HiroSim: A Modular, Deterministic Simulation Stack for Closed-Loop Control-Software Development

Massimo Di Pierro

2026 — technical white paper

Abstract

HiroSim is a modular, deterministic, closed-loop simulation stack for developing and validating control software for aerospace, robotics, and industrial vehicles. It couples an event-driven physics simulator with nanosecond-precision time, pluggable multi-rate models, and multiple numerical integrators to a flight computer written in a restricted Python dialect that compiles to native code, a telemetry pipeline, and a live browser dashboard, all wired together over the Zenoh transport. A hardware abstraction layer (HAL) lets the *same* compiled flight computer run against the simulator or against real sensors and actuators, so a controller can be carried along a software-in-the-loop (SIL) to hardware-in-the-loop (HIL) ladder without rewriting it. This white paper gives an overview of the stack: its main components, the design decisions that shape it (determinism, locality, polyglot plugins, a single-source build, and a compiled control DSL), its telemetry and visualization path, and its machine- learning and AI capabilities — including gray-box differentiable-filter models for engine and aerodynamic loads, and a deliberate design for authorability by large language models.

Keywords: closed-loop simulation; control software; hardware-in-the-loop; deterministic scheduling; telemetry; gray-box modeling; AI-assisted engineering.

1 Introduction

Developing a vehicle controller and validating it before flight is a recurring problem in aerospace and robotics. The controller must be exercised against a faithful model of the vehicle and its environment, observed and commanded interactively, and — ideally — carried onto real hardware without being rewritten. HiroSim addresses this end to end. It bundles a deterministic event-driven simulator, a flight computer authored in a restricted Python dialect that compiles to native code, a telemetry pipeline, and a live browser dashboard, connected over a common message bus so that the same flight-computer binary that flies the simulation can fly real hardware.

The guiding idea is a single artifact moving along a validation ladder. Figure 1 shows the user's view: a model and a control algorithm are authored, compiled, run in the simulator, and observed and commanded through the dashboard; the validated flight computer then moves onto a vehicle. Everything downstream of authoring is deterministic and reproducible, so a run can be replayed, snapshotted, and resumed exactly.

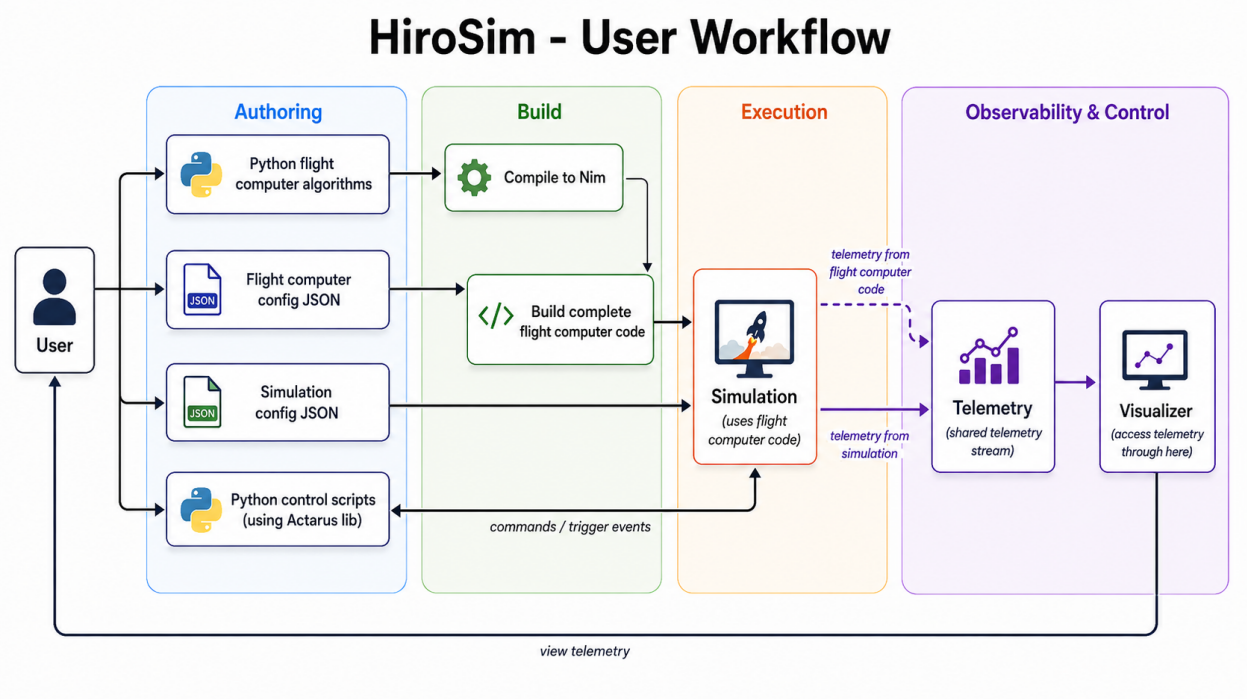

Figure 1: The HiroSim user workflow. Vehicle models, a flight-computer algorithm, and scenario configuration are authored, compiled to native code, executed in the simulator, and observed and commanded through the telemetry stack. The same flight-computer binary later moves onto real hardware.

2 Architecture and components

HiroSim is a stack of cooperating processes rather than a monolith, which keeps each concern testable in isolation and lets pieces be replaced or run remotely. Table 1 lists the main components. The simulator is the core; the telemetry receiver, service, and visualizer form the observability path; the onboard SDK (the flight-computer compiler and runtime) and the hardware abstraction layer form the control-software path; and a Python client library provides programmatic control. Two components are early and *experimental*: the hardware abstraction layer (FCSL-HAL), and an HLA adapter — a standalone Zenoh↔HLA bridge federate that lets a HiroSim simulation join a distributed High Level Architecture (IEEE 1516) federation and share state with other federates, without any HLA code in the simulator itself.

The pieces are deliberately decoupled by message passing. The simulator emits telemetry as UDP datagrams; the receiver compacts them to disk; the visualizer reads from disk and streams to the browser over WebSocket. Flight computers exchange data with the simulator over Zenoh, the same transport used for hardware-in-the-loop. This separation means the simulator does not depend on the existence of a dashboard, a dashboard does not depend on a live simulator, and a flight computer does not depend on running inside the simulator process.

3 The simulator core

The simulator is built from five concepts: *models*, the *DataStore*, the *scheduler*, the *clock*, and *integrators*.

Table 1: Main components of the HiroSim stack.

Component	Purpose
Simulator	Event-driven engine: model plugins, services, integrators, sensors/actuators, save/restore, telemetry emission.
Telemetry receiver	UDP ingest and row-to-column compaction into NumPy windows.
Telemetry service	HTTP supervisor that starts/stops receivers and allocates ports.
Telemetry visualizer	Browser dashboard: plots, 3D views, strings, commands.
Onboard SDK / FCSL	Restricted-Python flight-computer compiler and runtime.
FCSL-HAL *(experimental)*	Hardware abstraction layer: one API, swappable real or simulated I/O.
HLA adapter *(experimental)*	Standalone Zenoh↔HLA bridge federate: shares state with an HLA (IEEE 1516) federation via Portico, with no HLA code in the simulator.
simclient	Python sender, reader, command twins, async control surfaces.

Models. A model is a component with a small lifecycle — `init`, `wire`, `advance`, and optionally `getDerivatives` — that the scheduler invokes each tick. Models do not reference each other directly; they read and write named variables in the DataStore. Almost everything (physics bodies, sensors, actuators, controllers, I/O endpoints) is a model. Crucially, models are *plugins*: each model type lives in a shared library that the runner loads at startup, so the core binary stays small and the model set is open.

DataStore. The DataStore is a typed, double-buffered variable store. During a tick, models read the committed state and write the next state; after all models advance, the next state is committed atomically. This gives a clean, order-independent data-flow semantics and is also the unit of telemetry and of save/restore.

Scheduler and clock. Models are grouped by execution order and advanced in parallel within a group via a thread pool, then committed. Each model may run at its own update rate, so a fast inner control loop and a slower outer process coexist in one schedule. Time is an `int64` nanosecond counter: integrator steps, scheduler ticks, and telemetry timestamps share one monotonic origin with no floating-point drift. The runner can pace to wall-clock time (for hardware-in-the-loop and live viewing), run as fast as the host allows (for batch sweeps), or play back at a configured rate.

Integrators. Numerical integration is itself pluggable and selected per model from configuration: explicit Euler, second- and fourth-order Runge–Kutta (RK2, RK4), the fourth-order Adams–Bashforth (AB4) multi-step method, velocity Verlet (symplectic), and adaptive Runge–Kutta–Fehlberg (RK45) are provided and checked against closed-form solutions.

Determinism, commanding, and save/restore. Runs are deterministic: per-model pseudo-random number streams are seeded reproducibly, so the same configuration produces the same trajectory. A JSON-over-TCP variable server lets external clients `get`/`set`/`eval` any variable, pause and resume the clock, and trigger named events while the simulation runs. Any running scenario can be snapshotted to a JSON blob and resumed later, including across out-of-process flight computers.

Polyglot plugins. A plugin is any shared library that exports one entry point through the C application binary interface (ABI), loaded with `dlopen` at runtime; no language runtime is forced

Table 2: Run modes along the software-in-the-loop to hardware-in-the-loop ladder.

Mode	Flight computer runs as	I/O path
In-process SIL	a simulator plugin	direct simulation variables
SIL over Zenoh	its own flight binary	per-channel scalars over Zenoh
Virtual HIL	its own process	FCSL-HAL API, fed synthetically over Zenoh
On-hardware	its own process	FCSL-HAL API on real drivers

on it. Models can therefore be written in Nim, C, C++, Rust, or any language with a stable C ABI, which lets teams bring existing code into the simulation unchanged.

4 Control software and run modes

The flight computer is written in FCSL (Flight-Computer Specification Language), a restricted, Python-like domain-specific language (DSL), and compiled to native code rather than interpreted, so the control logic that is validated is the control logic that runs. Beneath it, the (experimental) hardware abstraction layer (FCSL-HAL) exposes one sensor and actuator interface whose backend is swappable between real Linux drivers and a simulated backend fed over Zenoh.

Together these let the *same* flight computer be exercised at several points along the standard validation ladder, distinguished by what is real at each layer — the flight-computer process, the transport, the I/O path, and the environment (Figure 2, Table 2). The progression runs from a fully in-process, fully deterministic co-simulation, through the real flight binary running as its own process over the flight transport, to that binary driving the hardware abstraction layer with synthetic I/O, and finally to the same code on real hardware. Because only the lowest layers change between steps, problems are localized: process isolation and wire timing are validated before the hardware seam, and the hardware seam is validated before physical hardware.

The hardware abstraction layer. FCSL-HAL is the seam that makes the same-binary promise concrete. It presents the flight computer with a single, narrow API for its sensors and actuators — and owns the clock and structured logging — behind which the implementation is swappable. A default backend binds to real Linux device interfaces: the Industrial I/O subsystem (IIO) for inertial and analog sensors, pulse-width modulation (PWM) for actuators, `evdev` for input devices, and `sysfs` for general-purpose I/O. A simulation backend serves the *identical* API from values carried over Zenoh. Because the flight-computer code is written against the API and never names a backend, the exact sensor-read and actuator-write code path that will later drive physical drivers is the one already exercised in virtual hardware-in-the-loop. This is what distinguishes mode 3 from ordinary software-in-the-loop — the abstraction boundary itself is under test, not merely the controller — and it lets a board bring-up reuse, unchanged, the controller that was validated in simulation. The layer is currently experimental, and its API and wire format may still change.

5 A minimal scenario

A scenario is two declarative files. The control software is an FCSL specification (Listing 1): one algorithm, one mode, running at 100 Hz, compiled to a native model type named after the file. The simulation (Listing 2) assembles a one-degree-of-freedom vehicle, an RK4 integrator, and that controller, mapping simulation variables to the controller's inputs and its output back to a command

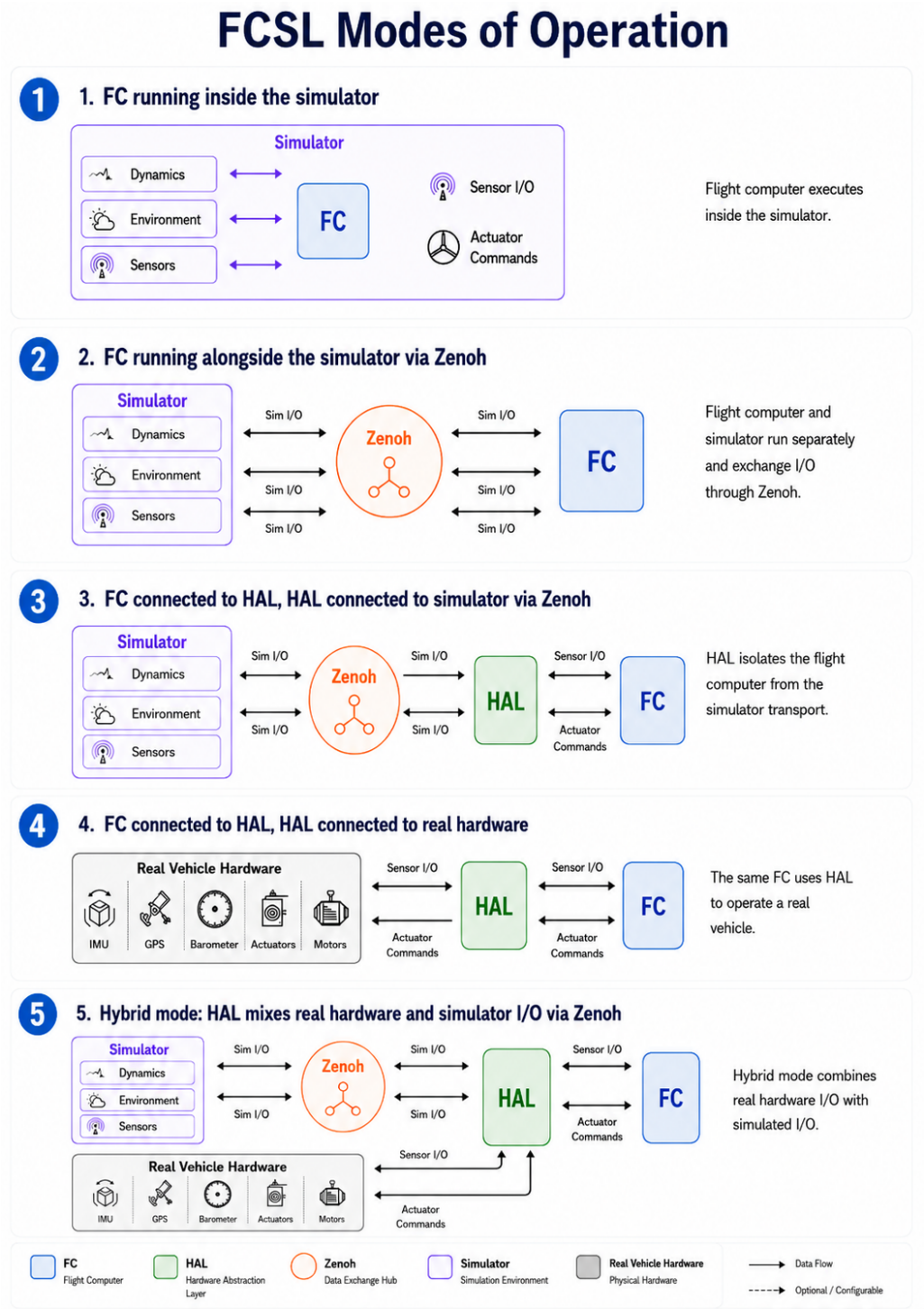

Figure 2: FCSL modes of operation. The same flight computer moves from in-process co-simulation (1), to its own process over Zenoh (2), to driving the hardware abstraction layer with simulated I/O (3, virtual hardware-in-the-loop), to real hardware (4), with a hybrid configuration (5) mixing real and simulated I/O.

channel. Nothing else is required beyond the algorithm body the FCSL file references. (JSON has no native comment syntax; the annotations are for exposition only.)

Listing 1: Minimal FCSL control-software specification (`hold.json`).

```
{
  "tick_hz": 100,                  // control-loop rate (Hz)
  "initial_state": "HOLD",         // starting mode
  "algorithms": {                  // algorithm modules (Python source)
    "Hold": { "source": "algorithms/hold.py" }
  },
  "instances": {                   // parameterized instances of algorithms
    "ctl": { "algorithm": "Hold",
             "parameters": { "target_z": 40.0, "kp": 0.5 } }
  },
  "states": {                      // per-mode schedule: instance -> rate (Hz)
    "HOLD": { "schedule": { "ctl": 100 } }
  }
}
```

Listing 2: Minimal simulation specification (`sim.json`) wiring a vehicle, an integrator, and the controller above.

```
{
  "duration": 20.0,                // run length (s)
  "clock_mode": "fast",            // "realtime" | "fast" | playback rate
  "paused_at_start": false,        // begin running immediately
  "telemetry_port": 9000,          // UDP destination for telemetry
  "status_vars": ["vehicle.z", "cmd.throttle"],  // console summary
  "plugins": [                     // model types to load at startup (dlopen)
    "plugins/libcore_plugin.so",
    "plugins/libexamples_plugin.so",
    "plugins/libfcsl_plugin.so"    // provides the "hold" controller type
  ],
  "models": [
    { "type": "ball", "name": "vehicle",   // a one-DoF point mass
      "integrator": "integ",               // evolved by "integ"
      "config": { "mass": 15.0, "z": 0.0 } },
    { "type": "rk4", "name": "integ",      // RK4 integrator
      "depends_on": ["vehicle"],
      "config": { "dt": 0.001 } },
    { "type": "hold", "name": "fc",        // the compiled FCSL controller
      "config": {
        "input_map":  { "fc.store.pos_z": "vehicle.z" },      // sim  -> FC
        "output_map": { "fc.store.throttle": "cmd.throttle" } // FC   -> sim
      } }
  ]
}
```

6 Telemetry and visualization

Observability is a first-class, decoupled path (Figure 3). The simulator publishes telemetry as UDP datagrams. A lightweight HTTP telemetry service supervises receiver processes and allocates ports; each receiver ingests the stream and compacts it from row form into columnar NumPy windows on

disk, which are convenient for later analysis with the scientific-Python ecosystem. The visualizer reads those columns and streams them to a browser dashboard over WebSocket. Because storage sits between producer and viewer, runs can be reviewed after the fact, and the simulator is never blocked by a slow or absent consumer.

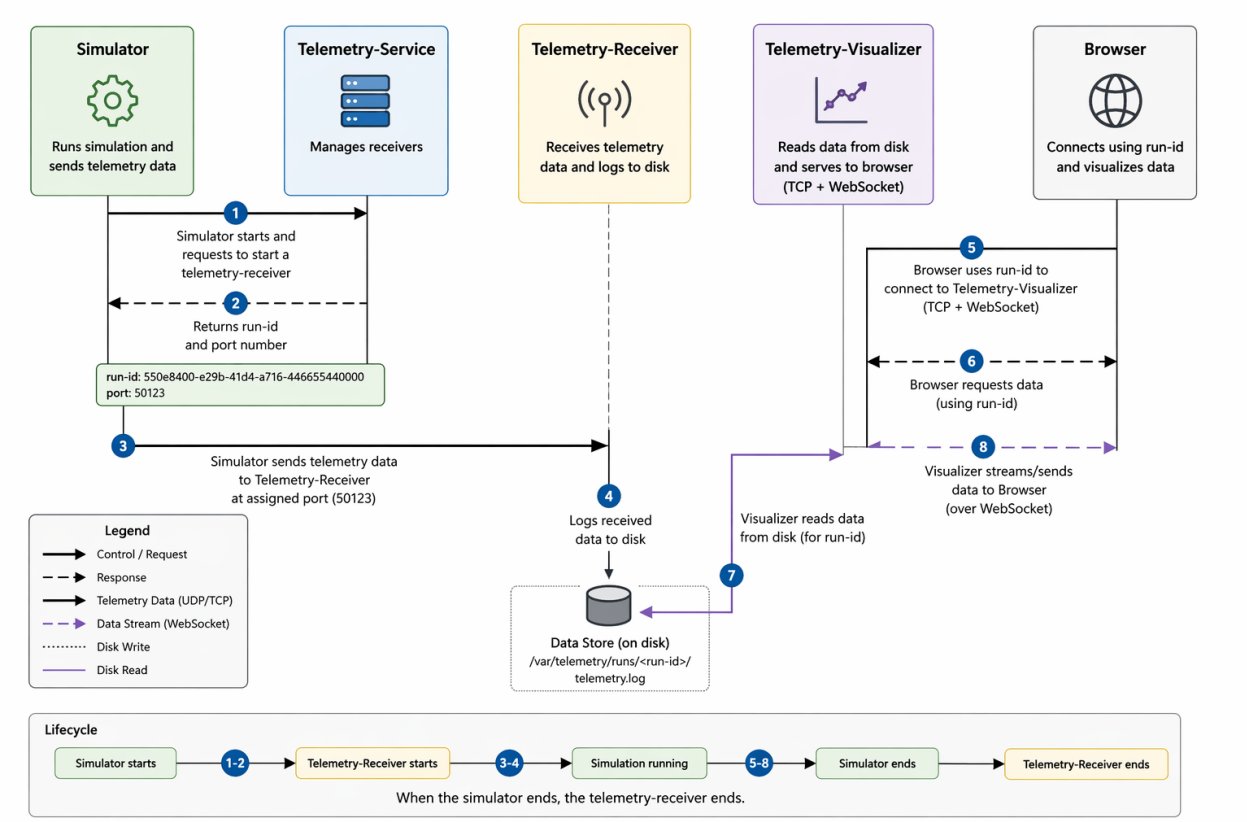

Figure 3: The telemetry path. The simulator asks the telemetry service to start a receiver, then streams UDP telemetry to it; the receiver logs columnar data to disk; the visualizer reads from disk and streams to the browser over WebSocket. The lifecycle is bracketed by the simulator's start and stop.

The dashboard (Figure 4) provides time-series plots, 3D vehicle views driven from the telemetry stream, string and diagnostic panels, and command tables that write back to the running simulation through the variable server. A single page can therefore both observe a run and steer it.

Programmatic control with simclient. Not all control happens in the browser. `simclient` is a Python library for driving and observing a run programmatically. It can send and synthesize telemetry, read live or recorded streams for analysis, and — through *command twins*, Python-side proxies of simulation variables — `get`, `set`, and `eval` state, pause and resume the clock, and fire named events against the variable server, with asynchronous control surfaces for scripting. This is how scenarios are scripted, parameter sweeps are launched, and automated tests drive the simulator and assert on its telemetry: the same control path a human uses from the dashboard, exposed as ordinary Python and backed by the scientific-Python ecosystem for analysis of the columnar logs.

Figure 4: The browser dashboard during a flight of the triangle vehicle — a three-body rigid assembly with three thrusters. A 3D view rendered live from the telemetry stream (left) sits beside synchronized plots of altitude, attitude, throttle, controller state, mass, and propellant (right), with a command panel (top) that writes back to the running simulation.

7 Design decisions

A few deliberate choices shape the stack.

Determinism first. Nanosecond integer time, double-buffered state, and seeded per-model randomness make runs reproducible and replayable. This is what makes save/restore, regression testing, and debugging tractable, and it is a prerequisite for trusting a controller validated in simulation.

Locality and declarative composition. Models communicate only through named DataStore variables and are assembled from a JSON configuration. Behavior is local to a model and scenarios are declarative, which keeps the system easy to reason about, test, and extend — by humans and, as discussed below, by machines.

A compiled control DSL. The flight computer is authored in a restricted, Python-like language and compiled to native code. Engineers write control logic at a high level, but the validated artifact is fast native code with no interpreter in the loop — and it is the same artifact that ships.

Polyglot plugins over a C ABI. Keeping the model interface a narrow C-ABI contract makes the simulator language-agnostic at its boundary while the host itself is implemented in Nim for compile-time metaprogramming, fast startup, and close Python interoperation for the control-DSL pipeline.

One transport for sim and hardware. Using Zenoh both between the simulator and out-of-process flight computers and for hardware-in-the-loop means the wire path is exercised long before hardware, and a bridge that encodes variables in Common Data Representation (CDR) lets robotics middleware see the simulation as ordinary topics.

Single-source, reproducible builds. The build is generated from a single declarative description and backed by a hermetic package manager, so the whole stack — simulator, flight-computer compiler, telemetry tools, and tests — builds reproducibly from one command.

8 Machine-learning and AI features

HiroSim treats data-driven modeling and AI-assisted engineering as first-class, in two distinct senses.

Gray-box learned models. Some vehicle subsystems are hard to model from first principles but easy to log. For these, HiroSim ships gray-box model types that are fitted offline from input/output data and then loaded as ordinary plugins. They share a differentiable "Wiener" core: a bank of learnable second-order infinite-impulse-response (IIR) filters (biquads) supplies the linear dynamics — the memory that produces lag and hysteresis — and a small neural network supplies the static nonlinear map. The filters' poles are parameterized so that they are stable by construction, and the runtime is a fixed-cost, allocation-free forward pass suited to the deterministic scheduler. Two instances ship today: an engine model mapping valve and condition inputs to thrust, mass flows, and shaft speed; and an aerodynamic model mapping flight state and control-surface deflections to aerodynamic coefficients and then to forces and torques. Because the linear block generalizes the rational-function approximations long used in aeroelasticity, the learned parameters remain interpretable, and the models capture unsteady effects that a static lookup table cannot.

Designed for AI authorship. The same properties that make the stack pleasant for human engineers — locality of behavior, declarative composition, strong typing, reproducibility, and documentation kept adjacent to code — also make it tractable for a large language model to extend correctly. Much of the stack, and several complete worked examples (a vehicle, its flight computer, and a test harness), were generated end to end from natural-language prompts. The narrow, well-typed model interface gives an AI agent a small target with fast, deterministic feedback: a generated plugin either compiles, registers, and reproduces a known trajectory, or it does not. AI assistance is thus not a bolt-on but a consequence of the architecture, and it is used under human review.

9 Conclusion

HiroSim is a modular, deterministic stack for closed-loop simulation and control-software development: an event-driven simulator with nanosecond time, multi-rate pluggable models, and multiple integrators; a compiled Python-DSL flight computer with a hardware abstraction layer that carries the same binary from simulation to hardware; a decoupled telemetry and visualization path; and a reproducible, single-source build. Its determinism-first, locality-oriented design serves both ends of its intended use: reliable engineering validation, and extension by data-driven models and AI agents. Detailed component documentation, worked examples, and the companion paper on the learned aerodynamic model accompany this overview in the project repository.