



EF-Builder

Tutorial for New Users

Design DEVS Experimental Frames from plain-language questions — grounded in the EFSUT methodology (Zeigler, 2026), delivered as compilable MS4 Systems Java code.

An RTSync Commercial Product

What Is EF-Builder?

From stakeholder question to running experiment — in one guided workflow

EF-Builder is RTSync’s AI-assisted environment for designing DEVS Experimental Frames. You describe your system and the questions you need answered; EF-Builder analyzes them, proposes candidate frames, and generates a complete, compilable test harness for the MS4 Systems framework — with every artifact traceable back to the question that motivated it.



EF Analysis Report

Full conceptual record: questions, candidate frames, partial order, best-model table, your selection rationale



EFSUT Table

One row per stakeholder question: Question → Frame → Model → Objectives



System Model (SUT)

Self-contained atomic DEVS model in Java, covering all ports your frames need



EF Java Files

Generator, Transducer, and Coupled EF per selected frame — compile-ready with a built-in test case

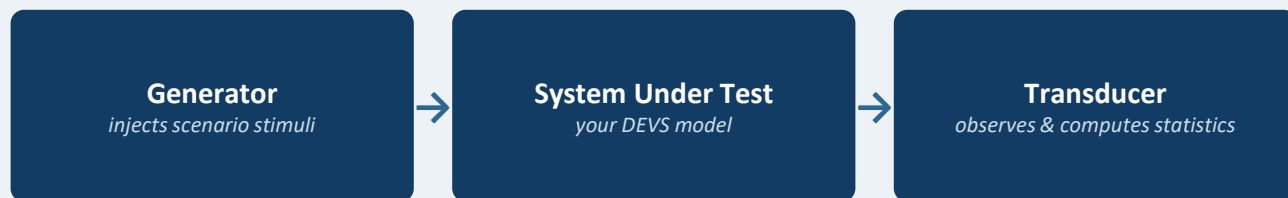
The Method Behind It: EFSUT in 60 Seconds

Specify what the simulation must answer — before any model is built



Every experiment starts from a stakeholder question. The Experimental Frame (EF) makes the experiment explicit — what to inject, what to observe, when to stop — so results are traceable, auditable, and defensible.

ANATOMY OF A GENERATED FRAME



stop signal — the frame ends its own experiment

THREE FORMAL RELATIONS

Derivability — coarser frames are derivable from finer ones (selection, aggregation, coarsening)

Applicability — a frame fits a model when the model has the ports and behaviors it needs

Morphism — models relate across abstraction levels, justifying the simplest sufficient model

How Experimental Frames Are Generated

Two guided phases — you stay in control at every checkpoint



Conceptual Phase (C1–C5)

- C1 — Analyzes and prioritizes your stakeholder questions
- C2 — Generates candidate experimental frames
- C3 — Builds the derivability partial order and best-model table
- C4 — Presents all candidates with coverage and complexity
- C5 — You select the frame(s) to develop



Developmental Phase (D1–D5)

- D1 — Designs the Java class structure
- D2 / D3 — Generates Generator and Transducer
- D3B — Generates the self-contained System Model
- D4 — Wires the Coupled EF with a built-in test case
- D5 — Validates structure and writes the files

HOW MANY FRAMES WILL I GET?

The frame count depends on two factors you control through your prompt: **how many candidates the skill generates** (driven by your system description and questions) and **how many you select at step C5**. You can **explicitly** specify the number of frames that you want.

Prompt Structure

Everything after the command becomes the ARGUMENTS the skill works from

```
/ef-builder <ARGUMENTS>
```

Your ARGUMENTS should contain three elements:

Element	Purpose	Effect on Frame Count
System description	What the system does, its components, inputs, outputs	More components and ports yield more candidate frames
Stakeholder questions	What you want the experiment to answer	Each distinct question category can produce a separate frame
Frame count directive	Explicit instruction on how many frames to generate	Overrides the default behavior of generating multiple candidates



Question categories the skill recognizes: performance, resilience, behavioral, parametric, comparative, and boundary. Phrasing questions in distinct categories is the most reliable way to broaden — or focus — the candidate set.

Controlling Frame Count — Strategy 1: Single Frame

Restrict the skill to exactly one comprehensive experimental frame

Use explicit language in your prompt:

```
"Generate a single experimental frame"  
"Build only one EF"  
"Restrict to one frame that addresses all questions"  
"Generate one comprehensive frame covering all aspects"
```

When you request a single frame, the skill will:

- Still analyze stakeholder questions (C1)
- Generate one comprehensive candidate frame covering all questions (C2)
- Skip the partial-order analysis — only one frame exists (C3)
- Present the single frame for confirmation (C4–C5)
- Proceed directly to code generation (D1–D5)



When a single frame is the right choice

- You have one clear question (or several that share the same ports and observations)
- You want the fastest path to a runnable test harness
- You are **learning** the tool and want a small, readable output set
- The system is simple — few ports, few states — so slicing it further adds no insight
- You **plan to broaden later**: a comprehensive frame can be coarsened into derived frames afterwards

Controlling Frame Count — Strategy 2: Multiple Frames

Let the skill propose a portfolio, then choose at checkpoint C5

Invite a broader candidate set:

```
"Generate candidate frames for each question"  
"Propose separate frames for throughput and for  
failure behavior"  
"Show me the full candidate set before I choose"
```

What the skill adds when candidates grow in number:

- A derivability partial order — which frames' data can be computed from which (C3)
- A best-model table — the minimally sufficient model per frame
- An applicability matrix rating every frame–model pair



Frame types you'll see proposed

Full-scope — all ports; comprehensive characterization

Single-concern — one question, minimal ports

Stress — inputs at extremes; boundary and failure behavior

Comparative — same ports, different scenario sets and/or models (A/B)

Focused-output — isolate one observable behavior

Rule of thumb: one frame per distinct question category, plus one full-scope frame if you need system-wide characterization.

Worked Example 1: The Grocery Store

A complete run from one plain-language prompt



The prompt (exactly as typed):

```
/ef-builder Generate a single frame for a customer arriving at a
small grocery store, doing some shopping in the items section,
then paying at the counter. There is never any congestion in the
items section and only one or two people waiting to pay.
How long is the average time in the store?
```

Why this prompt works — it contains all three elements:

System description

“customer arrives → shops in items section → pays at counter” — components, flow, and load assumptions

Stakeholder question

“How long is the average time in the store?” — a performance-category question

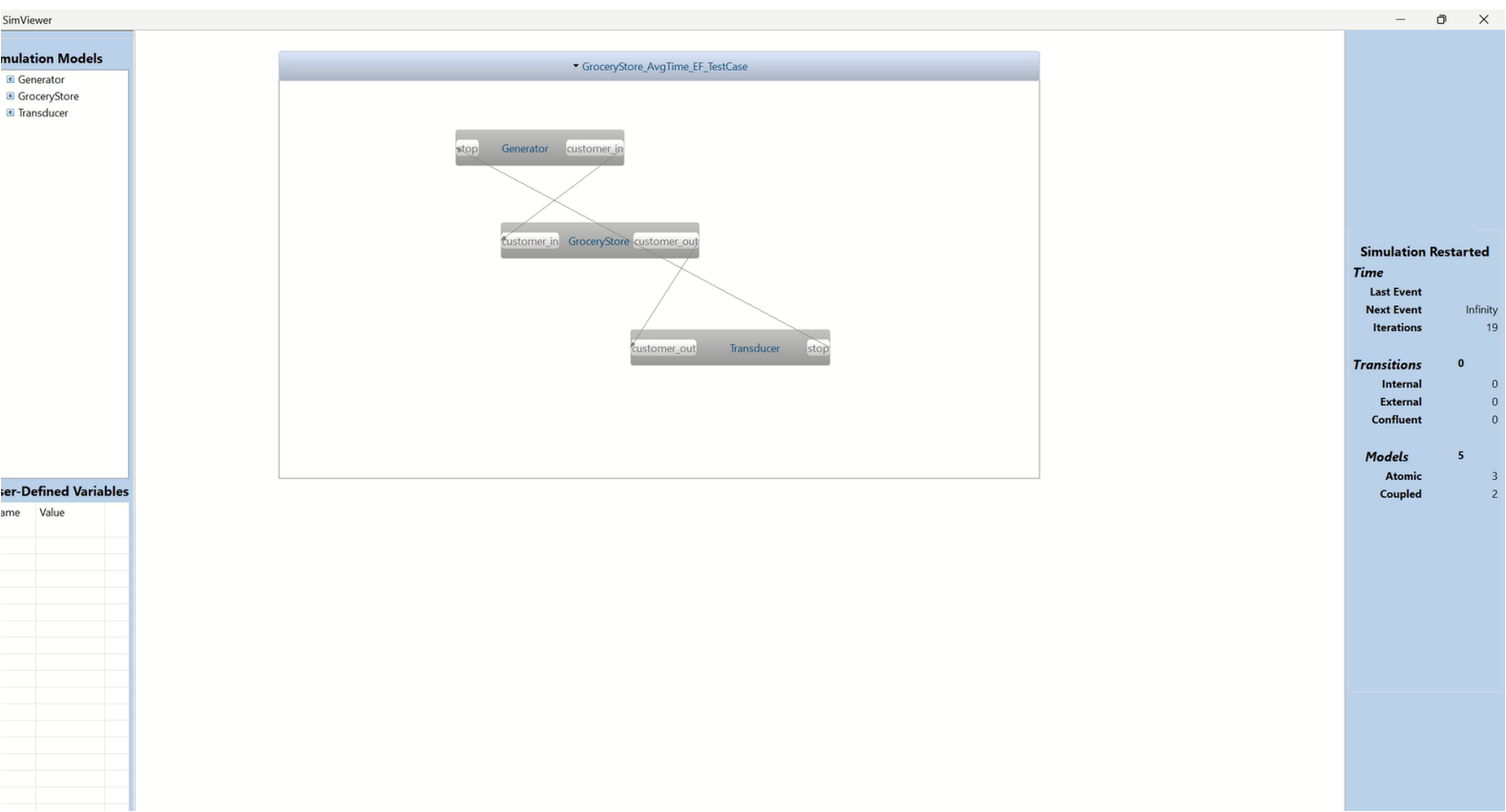
Frame count directive

“Generate a single frame” — Strategy 1, so C3 is skipped and one frame goes to code

Elapsed time from prompt to compiled result: about 5 minutes.

Example 1 Result: The Generated Experiment

GroceryStore_AvgTime_EF — assembled, wired, and simulation-ready



Files generated: GroceryStoreModel.java, GroceryStore_AvgTime_EFGenerator.java, GroceryStore_AvgTime_EFTransducer.java, GroceryStore_AvgTime_EF.java + EF Analysis Report and EFSUT Table

What you’re seeing

- SimViewer displaying the coupled model GroceryStore_AvgTime_EF_TestCase
- Generator injects customer_in arrivals into the GroceryStore SUT
- Transducer receives customer_out events and computes average time in store
- The stop coupling from Transducer back to Generator ends the run — the frame is fully self-contained
- 5 models total: 3 atomic + 2 coupled, exactly as the $N + M + 1$ coupling rule prescribes

Watch It Run

The grocery-store frame executing in SimViewer (embedded video)



Click to play • The Generator emits customer arrivals; watch transition counts advance and the Transducer's summary report the average time in store at the stop event.

Built-In Quality Gates

EF-Builder verifies its own work before you ever compile



Conceptual Verification (CV1–CV5)

Runs before any code is written

- Constructs a Maximal EF — the union of all candidate frames' ports and scenarios
- Audits that every frame port exists on your system (orphans flagged)
- Confirms the scenarios can reach every declared system state
- Proves each frame is derivable from the Maximal EF



Developmental Verification (DV1–DV6)

Runs after code generation

- Every accessor call in the EF resolves to a real System Model method
- No dead output ports — each port is emitted in at least one state
- Coupling counts match the $N + M + 1$ rule exactly
- Scenario parameters provably drive the SUT into its target states

If a gate fails, EF-Builder reports exactly what to fix — usually the system description or a scenario — and re-runs the audit.

Prompting Best Practices

Small changes in the prompt make large differences in the result

DO

- Name your components, ports, and states explicitly — port names flow directly into the Java code
- State load assumptions (“never any congestion”, “one or two waiting”) — they shape scenarios
- Phrase each question in one category: performance, resilience, behavioral, parametric, comparative, boundary
- Give a frame-count directive every time — don’t leave it to the default
- Review the C5 checkpoint carefully — it is your last stop before code generation
- Run the java code in MS4 Me to get results - each selected frame produces 4 Java files

AVOID

- Don’t bury multiple unrelated systems in one prompt — run the skill once per system
- Don’t ask questions your ports can’t observe — applicability will fail
- Don’t rename ports between runs — names are case-sensitive throughout the generated code
- Don’t skip reading the EF Analysis Report — it records why each frame exists

EF-Builder in the RTSync Environment

One skill in a connected model-engineering toolchain



devs-builder

Generates the System Under Test itself — DEVS atomic and coupled models built directly from EF-Builder's output files



ses-builder

Organizes model families as System Entity Structure hierarchies for variant management



MS4 Systems

The execution environment: compile the generated Java, run in SimViewer, or headless via the built-in test case



RTSync support

Commercial licensing, onboarding workshops, and methodology consulting from the team behind the EFSUT approach

Typical workflow: formulate questions → derive frames and the minimal model (ef-builder) → generate or refine the SUT (devs-builder) → organize variants (ses-builder) → execute and iterate (MS4 Systems)



Your First Frame in Four Steps

1

Describe the system of interest in a few sentences (subsequent interaction can add more specifics) or upload accurate documentation if you have it

2

Ask one question and add “generate a single frame”

3

Review and confirm the candidate at checkpoint C5

4

Compile the generated Java in MS4 Systems and run the files(s) having a main() routine

Then try Strategy 2: re-run with three questions in different categories and explore the candidate portfolio.