.065$	0.610 ± 0.091	$\mathbf{0.200} \pm 0.000$	**0.669**
Behavioral conformance ($\mathrm{Score}_{\mathrm{beh}}$)					
OpenHands-Lite	0.846 ± 0.112	0.591 ± 0.056	0.402 ± 0.071	0.107 ± 0.093	0.487
SWE-Agent-Lite	0.785 ± 0.064	0.480 ± 0.101	0.427 ± 0.064	0.112 ± 0.059	0.451
OpenHands	0.855 ± 0.058	$\mathbf{0.809} \pm 0.086$	0.488 ± 0.056	0.057 ± 0.029	0.552
SWE-Agent	0.810 ± 0.139	0.667 ± 0.155	$\mathbf{0.555} \pm 0.060$	0.109 ± 0.094	0.535
single	0.893 ± 0.045	0.549 ± 0.064	0.316 ± 0.051	0.043 ± 0.057	0.450
single-xdevs	0.594 ± 0.083	0.065 ± 0.067	0.003 ± 0.006	0.000 ± 0.000	0.166
DEVS-GEN	$\mathbf{0.894} \pm 0.017$	0.770 ± 0.037	0.494 ± 0.064	$\mathbf{0.146} \pm 0.004$	**0.576**

DEVS-GEN generally requires less time and fewer tokens at higher outcome levels, although baselines can be cheaper at lower levels. For brevity, we focus the following diagnosis on $\mathrm{Score}_{\mathrm{beh}}$. Completion and $\mathrm{Score}_{\mathrm{ope}}$ show similar qualitative patterns.

We examine baseline generations at or below the corresponding resource-dominance threshold, separately for time and tokens. This threshold is the smallest budget from which DEVS-GEN's cumulative behavioral score remains no lower than the baseline's.

Positive-scoring outputs within these budgets often fail to recur across repetitions of the same setting, indicating run-to-run variability instead of a stable low-resource advantage. The three easiest scenarios account for a larger share of the summed baseline behavioral scores below or at these thresholds than above. App. E.2 reports the repetition counts and scenario-level score shares underlying these observations. On the time axis, baselines may produce positive results on easier scenarios while DEVS-GEN is still establishing the hierarchy and port interfaces. Once interfaces are fixed, sibling component calls can run in parallel, speeding up generation.

The token advantage may further reflect the advantage of component-wise synthesis: each call handles a component under fixed interfaces, rather than processing a whole growing project.

5.4 ABLATION STUDIES

Framework Constraints and Cognitive Load. Here, cognitive load refers to the combination of (i) the demands of DEVS component and interface constraints and (ii) the `xdevs.py` API. We compare three execution-disabled settings: single, single-xdevs, and DEVS-GEN.

Table 1 shows that, requiring `xdevs.py` lowers both scores under single-pass generation. Yet DEVS-GEN achieves higher mean scores than both single-pass controls across all four backbones. These results suggest that DEVS-GEN can leverage DEVS hierarchy and interfaces and absorb the additional cognitive load. App. E.5 provides details.

Parallel Execution. We compare serial and parallel execution in DEVS-GEN on GPT-5.2 over the benchmark scenarios. Structural planning decreases from 183.8 s to 132.2 s ($1.39\times$), while behavioral synthesis decreases from 71.8 s to 31.7 s ($2.27\times$). Across the full process, the mean decreases from 274.1 s to 182.3 s ($1.50\times$). Planning is only partly parallelizable because its longest

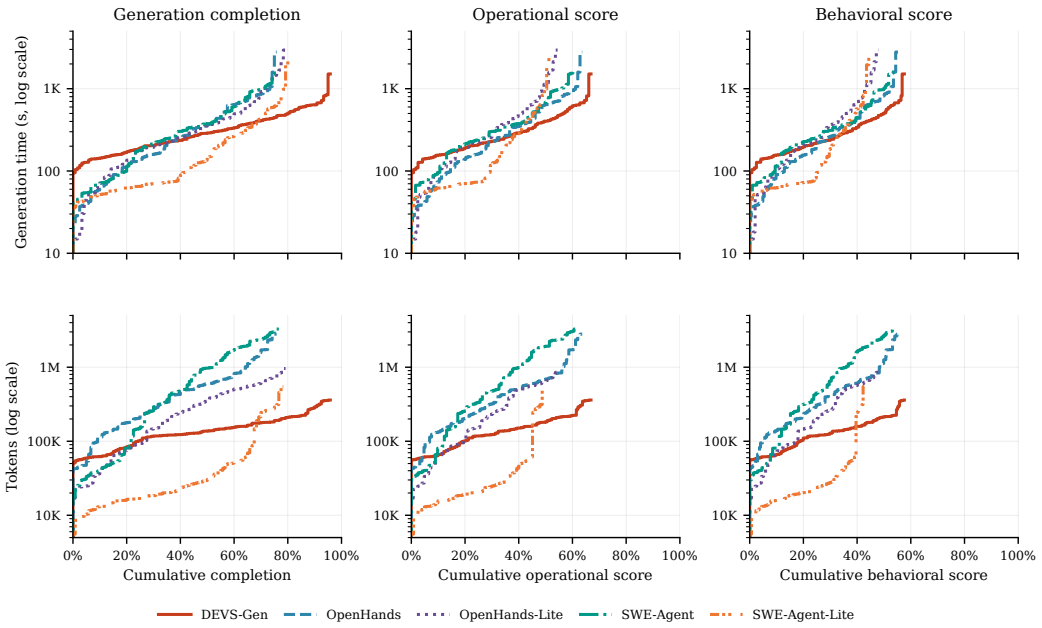

Figure 3: Resource-versus-outcome curves for generation completion (left), $\mathrm{Score}_{\mathrm{ope}}$ (middle), and $\mathrm{Score}_{\mathrm{beh}}$ (right). The horizontal axes show $C_{m,q}(b)$ as defined in the text. The vertical axes show generation time (top) or token use (bottom) on logarithmic scales.

Table 2: Normalized replenishment costs in three supply-chain scenarios. Each cost is divided by the lowest cost observed across all three methods and both backbones within the same scenario. Lower is better; bold marks the lowest cost in each scenario.

Backbone	**Policy generation**	**Scenario 01**	**Scenario 02**	**Scenario 03**
GLM-4.7	Direct-Policy	85.8×	2.44×	52.7×
GLM-4.7	Native-Python	15.3×	1.82×	48.2×
GLM-4.7	DEVS-Augmented	40.6×	1.41×	52.4×
GPT-5.2	Direct-Policy	66.0×	1.47×	52.6×
GPT-5.2	Native-Python	90.7×	2.46×	51.8×
GPT-5.2	DEVS-Augmented	**1.00×**	**1.00×**	**1.00×**

call creates the top-level structure before lower-level planning can branch, leaving a substantial serial prefix on the critical path. Synthesis has more parallel work: most of its time is spent generating the leaf atomic models that implement local state machines, and these calls can run concurrently once their interfaces have been fixed. App. E.4 reports the timing breakdown and variability.

5.5 Exploratory Downstream Planning

We examine whether generated simulators can support replenishment-policy generation in three supply-chain scenarios. We compare Direct-Policy, which generates policies without execution feedback; Native-Python, which can execute Python programs during policy development; and DEVS-Augmented, which uses a simulator generated by DEVS-GEN in the coding-agent workflow. All final policies are evaluated using the same Stockpyl-based evaluator. Table 2 shows that DEVS-Augmented yields lower replenishment costs than Direct-Policy in all settings. It also outperforms Native-Python in all scenarios with GPT-5.2, and achieves lower costs in one scenario with GLM-4.7. These results suggest that generated simulators can improve replenishment decisions over direct policy generation, while their advantage over agent-built supply-chain approximations depends on the backbone and scenario. App. K details the scenarios, evaluation protocol, and analysis.

AI USE STATEMENT

During benchmark construction, generative AI tools assisted with initial drafts of benchmark materials and with additional independent checks that complemented human review. Generative AI tools also assisted with manuscript drafting and revision, literature search and reference verification, figure preparation, and LaTeX formatting. Generative AI tools were not used to generate the reported experimental measurements. All AI-assisted outputs were reviewed by the authors, who take responsibility for the final content, claims, and artifacts.

REPRODUCIBILITY STATEMENT

Sections 3, 4, and 5 describe the generation procedure, evaluation protocol, and experimental setup. Appendices G, H, and J provide the agent prompts, benchmark scenario descriptions, and checker construction details.

Code, benchmark specifications, generated programs, and evaluation results are available at `https://anonymous.4open.science/r/devs_gen_code_iclr2027/`. A demo website presents the end-to-end Strategic Airlift workflow, including its visualization, together with two simulator-assisted planning examples at `https://demo1.devs-demo.workers.dev/`. Visitor analytics, per-request logging, and any other trackers are all disabled for the anonymous demo.

REFERENCES

Hyungjoo Chae, Namyoung Kim, Kai Tzu-iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a00548031e4647b13042c97c922fadf1-Abstract-Conference.html.

Jialu Chen, Baokun Hu, Wanwan Diao, and Yuqian Huang. Automatic generation of sysml requirement models based on chinese natural language requirements. In *Proceedings of the 2022 6th International Conference on Electronic Information Technology and Computer Engineering*, EITCE '22, pp. 242–248, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450397148. doi: 10.1145/3573428.3573470. URL https://doi.org/10.1145/3573428.3573470.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. *CoRR*, abs/2002.12543, 2020. URL https://arxiv.org/abs/2002.12543.

Michael Chui, Mark Collins, and Mark Patel. The Internet of Things: Catching up to an accelerating opportunity. Technical report, McKinsey Global Institute, 2021. URL https://www.mckinsey.com/capabilities/tech-and-ai/our-insights/iot-value-set-to-accelerate-through-2030-where-and-how-to-capture-it. Accessed: 2026-05-05.

Paulo Salem da Silva and Ana C. V. de Melo. On-the-fly verification of discrete event simulations by means of simulation purposes: Extended version. *Simulation*, 89(8):977–1008, 2013. doi: 10.1177/0037549713490439.

Eelco de Jong, Bhavna Lalla-Sewgoolam, and Gregory Vainberg. Unlocking the full power of automation in industrials. McKinsey & Company, 2019. URL https://www.mckinsey.com/industries/electric-power-and-natural-gas/our-insights/unlocking-the-full-power-of-automation-in-industrials. Accessed: 2026-05-05.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents, 2025. URL https://arxiv.org/abs/2512.12730. arXiv:2512.12730.

Jichen Feng, Yifan Zhang, Chenggong Zhang, Yifu Lu, Shilong Liu, and Mengdi Wang. Web world models. *arXiv preprint arXiv:2512.23676*, 2025. doi: 10.48550/arXiv.2512.23676. URL https://arxiv.org/abs/2512.23676.

Soroosh Gholami and Hessam S. Sarjoughian. Unified property evaluations of constrained-devs models for simulation and model checking. In *Proceedings of the 2021 Annual Modeling and Simulation Conference (ANNSIM)*, pp. 1–12. IEEE, 2021. doi: 10.23919/ANNSIM52504.2021.9552138.

Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, Huan Sun, and Yu Su. Is your llm secretly a world model of the internet? model-based planning for web agents, 2025. URL https://arxiv.org/abs/2411.06559.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, volume 31, pp. 2450–2462. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/hash/2de5d16682c3c35007e4e92982f1a2ba-Abstract.html.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.507. URL https://aclanthology.org/2023.emnlp-main.507/.

Kevin Henares, José L. Risco-Martín, José L. Ayala, and Román Hermida. Unit testing platform to verify DEVS models. In *Proceedings of the 2020 Summer Simulation Conference*, pp. 1–11. Society for Modeling & Simulation International, 2020.

Mengkang Hu, Tianxing Chen, Yude Zou, Yuheng Lei, Qiguang Chen, Ming Li, Yao Mu, Hongyuan Zhang, Wenqi Shao, and Ping Luo. Text2world: Benchmarking large language models for symbolic world model generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 26043–26066, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.1337. URL https://aclanthology.org/2025.findings-acl.1337/.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Gyuhong Lee and Su Man Nam. An automated verification framework for devs-coupled models to enhance efficient modeling and simulation. *Processes*, 13(5):1327, 2025. doi: 10.3390/pr13051327.

Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. doi: 10.1016/j.jlap.2008.08.004.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17160–17176, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.839. URL https://aclanthology.org/2025.acl-long.839/.

Xiaobo Li, Hans Vangheluwe, Yonglin Lei, Hongyan Song, and Weiping Wang. A testing framework for DEVS formalism implementations. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (TMS-DEVS '11)*, pp. 183–188. Society for Computer Simulation International, 2011.

Yixia Li, Hongru Wang, Jiahao Qiu, Zhenfei Yin, Dongdong Zhang, Cheng Qian, Zeping Li, Xiaoteng Ma, Guanhua Chen, and Heng Ji. From word to world: Can large language models be implicit text-based world models? In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8084–8111, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.366. URL https://aclanthology.org/2026.acl-long.366/.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=pPjZIOuQuF.

Sadegh Mahdavi, Raquel Aoki, Keyi Tang, and Yanshuai Cao. Leveraging environment interaction for automated pddl translation and planning with large language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 38960–39008. Curran Associates, Inc., 2024. doi: 10.52202/079017-1230. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/44af065477781e7f8a8589b14a62c489-Abstract-Conference.html.

Matthew B. McLaughlin and Hessam S. Sarjoughian. Devs-scripting: A black-box test frame for devs models. In *Proceedings of the 2020 Winter Simulation Conference (WSC)*, pp. 2196–2207. IEEE, 2020. doi: 10.1109/WSC48552.2020.9384024.

Meta AI. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. Meta AI, April 2025. URL https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

Next Generation Technologies Research Team. Process mining software market size report, 2025–2030. Technical report, Grand View Research, 2025. URL https://www.grandviewresearch.com/industry-analysis/process-mining-software-market-report. Published March 2025; accessed 2026-09-15.

OpenAI. Update to GPT-5 system card: GPT-5.2. Technical report, OpenAI, December 2025. URL https://cdn.openai.com/pdf/3a4153c8-c748-4b71-8e31-aecbde944f8d/oai_5_2_system-card.pdf.

James Oswald, Kavitha Srinivas, Harsha Kokel, Junkyu Lee, Michael Katz, and Shirin Sohrabi. Large language models as planning domain generators. *Proceedings of the International Conference on Automated Planning and Scheduling*, 34(1):423–431, 2024. doi: 10.1609/icaps.v34i1.31502. URL https://ojs.aaai.org/index.php/ICAPS/article/view/31502.

Jack Parker-Holder and Shlomi Fruchter. Genie 3: A new frontier for world models. Google DeepMind, 2025. URL https://deepmind.google/blog/genie-3-a-new-frontier-for-world-models/.

José L. Risco-Martín, Saurabh Mittal, Kevin Henares, Román Cardenas, and Patricia Arroba. xDEVS: A toolkit for interoperable modeling and simulation of formal discrete event systems. *Software: Practice and Experience*, 53(3):748–789, March 2023. doi: 10.1002/spe.3168.

Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pp. 1–38. Springer, 2008. doi: 10.1007/978-3-540-78917-8_1.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang H. Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. The openhands software agent sdk: A composable and extensible foundation for production agents. In *Proceedings of Machine Learning and Systems*, volume 8, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/8ae9cf363ea625161f885b798c1f1f78-Abstract-Conference.html.

Yuan Wang, Ning Ge, Jiangxi Liu, Zhilong Cao, Zheping Chen, and Chunming Hu. Generating sysml behavior models via large language models: an empirical study. In *Proceedings of the 16th International Conference on Internetware*, pp. 366–377. Association for Computing Machinery, 2025. doi: 10.1145/3755881.3755926.

Philipp Wu, Alejandro Escontrela, Danijar Hafner, Pieter Abbeel, and Ken Goldberg. Daydreamer: World models for physical robot learning. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 2226–2240. PMLR, 2023. URL https://proceedings.mlr.press/v205/wu23c.html.

Jiannan Xiang, Tianhua Tao, Yi Gu, Tianmin Shu, Zirui Wang, Zichao Yang, and Zhiting Hu. Language models meet world models: Embodied experiences enhance language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 75392–75412. Curran Associates, Inc., 2023. doi: 10.52202/075280-3295. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/ee6630dcbcff857026e474fc857aa9f0-Abstract-Conference.html.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. doi: 10.48550/arXiv.2505.09388. URL https://arxiv.org/abs/2505.09388.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Z.AI. GLM-4.7. Z.AI Developer Documentation, December 2025. URL https://docs.z.ai/guides/llm/glm-4.7. Technical documentation.

Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2 edition, 2000.

Bernard P. Zeigler, James J. Nutaro, and Chungman Seo. Combining DEVS and model-checking: concepts and tools for integrating simulation and analysis. *International Journal of Simulation and Process Modelling*, 12(1):2–15, 2017. doi: 10.1504/IJSPM.2017.082781.

Tianyi Zhang, Li Zhang, Zhaoyi Hou, Ziyu Wang, Yuling Gu, Peter Clark, Chris Callison-Burch, and Niket Tandon. PROC2PDDL: Open-domain planning representations from texts. In *Proceedings of the 2nd Workshop on Natural Language Reasoning and Structured Explanations (@ACL 2024)*, pp. 13–24, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.nlrse-1.2/.

A Related Work

A.1 World Models: Explicit and Implicit Approaches

World models have long served as predictive mechanisms to support planning and reinforcement learning, enabling agents to reason about the consequences of actions before execution and to generate simulated experience for improved learning efficiency (Ha & Schmidhuber, 2018). Broadly, existing work on world modeling can be categorized into two complementary approaches: *explicit world models*, in which environment dynamics are implemented inside executable simulators, and *implicit world models*, in which dynamics are captured by learned models, often neural networks.

Several works construct world models as human-designed simulators with well-defined state representations and transition dynamics. For example, prior work uses a robotic simulator as a world model and demonstrates that interaction with such a model can enhance the performance of language models on downstream tasks (Xiang et al., 2023). Similarly, Feng et al. (2025) present *Web World Models*, a hybrid approach in which web code implements world state and dynamics while LLMs generate context, narratives, and high-level decisions. These approaches emphasize executability and control over environment dynamics. In parallel, recent work has explored whether environment dynamics can be modeled implicitly by large neural networks, particularly LLMs. In this paradigm, the world model is not represented as a standalone simulator; instead, environment dynamics are encoded in the model's parameters and accessed through prediction during inference. LLMs are prompted or trained to predict future observations or outcomes conditioned on past events, and these predictions are used to guide action selection or planning (Hao et al., 2023; Gu et al., 2025). Learned world models have been used for language-based web planning (Gu et al., 2025; Chae et al., 2025), interactive visual environments (Parker-Holder & Fruchter, 2025), and physical robot learning from camera observations and sparse rewards (Wu et al., 2023).

These two lines of work reflect different trade-offs in how world dynamics are represented and accessed. Explicit world models provide executable, transparent representations of environment dynamics but typically require substantial manual effort to design and adapt, while implicit neural world models offer greater flexibility and scalability by leveraging learned representations, at the cost of reduced explicitness and long-horizon rollout consistency.

A.2 Symbolic and Programmatic World Model Generation

A closely related line of work focuses on generating structured representations of environment dynamics from natural language, with the goal of improving interpretability and enabling rule-based or formal verification. In the context of symbolic world or domain models, several approaches translate natural-language descriptions into executable representations such as the Planning Domain Definition Language (PDDL), using plan execution behavior or task outcomes as evaluation signals (Hu et al., 2025; Zhang et al., 2024; Oswald et al., 2024). Complementary efforts reduce reliance on expert supervision by incorporating environment interaction to iteratively refine generated planning domains (Mahdavi et al., 2024).

Beyond planning-oriented representations, related work has explored generating system-level models from natural-language requirements in software and systems engineering. In particular, several studies synthesize Systems Modeling Language (SysML) artifacts, such as structural diagrams or behavioral models, and evaluate them using consistency checks, requirement coverage, or simulation-based criteria (Chen et al., 2023; Wang et al., 2025). These approaches emphasize requirements traceability, design validation, and alignment with established engineering workflows, but typically focus on static structure or high-level behavioral descriptions rather than executable environment dynamics.

Our work aligns with these efforts in seeking explicit, programmatic representations of environment dynamics from natural language, but differs in its choice of modeling formalism and evaluation focus. Related formalisms can encode event timing, resource constraints, and multi-entity interaction, but they organize them around different primary abstractions. Markov decision process (MDP) models foreground state, action, and reward for sequential decision making, while PDDL-based representations emphasize symbolic action preconditions, effects, and goal satisfaction (Hu et al., 2025; Zhang et al., 2024; Oswald et al., 2024; Mahdavi et al., 2024). In contrast, we focus on simulators with explicit temporal semantics. This is important for environments with asynchronous external

events, overlapping activities, and contention for shared resources, such as jobs waiting for a server or robots competing for chargers. These patterns are natural in discrete-event models but require extra encoding in MDP or PDDL formulations. We therefore adopt a discrete-event perspective centered on event timing, component interaction, and observable trajectories. Relative to SysML-based approaches, which primarily support high-level system specification and design reasoning, DEVS offers precise simulation semantics and inherent executability, producing structured event traces that enable black-box, trace-based evaluation under controlled scenarios. Together, these properties position DEVS as a natural foundation for generating and evaluating discrete-event world models from natural-language descriptions.

A.3 LLM-Based Code Generation and Multi-Step Development

Recent research has extensively explored the capabilities of large language models and code agents for natural-language-driven software implementation, fixing issues in existing code repositories, and generating multi-file projects. Evaluations typically focus on unit-test-based functional correctness, issue-fixing success rates, or repository-level benchmarks. Examples include unit-test-driven functional correctness evaluation (Chen et al., 2021), repository-repair benchmarks based on real GitHub issues (Jimenez et al., 2024), and repository-level multi-file evaluation settings (Liu et al., 2024; Ding et al., 2025).

The problem addressed in this paper likewise involves multi-step code development: coordinating across multiple steps and files. Existing work often summarizes complex processes with automatable outcome metrics (e.g., test pass/fail or task completion rate) and evaluates systems on multi-step interactive software-engineering tasks or repository-level feature-implementation settings (Yang et al., 2024; Ding et al., 2025; Li et al., 2025).

Our evaluator is implemented as an automated black-box test suite, but its defining criterion is conformance to a scenario specification. The generated program is evaluated as an executable world model through its external interface: specified inputs are supplied, and the resulting event traces are checked for operational, temporal, causal, and aggregate requirements. A local checker may resemble a unit test, while the complete suite also contains cross-component and collection-level checks. This specification-driven evaluation is conceptually related to conformance testing (Tretmans, 2008) and to runtime verification through execution traces (Leucker & Schallhart, 2009). The distinction from many code-generation benchmarks is therefore not that they use tests and we do not, but that our test object is a complete executable world model and our oracle is a scenario-level trace specification.

A.4 Verification, Validation, and Conformance for Discrete-Event Models

Verification and validation (V&V) of discrete-event models and DEVS models have a long research history, including static consistency checks, simulation-based verification, and formal analysis of model composition. These approaches provide systematic tools for detecting specification violations. For instance, structural consistency verification can validate the structural definitions of coupled models prior to simulation (Lee & Nam, 2025); behavioral-level verification is often instantiated as simulation-driven unit tests or semantic-consistency tests (Henares et al., 2020; Li et al., 2011); Metamorphic testing addresses the oracle problem by checking relations between executions under transformed inputs (Chen et al., 2020); and work that integrates DEVS with model checking and applies mixed V&V in hierarchical/multi-resolution compositional modeling illustrates a path that combines formal analysis with compositional verification (Zeigler et al., 2017; Gholami & Sarjoughian, 2021).

In line with this tradition, execution trajectories can serve as a direct basis for systematic evaluation: given input drivers and observation schemes, the runtime traces of the system under test can be compared against specified properties to determine whether the expected behavior is satisfied. This perspective is commonly framed as runtime verification (RV) in the broader software and systems verification literature, where specifications are checked online or offline against event/state sequences (Leucker & Schallhart, 2009). In discrete-event simulation, similar trace-based checking can be achieved by explicitly modeling properties and performing on-demand checks during simulation (da Silva & de Melo, 2013).

This paper similarly adopts trace-based system evaluation, but under a distinct context: the models are automatically generated by LLM-driven generators, resulting in highly diverse implementation forms. Consequently, the evaluation framework must center on interfaces and observable behaviors to accommodate diverse implementations, and provide actionable localization signals when specification violations are detected. Furthermore, evaluation should target behavioral conformance rather than code-level equivalence to a single reference implementation. These requirements connect to black-box testing frameworks and their diagnostic capabilities for failing cases (McLaughlin & Sarjoughian, 2020).

B Glossary of Key Terms

Term	Definition
Core Concepts	
Executable World Model ($\mathcal{M}$)	The generated simulator project: an executable model that accepts initialization input $\mathcal{I}$ and optional external input $\mathcal{J}$, and emits a structured event trace $\mathcal{T}$.
Event Trace ($\mathcal{T}$)	A chronological sequence of discrete-event records emitted during execution. For example, each record captures the timestamp, emitting entity, event type, and payload, and the trace serves as the observable behavior for specification-driven evaluation.
Environment Specification (`Spec`)	The natural-language description supplied to the generation method. For evaluation, each benchmark `Spec` is partitioned into operational requirements ($\texttt{Spec}_{ope}$) and behavioral requirements ($\texttt{Spec}_{beh}$).
DEVS Formalism	
DEVS Model	The formal model at the core of DEVS-based $\mathcal{M}$, constructed from atomic and coupled components under the Discrete Event System Specification formalism. It provides the hierarchy, local dynamics, and event-routing structure realized by the simulator project.
Atomic Model	A DEVS component that maintains explicit internal state and dictates state-transition and timing logic (i.e., how the state evolves over time or in response to events). It contains no sub-components.
Coupled Model	A structural DEVS component that encapsulates multiple sub-components (which can be Atomic or other Coupled models). It contains no behavioral logic of its own, but defines the exact event routing (coupling) between the ports of its children and its own external interface.
Port	An explicit, typed communication interface through which DEVS components exchange discrete events.
Pipeline Artifacts	
`PlanTree`	The hierarchical intermediate representation produced by structural planning: a `Skeleton` with a `ModelPlan` for every node. It records the component hierarchy, interfaces, behavior, and coupling rules used in behavioral synthesis.
`ModelPlan`	The plan stored at a `PlanTree` node, specifying the component's type, constructor arguments, ports, responsibilities for external input and output, and behavior or coupling rules.
`Skeleton`	The initial, high-level blueprint inferred from the natural-language specification. It records module names, parent-child relationships, and brief responsibilities before node contracts and coupling rules are added.

C Formal Definition of Parallel DEVS

In this section, we provide the formal mathematical definitions of the Parallel DEVS (PDEVS) formalism (Zeigler et al., 2000; Risco-Martín et al., 2023) used to construct our discrete-event world models.

Atomic Models. An atomic model encapsulates the localized state and behavioral logic of an entity. We define an atomic model $\mathcal{M}_e$ for corresponding entity $e \in \mathcal{E}$ as the 8-tuple:

$$\mathcal{M}_e = \left(X_e, Y_e, \mathcal{S}_e, \delta^e_{\text{int}}, \delta^e_{\text{ext}}, \delta^e_{\text{con}}, \lambda^e, t^e_a\right)$$

where:

- Let P^{in}_e and P^{out}_e be the finite sets of input and output port names, and let $V^{\text{in}}_{e,p}$ and $V^{\text{out}}_{e,p}$ be their declared payload spaces. Then $X_e = \bigcup_{p \in P^{\text{in}}_e} (\{p\} \times V^{\text{in}}_{e,p})$ and $Y_e = \bigcup_{p \in P^{\text{out}}_e} (\{p\} \times V^{\text{out}}_{e,p})$ are the admissible port-labeled input and output messages.
- $\mathcal{S}_e$ is the local state space.
- $\delta^e_{\text{int}} : \mathcal{S}_e \to \mathcal{S}_e$ is the internal transition function.
- $\delta^e_{\text{ext}} : \mathcal{S}_e \times \mathbb{R}_{\geq 0} \times \mathcal{B}(X_e) \to \mathcal{S}_e$ is the external transition function. The real-valued argument denotes the elapsed simulation time since the last transition, and $\mathcal{B}(\cdot)$ denotes bags (multisets) of messages to handle concurrent inputs.
- $\delta^e_{\text{con}} : \mathcal{S}_e \times \mathcal{B}(X_e) \to \mathcal{S}_e$ is the confluent transition function, utilized to resolve conflicts when an external input arrives exactly at the scheduled internal transition time.
- $\lambda^e : \mathcal{S}_e \to \mathcal{B}(Y_e)$ is the output function.
- $t^e_a : \mathcal{S}_e \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is the time-advance function, determining the lifespan of the current state.

Operationally, an atomic model remains in state $s \in \mathcal{S}_e$ for $t^e_a(s)$ units of simulated time. Upon expiration, it emits outputs via λ^e and performs an internal transition via δ^e_{int}. If an input arrives before the scheduled internal transition, the model applies δ^e_{ext} using the elapsed time. If an input arrives exactly at the internal transition time, δ^e_{con} dictates the new state.

Coupled Models. Coupled models form the hierarchical structure of the system by composing sub-components and defining their routing topology. A coupled model $\mathcal{M}_c$ is defined as the 7-tuple:

$$\mathcal{M}_c = \left(X_c, Y_c, \mathcal{D}_c, \{\mathcal{M}_d\}_{d \in \mathcal{D}_c}, \text{EIC}_c, \text{EOC}_c, \text{IC}_c\right)$$

where:

- X_c and Y_c are the coupled model's admissible port-labeled messages, defined from P^{in}_c, P^{out}_c, and their payload spaces in the same manner as for an atomic model.
- $\mathcal{D}_c$ is an index set of sub-components, where each $\mathcal{M}_d$ is either an atomic or a coupled model.
- Define the child input and output endpoint sets as $\text{In}^D_c = \{(d,p) : d \in \mathcal{D}_c,\ p \in P^{\text{in}}_d\}$ and $\text{Out}^D_c = \{(d,p) : d \in \mathcal{D}_c,\ p \in P^{\text{out}}_d\}$.
- $\text{EIC}_c \subseteq P^{\text{in}}_c \times \text{In}^D_c$ is the External Input Coupling, which routes messages from the coupled model's external input ports to child input ports.
- $\text{EOC}_c \subseteq \text{Out}^D_c \times P^{\text{out}}_c$ is the External Output Coupling, which routes messages from child output ports to the coupled model's external output ports.
- $\text{IC}_c \subseteq \text{Out}^D_c \times \text{In}^D_c$ is the Internal Coupling, which governs direct communication among child components. Every coupling is required to connect payload-compatible source and destination ports.

This recursive structure yields the DEVS model hierarchy within the DEVS-based simulator $\mathcal{M}$. When the planned DEVS model exposes boundary ports, EIC and EOC route events through that root. The benchmark's command-line arguments are instead parsed by the generated entry script and passed through constructors, while standard input, standard output, and file operations are assigned separately as external-I/O responsibilities.

D Detailed Generation Algorithms

Algorithm 1 summarizes the pipeline described in Section 3; Algorithms 2 and 3 give the detailed planning and synthesis procedures.

Algorithm 1 DEVS-Gen Pseudocode

Input: specification `Spec`
Output: executable world model $\mathcal{M}$
`PlanTree` $\leftarrow$ PLAN(`Spec`) *// See Alg. 2*
$\mathcal{M} \leftarrow$ SYNTHESIZE(`PlanTree`, `Spec`) *// See Alg. 3*

Algorithm 2 Stage 1: Structural Planning

function PLAN(`Spec`)
    // Step 1: Global skeleton inference
    $\mathcal{Q} \leftarrow$ BUILDREQUIREMENTINDEX(`Spec`) *// App. G.1*
    `Skeleton` $\leftarrow$ INFERGLOBALSKELETON(`Spec`, $\mathcal{Q}$) *// App. G.1*
    `PlanTree`$\leftarrow$ INITIALIZETREEHIERARCHY(`Skeleton`) *// deterministic*

    // Step 2: Architecture refinement (Top-Down Parallel)
    $draft \leftarrow$ DRAFTROOTPLAN(`Spec`, $\mathcal{Q}$, `PlanTree`.$root$) *// App. G.2*
    $revised \leftarrow$ REWRITEROOTPLAN(`Spec`, $\mathcal{Q}$, `Skeleton`, $draft$) *// App. G.2*
    (`PlanTree`.$root$.`ModelPlan`, `PlanTree`.$root.children$) $\leftarrow revised$
    for all $child \in$ `PlanTree`.$root.children$ **in parallel do**
        REFINENODE(`Spec`, $\mathcal{Q}$, $child$)
    end for

    return `PlanTree`
end function
function REFINENODE(`Spec`, $\mathcal{Q}$, v)
    if v is Coupled Model **then**
        (v.`ModelPlan`, $v.children$) $\leftarrow$ REFINECOUPLEDPLAN(`Spec`, $\mathcal{Q}$, v) *// App. G.2*
        // Refine children in parallel once their contracts are set
        for all $child \in v.children$ **in parallel do**
            REFINENODE(`Spec`, $\mathcal{Q}$, $child$)
        end for
    else if v is Atomic Model **then**
        v.`ModelPlan` $\leftarrow$ REFINEATOMICPLAN(`Spec`, $\mathcal{Q}$, v) *// App. G.2*
    end if
end function

Algorithm 3 STAGE 2: BEHAVIORAL SYNTHESIS

function SYNTHESIZE(`PlanTree`, `Spec`)
  $S \leftarrow$ SYNTHESIZESUBTREE(`PlanTree`.$root$)
  $entry \leftarrow$ GENERATEENTRYPOINT($S.root$, `Spec`)
  $\mathcal{M} \leftarrow$ SIMULATORPROJECT($S.root, S.modules, entry$)
  return $\mathcal{M}$
end function

function SYNTHESIZESUBTREE(node)
  if $node$ is an Atomic Model **then**
    $root \leftarrow$ GENERATEATOMIC($node$.`ModelPlan`) *// App. G.3*
    return SUBTREE($root, \{root\}$)
  end if
  for all $child \in node.children$ **in parallel do**
    $S_{child} \leftarrow$ SYNTHESIZESUBTREE($child$)
    $excerpt_{child} \leftarrow$ EXTRACTCONSTRUCTIONEXCERPT($S_{child}.root$)
    `ChildContexts`[$child$] $\leftarrow$ ($child$.`ModelPlan`, $excerpt_{child}$)
    `ChildSubtrees`[$child$] $\leftarrow S_{child}$
  end for
  $root \leftarrow$ GENERATECOUPLED($node$.`ModelPlan`, `ChildContexts`) *// App. G.3*
  $modules \leftarrow \{root\} \cup \bigcup_{S \in \texttt{ChildSubtrees}} S.modules$
  return SUBTREE($root, modules$)
end function

E ADDITIONAL EXPERIMENTAL RESULTS AND ANALYSIS

E.1 GENERATION COMPLETION, FULL OPERATIONAL SUCCESS, AND CONDITIONAL SCORES

Table 4 reports generation completion, full operational success, and mean scores under each condition. A completed generation terminates normally and supplies the expected entry point. Full operational success requires every operational test case for a generated project to be valid. The final column averages $\mathrm{Score}_{\mathrm{beh}}$ over the fully operational generations of each method. These generations have $\mathrm{Score}_{\mathrm{ope}} = 1$, so their behavioral scores incur no operational-gate penalty.

Table 4: Generation counts and mean scores among completed or fully operational generations. Each method has 120 scheduled generations.

	Generation counts		Among completed		Among fully operational
Method	**Completed**	**Fully operational**	$\mathrm{Score}_{\mathrm{ope}}$	$\mathrm{Score}_{\mathrm{beh}}$	$\mathrm{Score}_{\mathrm{beh}}$
DEVS-GEN	115 / 120	74 / 120	0.698	0.601	0.860
OpenHands-Lite	97 / 120	59 / 120	0.684	0.602	0.874
SWE-Agent-Lite	103 / 120	57 / 120	0.607	0.525	0.863
OpenHands	92 / 120	67 / 120	0.830	0.720	0.867
SWE-Agent	95 / 120	69 / 120	0.771	0.676	0.878

DEVS-GEN completes 115 of 120 generations and produces 74 fully operational simulators, the highest counts among the five methods. Among completed generations, its mean behavioral score is close to OpenHands-Lite, higher than SWE-Agent-Lite, and lower than the two execution-enabled baselines. These values complement the end-to-end averages in Table 1 by separating output availability, validity across the full test suite, and the mean quality of completed simulators.

Among fully operational generations, the five mean behavioral scores range from 0.860 to 0.878. Each method's conditional mean uses its own successful subset, which may differ in backbone and scenario composition; these values diagnose behavior after operational success rather than compare methods on a matched subset.

E.2 EARLY BASELINE RESULTS

We analyze early baseline results in the resource-versus-outcome curves in Figure 3, using the cumulative outcome defined in the main text.

For each baseline, outcome, and resource measure, the *resource-dominance threshold* is the smallest budget from which onward DEVS-GEN's cumulative outcome remains no lower than the baseline's. A *baseline advantage contributor* is a baseline generation with a positive score and resource cost at or below its corresponding threshold. For brevity, we report the following analysis only for behavioral conformance. The same qualitative patterns hold for completion and operational validity.

We then examine the three repetitions for each baseline, backbone, and scenario. On the time axis, 71 such settings yield at least one baseline advantage contributor: contributors occur in all three repetitions for 22 settings (31.0%), in exactly one repetition for 30 (42.3%), and in exactly two repetitions for the remaining 19. On the token axis, 49 settings yield contributors, with the corresponding counts being 13 (26.5%), 24 (49.0%), and 12. Thus, a low-cost positive result in one run often fails to recur under the same settings, indicating substantial run-to-run variability.

For each resource axis, we sum baseline behavioral scores by scenario, separately for generations at or below their respective thresholds and those above them. HouseHeating, SEIRD, and StoreCashier are the three empirically easiest scenarios, with the highest mean behavioral scores across all methods, backbones, and repetitions. Together, they account for 56.3% of the total score at or below the time thresholds, compared with 32.6% above them; the corresponding token-based shares are 50.7% and 39.7%.

E.3 STATISTICAL COMPARISONS

Table 5 compares DEVS-GEN with each baseline using a two-sided permutation test stratified by the combinations of backbone and scenario. Within each combination, the six generation-level scores (three from each method) are randomly reassigned into two groups of three. We estimate the p-value from 200,000 permutations as the proportion whose absolute mean difference is at least as large as the observed difference.

DEVS-GEN achieves higher mean behavioral scores than all four baselines. The improvements over OpenHands-Lite and SWE-Agent-Lite are statistically significant, with mean differences of 0.089 and 0.125, respectively. After averaging over backbones and repetitions, DEVS-GEN scores higher in eight of ten scenarios against OpenHands-Lite and all ten against SWE-Agent-Lite. The smaller mean improvements over the Standard agents do not reach statistical significance.

Table 5: Comparisons of $\text{Score}_{\text{beh}}$ using all generation-level results. The p-values are obtained from the stratified permutation tests. The "Scenarios favoring ours" column counts scenarios in which DEVS-GEN has the higher mean after averaging over backbones and generations. Positive mean differences favor DEVS-GEN.

Baseline	**Mean difference**	**Scenarios favoring ours**	**p-value**
OpenHands-Lite	$+0.089$	$8/10$	0.0038
SWE-Agent-Lite	$+0.125$	$10/10$	0.0002
OpenHands	$+0.024$	$6/10$	0.408
SWE-Agent	$+0.041$	$6/10$	0.226

E.4 PARALLEL EXECUTION

The staged pipeline separates structural planning from component implementation. Independent calls can therefore run in parallel after their interfaces are fixed. For a balanced hierarchy, this can shorten the critical path relative to serial component generation, although the realized speedup depends on hierarchy depth, API latency, and orchestration overhead. We measure this effect on GPT-5.2 by comparing serial and parallel execution in DEVS-GEN over the benchmark scenarios.

Figure 4 reports the wall-clock time of the structural planning and behavioral synthesis stages under the two execution modes. The error bars summarize the timing distribution across independent

generations on the ten scenarios. Parallel execution reduces mean end-to-end generation time from 274.1 s to 182.3 s on GPT-5.2, a $1.50\times$ speedup. Structural planning decreases from 183.8 s to 132.2 s ($1.39\times$), while behavioral synthesis decreases from 71.8 s to 31.7 s ($2.27\times$). Planning is only partly parallelizable because its longest call creates the top-level structure before lower-level planning can branch, leaving a substantial serial prefix on the critical path. Synthesis has more parallel work: most of its time is spent generating the leaf atomic models that implement local state machines, and these calls can run concurrently once their interfaces have been fixed.

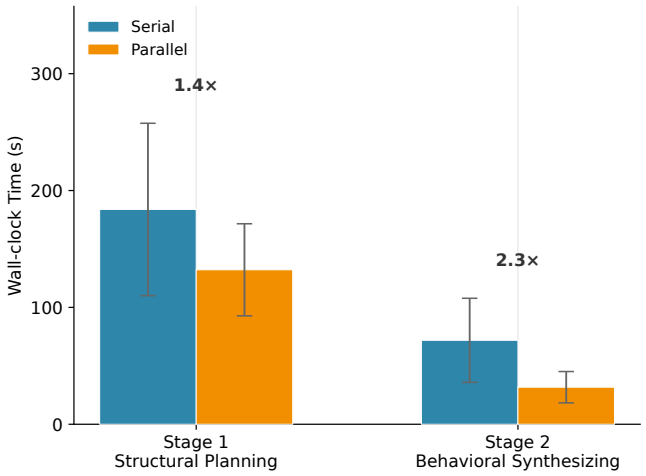

Figure 4: Ablation study on generation latency using GPT-5.2. The chart compares the wall-clock time required for planning and synthesis under serial and parallel execution. Bars show means and error bars indicate $\pm$ sample standard deviation across 30 generations (three per scenario). Parallel execution gives about a $1.39\times$ speedup in planning and a $2.27\times$ speedup in synthesis. The total wall-clock time is about 66.5% of serial execution.

E.5 Framework Constraints and Cognitive Load

We compare three execution-disabled settings: single, which generates a free-form monolithic simulator; single-xdevs, which directly generates an `xdevs.py`-based simulator without structural planning; and DEVS-GEN, which uses the same implementation requirement together with structural planning and behavioral synthesis. Every setting has three repetitions per scenario. Here, the term "cognitive load" refers to two demands of building DEVS models with `xdevs.py`: (i) DEVS-style component and interface discipline; and (ii) the concrete `xdevs.py` API.

Table 1 shows that single has higher operational and behavioral scores than single-xdevs for every backbone. The proportional decrease in both scores is larger for GLM-4.7, Qwen3-Coder, and Llama 4 Scout than for GPT-5.2. This pattern is consistent with the difficulty of handling global architecture and local formalism syntax at the same time.

The comparison between single-xdevs and DEVS-GEN holds the `xdevs.py` implementation requirement fixed and changes the generation procedure. DEVS-GEN has higher mean operational and behavioral scores for every backbone, indicating that hierarchical planning and component-wise synthesis improve reliability under this implementation requirement.

F Structured Intermediate Representations

The `ModelPlan` is the paper-level name for a node's planning contract. The excerpt below reflects the fields used by the evaluated implementation. An atomic node has no coupling rules; a coupled node stores one natural-language coupling rule per list item. Rules can describe fixed, repeated, indexed, conditional, EIC, IC, or EOC connections. The operational meaning of a rule is resolved against the declared child and boundary ports during code generation. External I/O is separate from DEVS ports.

```python
class GlobalPlanNode(BaseModel):
    name: str
    description: str
    children_names: list[str]
    related_requirement_ids: list[str]

class ProtocolSpec(BaseModel):
    initial_signal: str = "None"
    description: str = ""

class TypedEntity(BaseModel):
    name: str
    type: str
    structure: str = ""

class PortEntity(TypedEntity):
    protocol: ProtocolSpec

class ExternalIOStream(BaseModel):
    target: Literal["stdin", "stdout", "stderr", "file", "other"]
    content: str

class ModelSpecification(BaseModel):
    function: str
    external_io: list[ExternalIOStream]
    model_init_args: list[TypedEntity]
    input_ports: list[PortEntity]
    output_ports: list[PortEntity]
    implementation_example: str | None  # atomic example choice

class DetailedPlan(BaseModel):
    class_name: str
    model_type: Literal["atomic", "coupled"]
    specification: ModelSpecification
    coupling_rules: list[str]  # empty for atomic models

class SimpleDetailedPlan(BaseModel):
    class_name: str
    model_type: Literal["atomic", "coupled"]
    function: str
    external_io: list[ExternalIOStream]
    model_init_args: list[TypedEntity]
    input_ports: list[PortEntity]
    output_ports: list[PortEntity]
    related_requirement_ids: list[str]
```

Listing 1: Planning schema used in the evaluated implementation

`DetailedPlan` is the stored contract for a node. `SimpleDetailedPlan` is the child boundary declared by its parent and then refined without changing the inherited constructor, ports, or external-I/O allocation. Temporary response wrappers used during refinement are not part of the stored `ModelPlan`.

The requirement index $\mathcal{Q}$ records each extracted obligation, its source sections, and any inter-model flows. A requirement can remain relevant to several cooperating nodes through `related_requirement_ids`; the same index is available during planning and code generation.

G Agent Prompt Templates

Here we present the main prompt templates used for structural planning and component synthesis.

G.1 Requirement Extraction and Structural Planning

The first call converts the sectioned input into the requirement index $\mathcal{Q}$.

Requirement-index extraction template

```
## Role
You are preparing a loss-aware requirement ledger for a DEVS
    implementation.

## Source sections
{source_sections}

## Task
Extract the implementation obligations into a compact ledger. Do not
    classify requirements into categories. Behavior, IO, parameters,
    timing, schemas, and edge cases are all ordinary requirements in
    this ledger.

Each item must be self-contained. Preserve exact names, values,
    conditions, timing, schemas, exceptions, same-time ordering, and
    edge cases needed to implement it. A requirement that concerns
    several future modules remains one shared requirement; do not
    invent module assignments yet. Group statements only when they
    form one coherent obligation. Do not turn examples or background
    into new requirements.

For every cross-module message or handoff, keep the producer, intended
     receiver, payload/event meaning, explicit port names, and
    triggering condition together in the same self-contained item and
    in its `flows` list. Copy only explicitly stated facts into a flow
    ; leave an unspecified port/payload/trigger as an empty string. Do
     not record only the sending half of a flow.

For `source_section_ids`, cite whole section IDs from the input. Do
    not quote a small context window. Do not create IDs yourself;
    stable requirement IDs are assigned after extraction.

Return exactly one JSON object with a top-level `requirements` array.
    Each array item uses the keys `detail`, `source_section_ids`, and `
    flows`. Each flow uses `producer`, `consumer`, `producer_port`, `
    consumer_port`, `payload`, and `trigger`.

Condition-preservation example:
- Source text: "If dynamic input is required, read from stdin line by
    line."
- Correct detail: "If dynamic input is required, read from stdin line
    by line."
- Incorrect detail: "Read dynamic input from stdin line by line."

Multi-flow example:
- Source text: "A worker reports `worker_id` to the dispatcher; the
    dispatcher sends `{{worker_id, job_id}}` back to the selected
    worker."
- Correct result: one self-contained requirement with two flows,
    Worker -> Dispatcher carrying `worker_id`, and Dispatcher ->
    Worker carrying `{{worker_id, job_id}}`. Do not merge the two
    directions into one flow.

Response-shape example:
{{"requirements":[{{"detail":"The sender starts at time 0.","
    source_section_ids":["S01"],"flows":[]}}]}}
```

The structural-planning call receives both the raw specification and $\mathcal{Q}$.

Global-skeleton prompt template

```
## [Role]
You are a **DEVS System Architect**. Your task is to design the overall module hierarchy for a DEVS simulation system.

## [Input]
**System Name**: `{root_name}`
**Requirements**:
{requirements}

## [Requirement Ledger]
This is a navigation aid over the unabridged requirements above. Every ID must remain associated with at least one relevant module. It does not replace or override the original text.
{requirement_ledger}

## [Task]
Decompose the system into a hierarchical module structure. Return a list of modules.

## [Rules]
Make these boundary decisions before minimizing the module count:
- Command-line arguments are parsed by the generated `run.py` entry point and passed through the root constructor. Do not create an InputHandler atomic, configuration ports, or startup messages solely to transport CLI values.
- The generated runtime requires the root to be coupled. When the complete behavior is one coherent state machine, give the root exactly one atomic child; do not split CLI handling or final printing into extra children.
- If the request reads stdin, exactly one leaf description must explicitly say that it directly reads stdin. When stdin contains future event times, do not merge that reader into an atomic model that also schedules service, timeout, or other business events; use a small source atomic.
- For ordinary per-event JSONL, every leaf that produces a required record must say that it writes that record. Do not move all output to the stdin reader or invent a central writer unless aggregation is explicitly required.
- If stdout must be exactly one combined JSON document containing facts from several models, include one collector leaf. Producers send their structured facts to it through DEVS ports; only the collector writes stdout after the simulation. Reading the input does not by itself make a source the collector.

1. The first module MUST be the coupled root system `{root_name}` and must have at least one child. Use multiple children only for genuinely interacting state machines; otherwise use one coherent atomic child.
2. Every name mentioned in any `children_names` MUST appear as a `name` somewhere in the list.
3. Use hierarchical decomposition only when it clarifies the model. Prefer a shallow structure when a process can be implemented coherently by one atomic model or a small number of atomic models.
4. Keep descriptions SHORT (1-2 sentences). State what the module does, its key message handoffs, and any OS operation the module directly performs. Avoid the ambiguous phrase "emits an event": say either "sends a DEVS message to <module>", "writes a record to stdout", or explicitly say that it does both.
```

```
5. Module names should be valid Python class names (PascalCase, no spaces/special chars).
6. Atomic (leaf) modules should have `children_names: []`.
7. Do NOT over-decompose. 3-4 levels of hierarchy is usually sufficient, but the optimal number depends on the complexity of the system.
8. Except for the required root wrapper, do not introduce a coupled model only to satisfy a shape preference. If a subsystem has a single coherent state machine, make it atomic instead of splitting it into artificial controller/pool/interface modules.
9. DEVS Principles:
  - A system is hierarchical.
  - Atomic models have behavior but NO sub-components.
  - Coupled models have sub-components and routing (couplings) but NO behavior.
  - A child output may connect to a sibling input (IC) or to its current coupled boundary (EOC). A current boundary input may connect to a child input (EIC).
  - Mention required inter-module handoffs in descriptions; detailed planning will choose the exact port names.
10. If requirements specify multiplicity (e.g., count=N or count='arg:x'), the architecture MUST preserve that multiplicity at runtime. Represent the repeated entity as one atomic blueprint module whose parent instantiates it repeatedly; do not collapse all instances into one representative atomic model.
11. Requirement continuity: The root represents the complete system, so its `related_requirement_ids` should contain every supplied ledger ID. Put each ledger ID in `related_requirement_ids` for every module whose planning or implementation needs that requirement. This is an inclusive relevance association, not an assignment of OS operations: one ID may appear on several cooperating modules. Every supplied ID must appear at least once, and no new IDs may be invented.

## [DEVS messages and OS IO]
Keep these mechanisms separate:
1. `stdin`, `stdout`, `stderr`, files, and services are OS interactions performed directly by a leaf atomic model. They are recorded later in `external_io`.
2. Data needed by another DEVS model travels through a DEVS output port and a coupling. stdout and files are never parent ports or EOC targets.

## [IMPORTANT: KEEP IT SIMPLE]
- **Minimize the number of modules**: Only create modules that are truly necessary.
- **Prefer shallow hierarchies**: A flat structure with fewer levels is better than a deep one.
- **Each module should have a clear, distinct responsibility**: Avoid overlapping functions.
- **Atomic modules should be self-contained**: Each should implement a complete, coherent piece of functionality.
- **Keep tightly coupled state together**: Queues, resources, and timeout rules that must be updated in one consistent event order should stay in the same atomic model unless the requirements clearly describe separate interacting subsystems.

## [Field Guidance]
- `name`: Valid Python identifier, PascalCase. Example: "EventSource", "CoreProcessor". One name per module, unique across the entire list.
- `description`: 1-2 sentences stating what the module does, the messages it exchanges with other modules, and any OS operation it
```

```
    directly performs. Name the DEVS recipient of a message and
    distinguish that message from a stdout record. Do not invent exact
     port names at this stage.
- `children_names`: List of child module names. Empty list `[]` for
    atomic (leaf) modules.
- `related_requirement_ids`: Ledger IDs relevant to this module. Use
    `[]` only when no ledger is supplied or no listed requirement
    concerns the module.

## [Example]
For a system "SmartFactory" with requirements "read orders from stdin,
     process components, and write event records to stdout":
- modules (illustrative IDs):
  - name: "SmartFactory", description: "Top-level coupled model.
    Routes orders through assembly and quality checking.",
    children_names: ["OrderSource", "AssemblyRobot", "QualityChecker
    "], related_requirement_ids: ["R001", "R002", "R003"]
  - name: "OrderSource", description: "The only atomic model that
    reads stdin. It sends parsed orders to AssemblyRobot and writes
    the message records it produces to stdout.", children_names: [],
    related_requirement_ids: ["R001", "R003"]
  - name: "AssemblyRobot", description: "Atomic model. It assembles
    received orders, sends finished parts to QualityChecker, and
    writes its own state and sent-message records to stdout.",
    children_names: [], related_requirement_ids: ["R002", "R003"]
  - name: "QualityChecker", description: "Atomic model. It checks
    finished parts and writes its own state and result records to
    stdout.", children_names: [], related_requirement_ids: ["R002", "
    R003"]

For a timestamped stdin schedule that drives periodic state
    observations, use one source and one state process; the state
    process that creates each record writes it directly:
- name: "ScheduleSystem", description: "Coupled root. Routes scheduled
     values to the state process.", children_names: ["ScheduleSource",
     "StateProcess"], related_requirement_ids: ["R001", "R002", "R003
    "]
- name: "ScheduleSource", description: "The only atomic model that
    reads the timestamped stdin schedule. It sends the applicable
    value at every required observation time.", children_names: [],
    related_requirement_ids: ["R001", "R002"]
- name: "StateProcess", description: "Atomic model. Updates its
    retained state from each scheduled value and writes the
    observation it produces to stdout.", children_names: [],
    related_requirement_ids: ["R002", "R003"]
There is no separate writer because no cross-model aggregation is
    required.

For a multi-stage workflow whose only stdout is one combined final
    document, make aggregation explicit rather than letting each stage
     print JSONL:
- name: "AccessSystem", description: "Coupled root. Routes requests
    through the processing pipeline and routes observable facts to one
     final report collector.", children_names: ["RequestSource", "
    AccessPipeline", "ReportCollector"], related_requirement_ids: ["
    R001", "R002", "R003"]
- name: "RequestSource", description: "The only atomic model that
    reads the request stream. It sends each parsed request to
    AccessPipeline and sends the corresponding input observation to
    ReportCollector.", children_names: [], related_requirement_ids: ["
    R001", "R002"]
- name: "AccessPipeline", description: "Atomic model. Applies the
    ordered delays and busy-state rules, sends each resulting event
```

```
    and operation fact to ReportCollector, and does not write stdout
    .", children_names: [], related_requirement_ids: ["R002", "R003"]
- name: "ReportCollector", description: "Atomic model. Collects event
    and operation facts, derives the final summary fields, and writes
    exactly one combined JSON object to stdout when the simulation
    exits.", children_names: [], related_requirement_ids: ["R003"]

For a CLI-configured model with one tightly coupled numeric state and
    one final record, keep only a structural root and one stateful
    child:
`[{{"name":"PopulationSystem","description":"Coupled root. Contains
    the complete population process.","children_names":["
    PopulationProcess"],"related_requirement_ids":["R001","R002
    "]}},{{"name":"PopulationProcess","description":"Atomic model.
    Advances the complete state from constructor parameters and writes
     the required final record once.","children_names":[],"
    related_requirement_ids":["R001","R002"]}}]`.
A final record produced from this child's own state is not cross-model
     aggregation. Do not add an InputHandler, OutputHandler,
    configuration-message port, or final-state port around it.

For a configurable family of independent workers coordinated by one
    dispatcher, keep the repeated model as an atomic blueprint
    directly under the coupled root:
- name: "FleetSystem", description: "Coupled root. Creates `
    num_workers` Worker instances and routes tagged assignments and
    availability reports.", children_names: ["Dispatcher", "Worker"],
    related_requirement_ids: ["R001", "R002"]
- name: "Dispatcher", description: "Atomic model. Selects an available
     worker and sends a tagged assignment.", children_names: [],
    related_requirement_ids: ["R001"]
- name: "Worker", description: "Atomic blueprint instantiated `
    num_workers` times by FleetSystem, with a stable worker ID;
    accepts only assignments addressed to that ID.", children_names:
    [], related_requirement_ids: ["R001", "R002"]
Do not insert a leaf `WorkerPool` that claims to contain Worker
    instances: a module that actually contains child instances is
    coupled and must list its child blueprint in `children_names`.
```

G.2 Detailed Planning

Detailed-planning prompts are assembled for the current node. Non-root nodes receive their inherited contract.

Detailed-plan prompt assembly order

```
<SystemRole>
You are a DEVS System Architect.
</SystemRole>

<TargetContext>
Module Name: {target_name}
Module Type: {module_type}
Direct Children: {children_names_str}
</TargetContext>

{root_inheritance_rules_or_locked_inherited_contract}

<RelevantRequirementsForCurrentSubtree>
{requirement_focus}
</RelevantRequirementsForCurrentSubtree>
```

```

<SystemRequirements>
{original_requirements}
</SystemRequirements>

<GlobalPlanOverview>
{global_plan}
</GlobalPlanOverview>

{parent_context_and_sibling_contracts_for_non_root_nodes}
{responsibility_guidance}
{root_coupled_root_atomic_inherited_coupled_or_atomic_instruction}
{atomic_implementation_example_catalog_for_atomic_nodes}
{selected_stage_example}
{field_contract}
{explicit_inter_model_flow_index_for_coupled_nodes}

<FinalContractReminder>
{node_specific_final_check}
</FinalContractReminder>
```

The root plan is then rewritten as one complete object.

Root-plan rewrite template

```
<SystemRole>
You revise one draft of a ROOT COUPLED DEVS plan.
</SystemRole>

<TargetContext>
Root class: {target_name}
Required direct children: {children_names}
</TargetContext>

<SystemRequirements>
{original_requirements}
</SystemRequirements>

<GlobalPlanOverview>
{global_plan}
</GlobalPlanOverview>

<Candidate1>
{root_plan_draft}
</Candidate1>

{selected_root_stage_example}

<ExplicitInterModelFlowIndex>
{requirement_flow_index}
</ExplicitInterModelFlowIndex>

<TaskInstruction>
Produce one complete replacement plan in the required response schema.
    Do not return criticism, confidence, uncertainties, or a choice
    between candidates.

The candidate is a fallible working draft, not a locked contract.
    Audit its event flow and interfaces before reusing it. Preserve
    fields that survive that audit, correct every concrete omission,
    contradiction, or unrealizable interface, and return the complete
```

```
    revised plan. If no correction is needed, return the complete
    candidate unchanged.

Keep the required direct-child names and types from the global plan.
    Keep OS operations in child `external_io`; stdin, stdout, stderr,
    and files are not coupling endpoints. The coupled root itself
    performs no active behavior.

Return the complete root `detailed_plan`, every `children_plans` entry
    , and all actual `coupling_rules`. Finalize the revised child port
     lists first, then copy every coupling endpoint from those lists
    or the declared root boundary. For every route, confirm that its
    source emits the event and its target consumes it. Do not defer a
    discrepancy to later child planning or code generation.
</TaskInstruction>
```

G.3 Behavioral Synthesis

The code-generation call combines the outer template with the shared engineering standards and either the atomic- or coupled-model block shown below. Its remaining placeholders contain the node's plan, relevant requirements, local context, and selected example.

Code-generation prompt template

```
## [Task]
Construct a complete Python file containing a **{model_type} DEVS
    model** named `{name}` using `xdevs.py`.

## [LOCKED IMPLEMENTATION CONTRACT - READ FIRST]
The JSON below is the authoritative contract for THIS model. Its class
     name, constructor arguments, ports, port types, protocols,
    external IO operations, and behavioral requirements are already
    planned. Do not add, remove, rename, or retype interface fields.

If the original system requirements or a reference example appear to
    require a different interface, keep this locked interface.
    Implement compatible behavior when possible; otherwise report the
    incompatibility instead of silently changing the contract.

The locked JSON controls which interfaces exist. When its behavioral
    prose is shorter than the unabridged requirements--for example, `
    external_io` says only "read stdin"--recover the exact line format
    , field meanings, timing, and parsing rules from the original
    requirements. That is completion of the same interface, not an
    interface change. Never copy a reference example's input format
    over the format stated by the original requirements.

<locked_implementation_contract>
{spec}
</locked_implementation_contract>

{boundary_effect_notice}

## [Relevant Requirements for This Model/Subtree]
This ledger slice is a navigation aid. Use it to keep the module
    aligned with the obligations assigned during architecture planning
    . The unabridged original requirements remain authoritative if the
     slice is incomplete or ambiguous.

<relevant_requirements>
```

```
{requirement_focus}
</relevant_requirements>

## [Original System Requirements - Unabridged Reference]
Use the original text to recover constants, boundary conditions, timing, and global semantics that may be abbreviated in the module plan. It does not grant permission to implement sibling responsibilities or change the locked interface.

<original_system_requirements>
{original_requirements}
</original_system_requirements>

{global_standards}

{model_specific_instructions}

## [Selected Creator Skills]
{model_skills}

## [Context Info]
**Sub-Models (for Coupled definitions)**:
{sub_models}

**System Context**:
(The environment around this model)
{context_str}

## [Utils]
{util_desc}

## [Class Definitions]
{definitions}

## [Selected Complete-File Control-Flow Template]
Each selected file demonstrates an implementation mechanism, not the target domain or interface. Reuse its DEVS scheduling and control-flow shape when it matches the selected mechanism, while taking names, ports, constants, schemas, and external IO only from the locked contract.
{example}

{required_child_imports}

## [FINAL CONTRACT CHECK]
Before responding, re-read `<locked_implementation_contract>` and verify that the class, exact constructor arguments, exact ports, and external IO match it. DEVS output ports are written only in `lambdaf()`. Lifecycle/debug text must not be written to stdout. If the behavior is impossible under the locked interface, report the incompatibility instead of inventing an interface.

## [Output]
Return only the complete Python code enclosed in `<python_code>` and `</python_code>`. Do not use Markdown backticks or include an explanation.
```

Every code-generation call includes the following block.

Global engineering standards

```
## [Global Standards - STRICT]
### Code Basics
1. **Imports**: Include `from xdevs.models import Atomic, Coupled, Port` in the initial import block. Use normal Python 3.10+ annotation evaluation; do not enable postponed annotation evaluation. Allowed dependencies are the Python standard library, `numpy`, `pandas`, and `xdevs`. Use `devs_project.devs_utils.xxx` only for project utilities explicitly listed in [Utils].
2. **Typing**: Do not quote xDEVS types in annotations. Write `parent: Coupled | None`, with `Coupled` imported as above.
3. **Simulation Time**: If the contract needs the current simulation time, import `get_current_time` exactly as shown in [Utils] and call `get_current_time()`. Do not call `self.get_current_time()` or define a wrapper with that name.
4. **Scope**: Implement the specified behavior directly. Do not add unrelated features or dependencies.

### External IO
5. **Implementation**:
    - `external_io` is specification metadata, not an xDEVS callback name. Do not define an `external_io()` method: xDEVS will never call it. Perform normal Python IO at the required semantic point inside `initialize`, `deltext`, `lambdaf`, `deltint`, `exit`, or a helper that one of those methods actually calls. Do NOT use `self.logger`, `get_sim_logger`, or any custom logger helper.
    - For `target="stdout"` with JSON/JSONL content, use `print(json.dumps(record), flush=True)` where `record` follows `content` exactly.
    - For `target="stderr"`, use `print(..., file=sys.stderr, flush=True)` or `sys.stderr.write(...)`.
    - For `target="stdin"`, follow the exact read timing and lifetime stated in [LOCKED IMPLEMENTATION CONTRACT].
    - For `target="file"`, follow the path/resource source, read/write direction, and append/overwrite behavior described in `content`.
    - DO NOT emit external IO records that belong to sub-models.
    - Emit each record at the semantic event time required by `content`. Receiving input, completing processing, and sending a DEVS output are different moments unless `content` explicitly equates them. Use one DEVS clock unit consistently and convert only when the external schema requires it.
6. **External IO Field Name Convention**: The Specification defines the exact field names for external records (e.g., which field identifies the source module, which fields are in the payload). Use those exact field names and values from the Specification. If a Specification says the module identifier is "reception", use "reception" -- do not substitute the Python class name or change casing.
7. **Ports vs External IO**:
    - DEVS ports are simulator-level communication channels registered with `add_in_port` / `add_out_port`, accessed through `self.input[...]` and `self.output[...]`, and connectable through couplings.
    - external_io is OS/environment-level interaction such as stdin/stdout/stderr/files/other resources. It is a side effect, not a DEVS port.

### Interface Consistency
8. **Current Model Boundary**:
    - Ports of the current model MUST exactly match the names and types required by the applicable interface authority described in the model-specific rules. Do NOT add, remove, or rename ports.
```

```
    - `__init__` arguments of the current model MUST exactly match the applicable interface authority. Do NOT add `*args` or `**kwargs`.
    - Required external_io targets, event names, content schemas, and payload keys in [LOCKED IMPLEMENTATION CONTRACT] are HARD requirements. Do NOT replace required events with near-synonyms.
```

Atomic calls include the following block. The optional delayed-processing pattern is omitted when it conflicts with the selected direct-periodic-output example.

Atomic-model instructions

```
### [Atomic Core Rules - STRICT]
#### Interface Authority
- For an atomic model, [LOCKED IMPLEMENTATION CONTRACT] is authoritative for ports, `__init__` arguments, behavior, and external_io.
- Register all ports and `__init__` arguments exactly as stated there. `__init__` must always start with `(self, name: str, parent: Coupled | None, ...)`.
- Implement only the external streams listed in `external_io`. Ensure payload keys, source/derivation logic, timing, and values match the locked contract exactly.

#### Implementation Workflow
MUST implement all the following methods!
0. **Configuration ownership**: Model files do not parse CLI arguments and do not import `argparse`. The generated runner owns CLI parsing and passes behavior-relevant values through the constructor declared by the locked contract.
1. **Class Definition**: Import `from xdevs.models import Atomic, Coupled, Port` and inherit from `Atomic`.
2. **`initialize()`**: Initialize internal variables, then either schedule a required internal event with `self.hold_in(phase, sigma)` or wait for input with `self.passivate(phase)`.
    - DO NOT emit DEVS port output here.
    - If initialization requires an immediate DEVS output, schedule a zero-delay phase such as `self.hold_in("OUTPUT_READY", 0.0)` and emit in `lambdaf()`.
3. **`deltext(e)`**: Read all external DEVS inputs via `for packet in self.input["port"].values:`.
    - If the model remains in the same active phase, preserve remaining time with `self.hold_in(self.phase, max(0.0, self.ta() - e))`.
    - If the model is idle and newly received input should start work, store the current item and call `self.hold_in(some_phase, delay)`.
4. **`lambdaf()`**: This is the only place that may emit DEVS port output. Use `self.output["port"].add(payload)`. State changes inside `lambdaf()` are allowed when required by the specified transition design.
5. **`deltint()`**: This runs after `lambdaf()` when an internal event fires.
    - Update internal state, then schedule the next event with `self.hold_in(phase, sigma)` or wait for input with `self.passivate(phase)`.
6. **`exit()`**: Release resources and write final external IO only if required by `external_io`.

#### Output Scheduling Contract
For an output design that uses a prepared payload, use this cycle:
1. Before the output phase, prepare a payload variable such as `self.payload_to_send`.
```

```
2. Schedule the phase observed by the next `lambdaf()` using `self.
    hold_in(phase, sigma)`.
3. In `lambdaf()`, check the current phase and emit the prepared
    payload.
4. In the following `deltint()`, clear the emitted payload and advance
     to the next state.

"Immediate" or "zero-delay" output after `deltext()` means preparing
    the payload or batch and scheduling `hold_in(..., 0.0)`; it never
    means writing a DEVS output port from `deltext()`.

For outputs that occur only after a processing delay, a useful pattern
     is `PROCESSING -> OUTPUT_READY(0.0) -> IDLE or PROCESSING`. This
    is a pattern, not a mandatory state-machine shape; use the simpler
     direct-output transition when it correctly implements the locked
    contract. Any `sigma=0` phase chain must terminate by passivating,
     waiting for new input, or scheduling a positive delay.

Keep the default `deltcon` behavior unless the locked contract
    explicitly requires external events to be processed before
    simultaneous internal events.
```

Coupled calls include the following block instead.

Coupled-model instructions

```
### [Coupled Core Rules - STRICT]
#### Interface Authority and Precedence
1. The [LOCKED IMPLEMENTATION CONTRACT] controls this coupled model's
    own constructor, boundary ports, and direct external IO.
2. [Sub-Models] gives each child's `relative_file_path`, planned `
    interface_manifest`, and optional exact `
    construction_source_excerpt`. If the two conflict on constructor
    or port names, the excerpt controls because it is actual source.
    Use it only to instantiate/connect the child; never copy child
    statements into this parent.
3. [Coupling Rules] is topology only. Use real endpoints from 1-2;
    never invent or rename ports or arguments to match stale prose.

#### Container Rules
- Import `from xdevs.models import Atomic, Coupled, Port` and inherit
    from `Coupled`.
- A structural coupled model does not read simulation time at runtime:
     do not import or call `get_current_time` in this file.
- Treat this class as a pure structure container. Implement ONLY `
    __init__`: no state machine or event handlers.
- Do not add `main()`, CLI parsing, a `Coordinator`, or a simulation
    loop; generated `run.py` owns them.
- Copy every line from [Required Child Imports - COPY EXACTLY] into
    the initial import block. Each is derived from the generated child
    's actual path, so do not shorten it or replace it with a class-
    name guess.
- Every generated-child import starts with `from .`; an unprefixed
    child import is invalid inside the generated package.
- `name` and `parent` are framework-reserved constructor arguments.
    Include each exactly once. Do not pass a second keyword named `
    name` or `parent` to child constructors.

#### Constructor (`__init__`) Workflow
1. Signature MUST start exactly with `(self, name: str, parent:
    Coupled | None, ...)`.
```

```
2. Call `super().__init__(name)` and set `self.parent = parent`.
3. Register this coupled model's boundary ports using `self.add_in_port()` and `self.add_out_port()`.
4. Instantiate components and register them via `self.add_component(instance)`.
    - Pass exactly the required child arguments. Copy values from this model's arguments or explicit child constants; never substitute domain guesses.
    - Do not depend on mutating parent `self.param` after creating children.
    - If the locked contract or [Context Info] includes dynamic counts (e.g., `arg:num_x`), instantiate ALL required child instances with deterministic names like `<Group>_<index>` and build couplings programmatically. Do NOT collapse to one representative instance.
5. Define couplings using `self.add_coupling(src, dst)`.
    - Do NOT guess or invent port names. Use exact sub-model instance names and ports from the interface entry.
    - Use only EIC (self input -> child input), IC (child output -> child input), and EOC (child output -> self output).
6. Do not print, write files, or emit external IO unless the coupled model itself has an explicit `external_io` entry.
```

H BENCHMARK SCENARIO SPECIFICATIONS

This section provides detailed descriptions for the ten benchmark scenarios summarized in Table 6. For each scenario, we describe the system objective, the constituent entities, and the governing dynamics. We use the compact benchmark names throughout. For example, the release directories for FileTransfer and StratAirlift are `benchmark/oft/` and `benchmark/SA/`, respectively. The scenario specifications were constructed with reference to publicly available models in the Cadmium Simulation Environment. The parent snapshot at commit `62af4c65` pins the `DEVS-Models` collection and its model-specific repositories. Table 7 links each benchmark name directly to the corresponding pinned model revision. We adapted the described entities, interfaces, and dynamics to the benchmark's natural-language specification and trace contract. Generated simulators are evaluated against those specifications rather than by source-code equivalence.

Table 6: Benchmark scenario summary. The dataset covers diverse domains and dynamic characteristics. We report specification length (Words), number of entities (Ents), distinct event types (Evts), and reference implementation size (Ref. LOC, short for lines of code).

Scenario	Domain	Key Dynamics	Words	Ents	Evts	Ref. LOC
SEIRD	Bio-Math	Fixed-Step ODE Approx.	507	5	1	745
ABP	Network	Stop-and-Wait	854	4	5	573
Barbershop	Service	Blocking & Signals	700	3	2	570
HouseHeating	Control	Thermostat Feedback	384	4	1	639
SAA	Security	Access-Control Pipeline	431	4	4	726
StoreCashier	Service	Two-Server FIFO Queue	309	4	4	569
IOBS	Banking	Pipeline w/ Routing	822	6	7	703
OTrain	Transport	Schedule-driven	755	4	4	961
FileTransfer	Network	Dual-Loop FSM	750	8	12	937
StratAirlift	Logistics	Active Reneging	966	5	9	1476

H.1 ONLINE BANKING SYSTEM (IOBS)

Domain: Banking **Dynamics Type:** Pipeline with Probabilistic Routing

Table 7: Per-scenario provenance. Each link targets the model-specific revision pinned by the Cadmium repository snapshot.

Benchmark scenario	Reference model directory
SEIRD	`Cadmium-SEIRD@2bafab36`
ABP	`Cadmium-AlternatingBitProtocol@42b59474`
Barbershop	`Cadmium-Barbershop@8aa1a080`
HouseHeating	`Cadmium-House_Heating_System@c848584e`
SAA	`Cadmium-Secure_Area_Access@f1e70fb5`
StoreCashier	`Cadmium-Store_Cashier@2bd750fc`
IOBS	`Cadmium-Online_Banking_System@4aee863d`
OTrain	`Cadmium-otrain@c317f182`
FileTransfer	`Cadmium-Offline_File_Transfer@20dbf3ec`
StratAirlift	`Cadmium-Start_Airlift@f492ffe1`

- This scenario models a linear transaction-processing pipeline for an online banking system, in which requests pass through security and processing stages.
- The system contains an input reader followed by five processing modules: Account Access Manager, Account Number Verifier, Password Verifier, Bill Payment Manager, and Transaction Process Manager.
- A request enters at a specified time and passes through the pipeline, with a fixed processing delay of 10 seconds at each stage.
  - At the ANV stage, a request passes with probability 0.5; otherwise it leaves the pipeline. At the PV stage, password attempts continue until one succeeds, with success probability 0.5 per attempt.
  - The final TPM stage maintains the account balance and updates it only after the request has completed the preceding stages.

H.2 O-Train Light Rail (OTrain)

Domain: Transportation **Dynamics Type:** Schedule-driven with Serial Delays

- This scenario simulates a light-rail system in which one train travels in both directions between five stations, from Bayview to Greenboro. It combines a fixed train schedule with stochastic passenger arrivals.
- The four modeled component types are the train scheduler, station passenger generators, station waiting queues, and in-train alighting queues. Individual passengers appear as transient records and message payloads.
- Its main dynamics are:
  - The train follows a timetable with fixed travel intervals between stations.
  - Boarding and alighting occur one passenger at a time. Each passenger takes 0.025 seconds, so the total dwell time grows with the number of passengers.
  - Stations generate passengers at normally distributed intervals subject to origin–destination constraints.

H.3 Epidemic Compartmental Model (SEIRD)

Domain: Biological Systems **Dynamics Type:** Fixed-Step ODE Approximation

- This scenario implements a SEIRD (Susceptible–Exposed–Infective–Recovered–Deceased) model for disease spread in a closed population.
- The population is divided into five aggregate compartments (S, E, I, R, D).
- The continuous dynamics are approximated with Forward Euler integration.

  - Updates occur every Δt days; the evaluated cases use $\Delta t = 0.01$. Compartment flows depend on the case-specific transmission rate β, incubation period, infectivity period, and mortality rate. In these cases, β is 0, 2.5, or 5 per day.
  - The five compartments preserve the total population up to the numerical tolerance used by the evaluator.

H.4 Offline File Transfer (FileTransfer)

Domain: Network Systems **Dynamics Type:** Dual-Loop FSM with Interrupts

- This scenario simulates file synchronization in which an intermediate server buffer separates upload from download.
- The system contains a command-stream input handler, a *Sender*, a buffering *Server*, a *Receiver*, and four reliable *Subnets*.
- It contains two asynchronous loops:
  - The upload loop (Sender–Server) and download loop (Server–Receiver) operate independently and share the server's storage queue. Both use the Alternating Bit Protocol (ABP).
  - Commands can pause or resume the download loop, while queued files and protocol state persist across the interruption.

H.5 Alternating Bit Protocol (ABP)

Domain: Network Protocols **Dynamics Type:** Stop-and-Wait with Deterministic Noise

- This scenario uses the Alternating Bit Protocol (ABP) for reliable transmission over lossy channels.
- It consists of a *Sender*, a *Receiver*, and two *Subnets* for the forward and backward directions.
- Its main dynamics are:
  - The Sender follows a stop-and-wait state machine: after sending a packet, it waits for an ACK with the matching bit or for the timeout to expire.
  - Each subnet uses a Linear Congruential Generator to decide whether a packet passes or is dropped. A fixed seed therefore produces the same loss sequence.

H.6 Strategic Airlift Operations (StratAirlift)

Domain: Logistics **Dynamics Type:** Active Reneging & Resource Cycling

- This scenario models the transport of cargo pallets by a fleet of aircraft.
- It includes a source *Facility*, a *Loading Queue*, a *Fleet Coordinator*, multiple *Aircraft*, and a *Destination*.
- Its main dynamics are:
  - Pallets have a finite waiting time and leave the queue when that time expires, including when no new external event arrives.
  - Aircraft follow a multi-stage cycle, and the coordinator matches available cargo with idle aircraft.

H.7 Barbershop (Barbershop)

Domain: Service Queueing **Dynamics Type:** Blocking & Handshake Signaling

- This scenario models a barbershop with a reception area and two service stages: inspection and cutting.
- The components are a *Reception Desk*, an *Inspection Station*, and a *Cutting Station*.
- Its main dynamics are:

  - The reception queue has capacity eight; arrivals are rejected while it is full.
  - Inspection and cutting operate sequentially. Inspection waits for the cutting station's completion signal before releasing its current customer and accepting another from reception.

H.8 HOUSE HEATING (HOUSEHEATING)

Domain: Control Systems **Dynamics Type:** Thermostat Feedback

- This scenario models a single heated room controlled by a bang-bang heater. The controller keeps the room near a target temperature while outdoor readings affect heat loss.
- The system includes the room temperature, an outdoor-temperature schedule, a heater, and a threshold controller.
- Its main dynamics are:
  - At each integer simulation second, the room loses a fixed fraction of the gap between its previous temperature and the effective outdoor temperature. Outdoor temperature can reduce heat loss but does not directly warm the room.
  - Heater output follows the preceding control signal. After observing the new room temperature, the controller sets the signal for the next step.

H.9 SECURE AREA ACCESS CONTROL (SAA)

Domain: Security Workflow **Dynamics Type:** Access-Control Pipeline

- This scenario simulates a secure-area alarm system that accepts arm and disarm requests, authenticates them, and updates a display.
- The system consists of an input reader, an alarm administrator, an authentication step, and a display.
- Its main dynamics are:
  - Accepted requests pass through the administrator, authentication, and display stages with configured delays. Authentication succeeds deterministically.
  - A request that arrives while the administrator is busy is recorded as input but does not enter the downstream pipeline. Redundant accepted requests still traverse the pipeline without creating duplicate state transitions.

H.10 STORE CASHIER (STORECASHIER)

Domain: Service Queueing **Dynamics Type:** Two-Server FIFO Queue

- This scenario models a cashier system with one client generator, a FIFO waiting queue, and two employees.
- The system consists of a client generator, a waiting queue, and two employee servers.
- Its main dynamics are:
  - A waiting customer is paired with an available employee as soon as both are present. FIFO order prevents later customers from bypassing earlier unpaired customers.
  - Each employee serves at most one customer at a time. Completion records the customer's arrival time, completion time, and delay, after which the employee becomes available again.

I STRATEGIC AIRLIFT CASE STUDY

The Strategic Airlift (SA) example in Figures 1 and 2 follows the modeling workflow from a specification through planning artifacts to code and an event trace. The execution trace comes from a generated project with local corrections to trace output, queue handling, and the simulation horizon. This demonstration passes all nine SA test cases with $\mathrm{Score}_{\mathrm{ope}} = \mathrm{Score}_{\mathrm{beh}} = 1.0$.

I.1 Specification and Requirement Index

Listings 2–3 present the SA behavioral and operational specifications.

```
### Scenario: Airfreight Logistics Operations

1. **System Objective**:
   Simulate an airfreight logistics system managing cargo transport via multiple aircraft. The system continuously receives cargo (pallets) at a loading facility, queues them for transport, and delivers them via scheduled aircraft to a destination. Each pallet has a deadline; pallets not delivered by deadline are discarded.

2. **System Entities and Behaviors**:

   a) **Facility (Cargo Source)**:
      - Continuously generates new cargo pallets at regular intervals.
      - Each pallet receives a unique ID and a deadline (expiration time, wich is considered ONLY while the pallet is in the Loading Queue. Once assigned to an aircraft, the pallet is considered safe and will be delivered regardless of time.).
      - Newly generated pallets are immediately sent to the Loading Queue.

   b) **Loading Queue**:
      - Holds incoming pallets awaiting aircraft assignment.
      - **Active Expiration**: Monitors pallet deadlines. If a pallet remains in the queue and its deadline is reached, it is IMMEDIATELY discarded, triggers a `pallet_expired` event, and removes it from the queue.
      - Non-expired pallets remain queued until assigned to an aircraft by the Coordinator.

   c) **Fleet Coordinator**:
      - Monitors aircraft availability and cargo demand.
      - When (Loading Queue has pallets) AND (an Aircraft is Idle), assigns the next pallet (FIFO) to the aircraft.
      - Tracks aircraft state (idle, in-flight, unloading, returning, maintenance).

   d) **Aircraft** (multiple, e.g., Aircraft-1, Aircraft-2):
      - Each aircraft performs a cyclic operation:
        1. Wait idle at facility (until assigned)
        2. Load assigned cargo (Instantaneous: 0s)
        3. Fly to destination (takes `flight_time`)
        4. Unload cargo (takes `unload_time`) -> Delivery happens here
        5. Return to facility (takes `return_time`)
        6. Undergo maintenance/rest (takes `maintenance_time`)
      - Aircraft transitions between states as timers expire.
      - Reports status changes to Fleet Coordinator.

   e) **Destination**:
      - Logically receives delivered cargo from aircraft.
      - Records successful delivery upon completion of the aircraft's "Unload" phase.

3. **Operational Flow**:

   1. Facility generates pallet at t=0, t+pallet_interval, ...
   2. New pallet joins Loading Queue.
   3. If pallet stays in queue > `pallet_expiration_time` -> Discarded immediately.
   4. Fleet Coordinator checks: if aircraft available AND cargo available → assign.
   5. Aircraft begins transport cycle:
```

```
      - Load (0s) → Fly (30s) → Unload (2s) → [Trigger Delivery] → Return (30s) → Maintain (10s) → Idle.
   6. Simulation continues until `total_duration` time is reached.

4. **Scenario Rules**:

   - **Time Unit**: All durations in seconds (configurable scale).
   - **Pallet Deadline**: Pallet expires exactly at `generation_time + pallet_expiration_time`.
   - **Latency Calculation**: `latency = delivery_time - generation_time`. Delivery time is the moment unloading completes.
   - **Aircraft Utilization**: Aircraft capacity = 1 pallet per trip.
   - **Loading Time**: Assumed to be negligible (0.0s).
   - **Fair Queuing**: Pallets loaded in FIFO order.
   - **No Preemption**: Once aircraft departs with cargo, it completes full cycle.
```

Listing 2: SA behavioral specification

```
1. Command Line Arguments:
The script must accept the following named arguments:
* `--duration` (float): Total simulation time in time units. Default: 10000.0
* `--num_aircraft` (int): Number of aircraft in the system. An integer >= 1. Default: 2
* `--pallet_interval` (float): Time interval between pallet generations in time units. Default: 25.0
* `--pallet_expiration_time` (float): Time window for pallet expiration (relative to generation). Default: 150.0
* `--flight_time` (float): Flight duration for aircraft transport in time units. Default: 30.0
* `--unload_time` (float): Time required for unloading cargo at destination. Default: 2.0
* `--return_time` (float): Return flight duration for aircraft. Default: 30.0
* `--maintenance_time` (float): Duration of maintenance phase for aircraft. Default: 10.0

2. stdin Format:
* No stdin input is required. The system uses command line arguments for configuration.

3. **Standard Output (stdout)**:
* Format: JSONL (JSON Lines - one JSON object per line).
* Each line is an independent JSON object representing a key event.
* Schema for each log line:
```json
{
    "time": <float>,        # Absolute simulation time
    "entity": <str>,        # Entity name: "facility", "queue", "coordinator", "aircraft", "destination"
    "event": <str>,         # Event name (see Event Types below)
    "payload": <dict>       # Event-specific data
}
```
* Example: `print(json.dumps({"time": 10.5, "entity": "aircraft", "event": "pallet_delivered", "payload": {"aircraft_id": 1, "pallet_id": 42, "latency": 65.5}}))`

Event Types and Formats:

Facility Events:
* `pallet_generated`: New pallet created
  - entity: `"facility"`
  - payload:
```

````
    ```json
    {
      "pallet_id": <int>,       # Unique ID of the generated pallet
      "expiration_time": <float> # Absolute simulation time (Deadline) when pallet expires
    }
    ```
  - Trigger: When Facility generates a new pallet

Queue Events:
* `pallet_queued`: Pallet added to loading queue
  - entity: `"queue"`
  - payload:
    ```json
    {
      "pallet_id": <int>,       # ID of the pallet entering the queue
      "queue_size": <int>       # Current number of pallets in queue (including this one)
    }
    ```
  - Trigger: When pallet enters queue

* `pallet_expired`: Pallet deadline exceeded
  - entity: `"queue"`
  - payload:
    ```json
    {
      "pallet_id": <int>,       # ID of the pallet being discarded
      "total_expired": <int>    # Cumulative count of expired pallets so far
    }
    ```
  - Trigger: Exactly when current_time == pallet_expiration_deadline

Coordinator Events:
* `assignment_created`: Cargo assigned to aircraft
  - entity: `"coordinator"`
  - payload:
    ```json
    {
      "aircraft_id": <int>,     # ID of the assigned aircraft
      "pallet_id": <int>        # ID of the assigned pallet
    }
    ```
  - Trigger: When Coordinator assigns a pallet to aircraft (Loading starts immediately)

Aircraft Events:
* `depart`: Aircraft departs with cargo
  - entity: `"aircraft"`
  - payload:
    ```json
    {
      "aircraft_id": <int>,     # ID of the departing aircraft
      "pallet_id": <int>        # ID of the cargo on board
    }
    ```
  - Trigger: When Aircraft finishes loading (0s duration). Mathematically: time == assignment_time.

* `return`: Aircraft returns to facility
  - entity: `"aircraft"`
  - payload:
    ```json
    {
````
````

```
      "aircraft_id": <int>      # ID of the returning aircraft
    }
    ```
  - Trigger: When Aircraft completes return flight

* `maintenance_start`: Aircraft enters maintenance
  - entity: `"aircraft"`
  - payload:
    ```json
    {
      "aircraft_id": <int>      # ID of the aircraft starting maintenance
    }
    ```
  - Trigger: When Aircraft begins maintenance phase (immediately after
    return)

* `maintenance_end`: Aircraft completes maintenance
  - entity: `"aircraft"`
  - payload:
    ```json
    {
      "aircraft_id": <int>      # ID of the aircraft finishing
    maintenance
    }
    ```
  - Trigger: When Aircraft returns to idle state

Destination Events:
* `pallet_delivered`: Pallet successfully delivered
  - entity: `"destination"`
  - payload:
    ```json
    {
      "pallet_id": <int>,       # ID of the delivered pallet
      "aircraft_id": <int>,     # ID of the transporting aircraft
      "latency": <float>        # Total time taken (delivery_time -
    generation_time) in seconds
    }
    ```
  - Trigger: When Aircraft finishes the 'unload' phase
  - Note: Latency = time_now - pallet_gen_time.
```

Listing 3: SA operational specification

Listing 4 shows two entries from the requirement index reconstructed for this demonstration: pallet assignment (R006) and aircraft readiness feedback (R008). S02 and S03 refer to the scenario behavior and external trace and internal interfaces sections of that reconstructed index. The selected entries are unchanged; source-section text and other entries are omitted.

```
[
  {
    "id": "R006",
    "detail": "When queue stock and idle capacity exist, request one FIFO
     pallet using a boolean, keep one request outstanding, and assign to
    the lowest-ID idle aircraft. Capacity is one; departure is
    instantaneous.",
    "source_section_ids": ["S02","S03"],
    "flows": [
      {"producer": "FleetCoordinator", "consumer": "PalletQueue",
       "producer_port": "queue_request", "consumer_port": "request_pallet
    ",
       "payload": "bool: True",
       "trigger": "Stock and idle aircraft; no pending request"},
      {"producer": "PalletQueue", "consumer": "FleetCoordinator",
       "producer_port": "pallet_out", "consumer_port": "pallet_in",
```
```

```
        "payload": "{pallet_id:int, expiration_time:float, generation_time
    :float}",
        "trigger": "FIFO release after a request"},
      {"producer": "PalletQueue", "consumer": "FleetCoordinator",
       "producer_port": "queue_status", "consumer_port": "queue_size_in",
       "payload": "int: number of waiting pallets",
       "trigger": "After queue changes"},
      {"producer": "FleetCoordinator", "consumer": "Aircraft",
       "producer_port": "assign_to_aircraft_i", "consumer_port": "
    assignment_in",
        "payload": "{pallet_id:int, expiration_time:float, generation_time
    :float}; i=1,...,num_aircraft",
        "trigger": "Lowest-ID idle aircraft selected"}
    ]
  },
  {
    "id": "R008",
    "detail": "Send the 1-based aircraft ID as a string at t=0 and after
    maintenance, through Aircraft to the coordinator, restoring
    availability.",
    "source_section_ids": ["S02","S03"],
    "flows": [
      {"producer": "MissionController", "consumer": "Aircraft",
       "producer_port": "ready_out", "consumer_port": "ready_out",
       "payload": "str(aircraft_id)",
       "trigger": "At t=0 and after maintenance"},
      {"producer": "Aircraft", "consumer": "FleetCoordinator",
       "producer_port": "ready_out", "consumer_port": "aircraft_ready_i",
       "payload": "str(aircraft_id); i=1,...,num_aircraft",
       "trigger": "At t=0 and after maintenance"}
    ]
  }
]
```

Listing 4: Selected entries from the SA requirement index

The trace in Section I.4 uses the following integration-test configuration rather than the default argument values. Events at the simulation horizon are excluded.

```
python run.py --duration 1000 --num_aircraft 3 \
  --pallet_interval 15 --pallet_expiration_time 180 \
  --flight_time 35 --unload_time 3 \
  --return_time 35 --maintenance_time 10
```

Listing 5: SA integration-test configuration

I.2 Planning Skeleton

The skeleton contains seven model types (Listing 6). The root is coupled, and its Aircraft child is itself coupled: MissionController manages the transport cycle, while FlightTimer schedules the completion of each phase. This nested structure separates fleet coordination from the timing and state of an individual aircraft. The root creates `num_aircraft` instances of the Aircraft type; the three-aircraft configuration therefore has thirteen model instances, comprising four coupled and nine atomic models.

Each node retains the identifiers of its assigned requirements. For example, R008 links fleet coordination to the Aircraft boundary and its MissionController, making the readiness feedback path explicit before implementation.

```
[
  {
    "name": "StrategicAirlift_D0",
    "children_names":
      [
```

```
        "PalletFacility", "PalletQueue", "FleetCoordinator",
        "Aircraft"
      ],
    "related_requirement_ids": [
      "R001", "R002", "R003", "R004", "R005", "R006", "R007", "R008",
      "R009"
    ]
  },
  {
    "name": "PalletFacility",
    "children_names": [],
    "related_requirement_ids": ["R002", "R004"]
  },
  {
    "name": "PalletQueue",
    "children_names": [],
    "related_requirement_ids": ["R002", "R004", "R005", "R006"]
  },
  {
    "name": "FleetCoordinator",
    "children_names": [],
    "related_requirement_ids": ["R002", "R005", "R006", "R008"]
  },
  {
    "name": "Aircraft",
    "children_names": ["MissionController", "FlightTimer"],
    "related_requirement_ids":
      ["R002", "R006", "R007", "R008", "R009"]
  },
  {
    "name": "MissionController",
    "children_names": [],
    "related_requirement_ids":
      ["R002", "R006", "R007", "R008", "R009"]
  },
  {
    "name": "FlightTimer",
    "children_names": [],
    "related_requirement_ids": ["R002", "R007"]
  }
]
```

Listing 6: Model names, children, and requirement links from the reconstructed skeleton

I.3 Structured PlanTree

The component plans add constructor arguments, port payloads, timing responsibilities, external I/O, and routing to the skeleton. The hierarchy with these plans forms the `PlanTree` used for behavioral synthesis. Listing 7 shows selected fields from the controller and timer plans. The controller sends a phase label and duration to the timer; the timer returns that label when the duration has elapsed. For the surrounding Aircraft model, `assignment_in` receives the pallet and `ready_out` reports availability at initialization and after maintenance.

```
[
  {
    "class_name": "MissionController",
    "model_type": "atomic",
    "specification": {
      "input_ports": [
        {
          "name": "assignment_in",
          "type": "dict",
          "structure": "Pallet dict: ID, deadline, generation time.",
          "protocol": {
```

```
            "initial_signal": "None",
            "description":
              "Accept a pallet; begin flight with zero delay."
          }
        },
        {
          "name": "timer_done",
          "type": "str",
          "structure": "flight, unload, return or maintenance",
          "protocol": {
            "initial_signal": "None",
            "description": "Advance to the next phase."
          }
        }
      ],
      "output_ports": [
        {
          "name": "start_timer",
          "type": "dict",
          "structure": "{phase:str, duration:float}",
          "protocol": {
            "initial_signal": "None",
            "description":
              "Start the named phase for duration seconds."
          }
        },
        {
          "name": "ready_out",
          "type": "str",
          "structure": "str(aircraft_id); 1-based ID",
          "protocol": {
            "initial_signal": "Actively sends at t=0",
            "description": "Ready at t=0 and after maintenance."
          }
        }
      ],
      "external_io": [
        {
          "target": "stdout",
          "content": "Write aircraft and destination event JSONL."
        },
        {
          "target": "stderr",
          "content": "Write lifecycle diagnostics; never stdout."
        }
      ]
    },
    "coupling_rules": []
  },
  {
    "class_name": "FlightTimer",
    "model_type": "atomic",
    "specification": {
      "input_ports": [
        {
          "name": "start_timer",
          "type": "dict",
          "structure": "{phase:str, duration:float}",
          "protocol": {
            "initial_signal": "None",
            "description":
              "Start the named phase for duration seconds."
          }
        }
      ],
```

```
      "output_ports": [
        {
          "name": "timeout",
          "type": "str",
          "structure": "flight, unload, return or maintenance",
          "protocol": {
            "initial_signal": "None",
            "description": "Return phase after duration seconds."
          }
        }
      ],
      "external_io": [
        {
          "target": "stderr",
          "content": "Write lifecycle diagnostics; never stdout."
        }
      ]
    },
    "coupling_rules": []
  }
]
```

Listing 7: Port protocols and external I/O from the reconstructed component plans, abridged

The coupling rules below belong to the Aircraft plan. They connect the Aircraft's boundary ports to its children as well as connecting the two children to one another.

Here `controller` and `timer` are the child instance names, and unqualified ports belong to Aircraft. The four routes comprise an external input coupling (EIC), two internal couplings (IC), and an external output coupling (EOC).

```
Connect assignment_in to controller.assignment_in (EIC).
Connect controller.start_timer to timer.start_timer (IC).
Connect timer.timeout to controller.timer_done (IC).
Connect controller.ready_out to ready_out (EOC).
```

Listing 8: Boundary and internal routing in the reconstructed Aircraft plan

I.4 Implementation and Event Trace

For brevity, we only provide two representative scripts.

The Aircraft coupled model creates its MissionController and FlightTimer children, registers them, and connects their ports. Listing 9 shows the class declaration and complete constructor, including boundary couplings and the controller–timer connections.

```python
from xdevs.models import Atomic, Coupled, Port
from .Aircraft_libs.MissionController import MissionController
from .Aircraft_libs.FlightTimer import FlightTimer


class Aircraft(Coupled):

    def __init__(
        self,
        name: str,
        parent: Coupled | None,
        aircraft_id: int,
        flight_time: float,
        unload_time: float,
        return_time: float,
        maintenance_time: float
    ):
        """
        Args:
```

```
            name (str): The unique name of the model.
            parent (Coupled | None): The parent model.
            aircraft_id (int): Unique integer ID assigned to the aircraft.
            flight_time (float): Duration for the FLYING phase.
            unload_time (float): Duration for the UNLOADING phase.
            return_time (float): Duration for the RETURNING phase.
            maintenance_time (float): Duration for the MAINTENANCE phase.
        """
        super().__init__(name)
        self.parent = parent

        # Register Ports
        self.add_in_port(Port(dict, "assignment_in"))
        self.add_out_port(Port(str, "ready_out"))

        # Instantiate Components
        controller = MissionController(
            name="controller",
            parent=self,
            aircraft_id=aircraft_id,
            flight_time=flight_time,
            unload_time=unload_time,
            return_time=return_time,
            maintenance_time=maintenance_time
        )

        timer = FlightTimer(
            name="timer",
            parent=self,
            flight_time=flight_time,
            unload_time=unload_time,
            return_time=return_time,
            maintenance_time=maintenance_time
        )

        self.add_component(controller)
        self.add_component(timer)

        # Define Couplings

        # EIC: External Input -> Sub-model Input
        self.add_coupling(self.input["assignment_in"], controller.input["assignment_in"])

        # IC: Internal Sub-model Couplings
        # Controller commands Timer to start
        self.add_coupling(controller.output["start_timer"], timer.input["start_timer"])
        # Timer notifies Controller when duration elapses
        self.add_coupling(timer.output["timeout"], controller.input["timer_done"])

        # EOC: Sub-model Output -> External Output
        self.add_coupling(controller.output["ready_out"], self.output["ready_out"])
```

Listing 9: Port registration, child construction, and couplings in Aircraft

Listing 10 shows the FlightTimer class declaration, complete constructor, and initialization, transition, and output methods. It starts idle; an external request supplies a phase label and duration, and `hold_in` schedules the next internal transition. At that time, `lambdaf` emits the phase label and `deltint` returns the timer to idle.

```
import json
```

```python
import sys
from xdevs.models import Atomic, Coupled, Port
from devs_project.devs_utils.devs_context import get_current_time


class FlightTimer(Atomic):

    def __init__(self, name: str, parent: Coupled | None, flight_time: float, unload_time: float, return_time: float, maintenance_time: float):
        """
        Args:
            name (str): The unique name of the model.
            parent (Coupled | None): The parent coupled model.
            flight_time (float): Duration of outbound flight.
            unload_time (float): Duration of unloading.
            return_time (float): Duration of return flight.
            maintenance_time (float): Duration of maintenance.
        """
        super().__init__(name)
        self.parent = parent

        # Register Ports
        self.add_in_port(Port(dict, "start_timer"))
        self.add_out_port(Port(str, "timeout"))

        # Parameter Storage
        self.params = {
            "flight_time": flight_time,
            "unload_time": unload_time,
            "return_time": return_time,
            "maintenance_time": maintenance_time
        }

        # Internal State Variables
        self.active_phase_name = ""

        # Initial State
        self.hold_in("IDLE", float('inf'))

        print(json.dumps({
            "event": "Model Created", "params": self.params,
            "time": get_current_time()
        }), file=sys.stderr, flush=True)

    def initialize(self):
        """Initializes the model to IDLE state."""
        self.active_phase_name = ""
        self.hold_in("IDLE", float('inf'))
        print(json.dumps({
            "event": "Model Initialized", "time": get_current_time()
        }), file=sys.stderr, flush=True)

    def deltext(self, e: float):
        """
        Handles external timer requests.
        Receives a dictionary with 'phase' and 'duration'.
        """
        # Process all incoming timer requests (typically one per aircraft cycle step)
        for msg in self.input["start_timer"].values:
            phase_label = msg.get("phase", "unknown")
            duration = msg.get("duration", 0.0)

            # Update state to track what we are timing
```

```
            self.active_phase_name = phase_label
            # Schedule the timeout
            self.hold_in("BUSY", float(duration))

    def lambdaf(self):
        """Outputs the completed phase name when the timer expires."""
        if self.phase == "BUSY":
            self.output["timeout"].add(self.active_phase_name)

    def deltint(self):
        """Transitions back to IDLE after the timer expires."""
        if self.phase == "BUSY":
            self.active_phase_name = ""
            self.hold_in("IDLE", float('inf'))
        else:
            self.hold_in(self.phase, self.ta())
```

Listing 10: Class definition, construction, and lifecycle methods of FlightTimer

The *construction excerpt* used for parent generation is distinct from the illustrative code selections above. Listing 11 shows the output of the construction-excerpt extractor described in Section 3.2, applied to FlightTimer's source. The extractor retains the class declaration and complete constructor, including parameter storage, state initialization, and logging, but excludes the lifecycle methods. Together with FlightTimer's `ModelPlan`, this is the child context supplied when generating Aircraft; the constructor arguments and port names correspond to those used in Listing 9.

```
class FlightTimer(Atomic):
    def __init__(self, name: str, parent: Coupled | None, flight_time:
    float, unload_time: float, return_time: float, maintenance_time:
    float):
        """
        Args:
            name (str): The unique name of the model.
            parent (Coupled | None): The parent coupled model.
            flight_time (float): Duration of outbound flight.
            unload_time (float): Duration of unloading.
            return_time (float): Duration of return flight.
            maintenance_time (float): Duration of maintenance.
        """
        super().__init__(name)
        self.parent = parent

        # Register Ports
        self.add_in_port(Port(dict, "start_timer"))
        self.add_out_port(Port(str, "timeout"))

        # Parameter Storage
        self.params = {
            "flight_time": flight_time,
            "unload_time": unload_time,
            "return_time": return_time,
            "maintenance_time": maintenance_time
        }

        # Internal State Variables
        self.active_phase_name = ""

        # Initial State
        self.hold_in("IDLE", float('inf'))

        print(json.dumps({
            "event": "Model Created", "params": self.params,
            "time": get_current_time()
```

```
        }), file=sys.stderr, flush=True)
```

Listing 11: Construction excerpt extracted from FlightTimer

Listing 12 shows the first 26 records of the integration-test trace, through $t = 83$, using the configuration in Listing 5. All events in this interval are retained. Aircraft 1 departs with pallet 0 at $t = 0$, delivers it at $t = 38$, returns at $t = 73$, and completes maintenance and takes pallet 3 at $t = 83$. The `destination` records are emitted by MissionController, not by a separate Destination model.


```
{"time": 0, "entity": "facility", "event": "pallet_generated", "payload": {"pallet_id": 0, "expiration_time": 180.0}}
{"time": 0, "entity": "queue", "event": "pallet_queued", "payload": {"pallet_id": 0, "queue_size": 1}}
{"time": 0, "entity": "coordinator", "event": "assignment_created", "payload": {"aircraft_id": 1, "pallet_id": 0}}
{"time": 0, "entity": "aircraft", "event": "depart", "payload": {"aircraft_id": 1, "pallet_id": 0}}
{"time": 15.0, "entity": "facility", "event": "pallet_generated", "payload": {"pallet_id": 1, "expiration_time": 195.0}}
{"time": 15.0, "entity": "queue", "event": "pallet_queued", "payload": {"pallet_id": 1, "queue_size": 1}}
{"time": 15.0, "entity": "coordinator", "event": "assignment_created", "payload": {"aircraft_id": 2, "pallet_id": 1}}
{"time": 15.0, "entity": "aircraft", "event": "depart", "payload": {"aircraft_id": 2, "pallet_id": 1}}
{"time": 30.0, "entity": "facility", "event": "pallet_generated", "payload": {"pallet_id": 2, "expiration_time": 210.0}}
{"time": 30.0, "entity": "queue", "event": "pallet_queued", "payload": {"pallet_id": 2, "queue_size": 1}}
{"time": 30.0, "entity": "coordinator", "event": "assignment_created", "payload": {"aircraft_id": 3, "pallet_id": 2}}
{"time": 30.0, "entity": "aircraft", "event": "depart", "payload": {"aircraft_id": 3, "pallet_id": 2}}
{"time": 38.0, "entity": "destination", "event": "pallet_delivered", "payload": {"pallet_id": 0, "aircraft_id": 1, "latency": 38.0}}
{"time": 45.0, "entity": "facility", "event": "pallet_generated", "payload": {"pallet_id": 3, "expiration_time": 225.0}}
{"time": 45.0, "entity": "queue", "event": "pallet_queued", "payload": {"pallet_id": 3, "queue_size": 1}}
{"time": 53.0, "entity": "destination", "event": "pallet_delivered", "payload": {"pallet_id": 1, "aircraft_id": 2, "latency": 38.0}}
{"time": 60.0, "entity": "facility", "event": "pallet_generated", "payload": {"pallet_id": 4, "expiration_time": 240.0}}
{"time": 60.0, "entity": "queue", "event": "pallet_queued", "payload": {"pallet_id": 4, "queue_size": 2}}
{"time": 68.0, "entity": "destination", "event": "pallet_delivered", "payload": {"pallet_id": 2, "aircraft_id": 3, "latency": 38.0}}
{"time": 73.0, "entity": "aircraft", "event": "return", "payload": {"aircraft_id": 1}}
{"time": 73.0, "entity": "aircraft", "event": "maintenance_start", "payload": {"aircraft_id": 1}}
{"time": 75.0, "entity": "facility", "event": "pallet_generated", "payload": {"pallet_id": 5, "expiration_time": 255.0}}
{"time": 75.0, "entity": "queue", "event": "pallet_queued", "payload": {"pallet_id": 5, "queue_size": 3}}
{"time": 83.0, "entity": "aircraft", "event": "maintenance_end", "payload": {"aircraft_id": 1}}
{"time": 83.0, "entity": "coordinator", "event": "assignment_created", "payload": {"aircraft_id": 1, "pallet_id": 3}}
{"time": 83.0, "entity": "aircraft", "event": "depart", "payload": {"aircraft_id": 1, "pallet_id": 3}}
```


Listing 12: Beginning of the SA integration-test trace, through the first completed aircraft cycle

I.5 Connection to Visualization

The same trace can drive a scene containing a loading facility, queued pallets, aircraft, and a destination. Assignment associates a pallet with an aircraft, departure starts outbound motion, delivery marks unloading completion, and return marks arrival back at the facility. The frontend can interpolate motion between these events using the configured phase durations; geometry and continuous trajectories are not part of the DEVS trace.

This demonstration uses timer-driven execution and does not implement physical feedback. The lower part of Figure 2 depicts the proposed extension from Section 3.3. Closing that loop would require time-advancement and event-exchange interfaces, with physical completion replacing the corresponding timer trigger rather than advancing the same task twice.

J Checker Audit

The evaluator first checks execution and trace-schema validity for each test case. Behavioral requirements are then scored at either case or collection scope, with operationally invalid cases receiving no credit. Table 8 counts the requirements declared in the ten benchmark manifests; operational gating is shared evaluator logic and is not counted as a separate set of per-scenario behavioral rules.

Checker construction and validation. The checkers were constructed from the completed specifications rather than from the implementations being evaluated. Each stated behavioral requirement was translated into a rule over observable trace evidence, and the associated test cases were selected to exercise both nominal behavior and specification-relevant boundary conditions. During development, we ran the rules on representative conforming and erroneous traces and manually checked that conforming behavior received credit while targeted violations reduced the corresponding rule scores.

Table 8: Behavioral requirement counts in the benchmark manifests.

Scenario	Micro	Macro
ABP	5	2
IOBS	5	3
OTrain	7	2
StratAirlift	5	1
SEIRD	5	1
Barbershop	3	1
FileTransfer	6	1
HouseHeating	4	1
SAA	4	2
StoreCashier	6	2
Total	**50**	**16**

For example, the SEIRD checker has five case-scoped micro requirements for compartment-level dynamics and one macro requirement for aggregate behavior. Four requirements across the benchmark are collection-scoped; the others are case-scoped. Every rule is evaluated over the full test suite. A collection-scoped score combines its quality on operationally valid cases with the fraction of all test cases that are valid, as specified in Section 4.3. For the zero-population SEIRD case, the mortality-partition rule checks that the simulator does not create recovered or deceased population; the mortality ratio is evaluated only when the reference dynamics contain resolved infective outflow.

Table 9: Representative mapping from IOBS task features to current evaluator requirements. Names in the middle column are manifest requirement identifiers.

Task feature	Evaluator requirement	Trace evidence used
Input and AAM routing	`iobs.input_aam_routing`	Matches input records to configured requests, and checks AAM-generated and logout events at the corresponding sequential-server completion times.
ANV verification	`iobs.anv_verification`	Matches ANV records to AAM outputs, including complementary pass/fail flags and the ten-second stage delay.
Password retries	`iobs.password_retry`	Matches successful PV records to ANV passes; each must report a positive integer attempt count at the expected time.
Bill generation	`iobs.bill_generation`	Matches BPM bills to PV completions and checks integer amounts in the specified range.
Transaction state	`iobs.transaction_state`	Replays balance and transaction count from the initial state while matching TPM records to bill events.
Cross-stage flow	`iobs.flow_conservation`	Compares event counts at adjacent pipeline stages and against configured request totals.
ANV pass probability	`iobs.anv_probability`	Scores the distance of the observed pass rate from 0.5 over groups of valid test cases.
PV success probability	`iobs.password_probability`	Scores the observed success-per-attempt ratio over groups of valid test cases.

The mapping illustrates how declared behavior is evaluated from observable traces. It does not inspect every internal implementation choice; in particular, collection-scoped probability rules assess realized rates within the configured tolerance rather than identifying the simulator's underlying random process.

K Downstream Planning Protocol and Analysis

This appendix documents the task interface, policy-generation settings and evaluation protocol for the exploratory study in Sec. 5.5.

Tasks and policy interface. The scenarios cover a single-product tree network, a two-product network with a shared warehouse, and a bill-of-materials assembly and distribution network. Each task provides a natural-language description of a supply-chain network, including its topology, demand process, lead times, holding and stockout costs, and the required Python policy interface. On each simulated day, the evaluator calls a mounted policy function with the current inventory position of a node. The function returns replenishment orders to permitted upstream nodes, and those orders affect costs after routing, production, and transportation delays. The generated policy file contains the replenishment functions and a `POLICY_MOUNTS` dictionary that assigns them to node groups. Table 10 summarizes the three tasks.

Table 10: Supply-chain planning scenarios used in the downstream study.

Scenario	Topology	Products	Main decision challenge
01	9-node branching tree	1	Multi-echelon replenishment coordination
02	6-node shared-warehouse network	2	Coordinating inventories for multiple products
03	8-node assembly and distribution network	5	Coordinating assembly and downstream distribution

Policy-generation settings. All three methods receive the same task description. Direct-Policy uses one LLM call to produce a policy from the specification. Native-Python uses a coding agent for at most 50 steps and may create and execute its own analytical calculations or supply-chain approximations. DEVS-Augmented uses at most 30 steps, including generation of the DEVS simulator, and provides an interface for loading candidate policies and obtaining holding, stockout, and total costs. We designed the budget to be large enough for an LLM to explore and optimize.

Evaluation protocol. Each generated policy is loaded into the same Stockpyl 1.0.2 evaluator and simulated for 100 periods with seed 42. Total cost is the sum of holding and stockout costs. To present costs from scenarios with very different scales in one table, we divide each cost by the lowest cost observed in that scenario; $1.00\times$ therefore denotes the best observed result. The normalization denominators in Table 2 are the lowest absolute totals observed in each scenario: 7.1367×10^4, 2.4136×10^6, and 1.4451×10^8 for Scenarios 01–03, respectively.

Results. Table 2 shows that DEVS-Augmented obtains the lowest cost in all three scenarios with GPT-5.2. DEVS-Augmented achieves lower replenishment costs than Direct-Policy in all three scenarios with both backbones. With GPT-5.2, it also outperforms Native-Python in all three scenarios. With GLM-4.7, it outperforms Native-Python in Scenario 02, while Native-Python achieves lower costs in Scenarios 01 and 03. These results suggest that simulator-supported reasoning can improve downstream decisions, while its effectiveness depends on the underlying model and task.

Cost decomposition. We inspect the final policy code and independently evaluated cost components for Scenario 01, where the GPT-5.2 cost difference is largest. The GPT-5.2 DEVS-Augmented policy incurs a holding cost of 5.60×10^4 and a stockout cost of 1.54×10^4. Direct-Policy and Native-Python incur stockout costs of 4.62×10^6 and 6.40×10^6, respectively. Both policies search the input dictionary for a retailer identifier that the specified interface does not provide and consequently send every retailer order to `Regional_DC_0`. The DEVS-Augmented policy preserves the two network branches by routing the first three retailers to `Regional_DC_0` and the remaining three to `Regional_DC_1`. By preserving the two-branch routing, the DEVS-Augmented policy achieves a normalized cost of $1.00\times$, compared with $66.0\times$ for Direct-Policy and $90.7\times$ for Native-Python. In Scenario 03, the same backbone shows another large reduction in stockout cost: the stockout costs are approximately 7.60×10^9 for Direct-Policy, 7.48×10^9 for Native-Python, and 1.44×10^8 for DEVS-Augmented.

L Qualitative Analysis: Failure Mechanisms

The cases below illustrate failure mechanisms rather than their relative frequency across methods.

The score distribution for DEVS-GEN contains two distinct failure regimes. Among its 120 generations, 74 pass every operational test case, 13 pass only a subset, and 33 produce no operationally valid case. For the 87 generations with at least one valid case, the median behavioral score is 0.936 and the lower quartile is 0.645. Thus, some generated projects fail at the boundary between the code and the evaluation environment, whereas others execute reliably but implement only part of the requested dynamics. We discuss these DEVS-GEN failures first, then contrast them with failures observed in the Standard and Lite baselines. The two single-pass settings are ablations of the generation procedure and are not used to characterize the baseline agents here.

Operational failures in DEVS-GEN. These cases illustrate two behavioral failure modes: missing required workflow behavior and inaccurate local state evolution. A narrow implementation error can invalidate an otherwise substantial generated project. In the second Llama 4 Scout generation for IOBS, the top-level model imports `get_current_time` from `xdevs.models`, where that symbol does not exist. The first Llama 4 Scout generation for OTrain reaches the runner but calls the nonexistent `sys.time_ns()`. The first GLM-4.7 generation for FileTransfer prints records, but its initial `control_cmd` records omit `total_remaining`, and several cases also contain decreasing timestamps. In all three DEVS-GEN failures, the generated component hierarchy is present, but an error in the code, runner, or trace format prevents valid evaluation. Deterministic scaffolding for runtime imports and runners, together with schema-aware trace checks, directly targets this class of failure.

Behavioral failures in DEVS-GEN. Other generations execute on every test case yet implement an incomplete or inaccurate process. The second Qwen3-Coder generation for IOBS produces operational traces but no valid password-verification attempts, resulting in a behavioral score of 0.356. Here, the required verification pathway has not survived the requirement–plan–code chain. The first Llama 4 Scout generation for HouseHeating instead exposes all required thermal signals, but its temperature and heat-loss trajectories do not match the requested dynamics closely enough, yielding 0.882. The event pathway exists in this case; the error lies in the local state evolution. These examples identify two separate weaknesses: an early omission can propagate through otherwise consistent component contracts, while a sufficient plan can still be realized with an incorrect API call or transition equation. Interface consistency alone therefore does not establish behavioral completeness.

Failures in the Standard baselines. Execution-enabled agents can produce programs that run successfully while failing to realize required scenario behavior. In the second Qwen3-Coder generation by OpenHands for StratAirlift, the program runs on every case but never creates the scheduled pallets that should activate the queue, producing a behavioral score of 0.200. In the first Llama 4 Scout generation by SWE-Agent for OTrain, the output is operational but contains no train-arrival events, leaving the behavioral score at 0.478. These are not primarily launch failures. They show that execution feedback can establish that a program runs without revealing that a required producer, trigger, route, or observable consequence is missing. These failures affect the simulated process rather than merely program execution: operationally valid traces can still omit the events needed to represent the required workflow.

Failures in the Lite baselines. Without execution feedback, both uncontrolled runtime behavior and missing domain events can survive into the final project. The first Llama 4 Scout generation by OpenHands-Lite for StoreCashier emits about 181K lines in each test before the output guard terminates it. The third Llama 4 Scout generation by SWE-Agent-Lite for ABP runs, but omits the packet-position and packet-retrieval events needed to replay the protocol, producing a behavioral score of 0.164. The Lite examples exhibit failures at both levels: uncontrolled output prevents valid execution, while missing protocol events leave an executable program behaviorally incomplete under the specification.

Across the examined cases, failures range from execution and interface errors to missing workflow behavior and incorrect state evolution. Execution feedback alone does not ensure behavioral conformance, while explicit component plans do not guarantee that all specified behavior survives implementation.

M LIMITATIONS AND FUTURE WORK

The current study has four main boundaries.

Environment Dynamics. DEVS-GEN targets discrete-event environments such as supply chains, service operations, and logical workflows. The SEIRD scenario represents continuous dynamics through fixed-step Euler updates inside a discrete-event execution; the current system does not generate continuous-time solvers. A next step is to further optimize and evaluate the physics-engine coupling outlined in Section 3.3, extending the generated models to environments where physical outcomes affect task progression.

Generation Reliability. DEVS-GEN depends on the backbone model implementing output, interface, and DEVS-specific requirements correctly. App. L shows both localized integration errors and behavioral errors in the requirement–plan–code–trace chain. More systematic plan review and behavior-directed repair may reduce these failures, but their effects require controlled evaluation.

Specification and Checker Scope. The benchmark contains ten scenarios described by completed, checkable specifications. It therefore evaluates specification-to-simulator generation. The checker observes schema validity, events, state transitions, cross-component flows, timing, and collection-level behavior through structured traces. It does not establish correctness for behavior left unconstrained by the specification or behavior that is not observable through that interface.

Empirical Scope and Future Work. Although DEVS-GEN achieves higher average behavioral scores, its differences from the execution-enabled Standard agents are not statistically significant on the current benchmark. The direct-generation comparison between single and single-xdevs changes both representation and simulation API, while the downstream study contains only three exploratory tasks. Larger scenario collections, PlanTree-conditioned coding-agent baselines, and a baseline that follows DEVS without requiring `xdevs.py` would better separate the effects of staged planning, the DEVS formalism, and the concrete xdevs.py implementation.