Designing a Simulation Stack for Large-Language-Model Collaboration: The HiroSim Experience

Massimo Di Pierro, *University of California, Santa Cruz, CA, 95064, USA*

Abstract—We report on the design and construction of HiroSim, a modular, deterministic, closed-loop simulation stack for flight-software development, the bulk of which was authored in collaboration with a large language model (LLM). Our interest is less in the productivity anecdote—which by now is unremarkable—than in a more structural question: which properties of a simulation framework make it tractable for an AI agent to extend it correctly? *We argue that the features that make a stack pleasant for a human engineer (locality of behavior, declarative composition, strong typing, reproducibility, and documentation kept adjacent to code) are precisely the features that make it amenable to machine authoring, and that several design choices we originally made for unrelated reasons turned out to be load-bearing for LLM collaboration. We illustrate the argument with a worked case study—a three-body vehicle, its flight computer, and a test harness, all generated end-to-end from natural-language prompts—and we discuss the failure modes we observed, where the abstraction leaked, and what we would recommend to others building tools intended to be extended by both humans and machines.*

Physics simulators for vehicle and flight-software development have a long lineage, from NASA's Trick to Gazebo to the software-in-the-loop (SIL) harnesses shipped with PX4 and ArduPilot. They share a common shape: an event-driven scheduler advances a set of models, a controller is exercised against that simulated environment, and telemetry is recorded for offline analysis. HiroSim [1] sits squarely in this tradition. It comprises a deterministic, nanosecond-resolution Nim simulator with dynamically loaded model plugins; a restricted-Python domain-specific language (DSL), the Flight Computer Specification Language (FCSL), that compiles flight-control logic to native code; a User Datagram Protocol (UDP) telemetry pipeline that compacts samples into columnar storage; and a browser dashboard, all connected over the Zenoh [9] pub/sub transport so that the same flight-computer binary that flies the simulation can fly real hardware.

XXXX-XXX
Digital Object Identifier 10.1109/XXX.0000.0000000

What distinguishes the present report is not the artifact but the process. A substantial fraction of HiroSim was written through a conversational interface to an LLM: the framework primitives, large parts of the simulator, the telemetry receiver, the DSL compiler, and—most relevant here—several complete vehicles and their controllers were produced by issuing natural-language instructions and reviewing the resulting code. We were therefore in the unusual position of designing a tool while simultaneously observing, in real time, which of its design choices helped or hindered the machine that was extending it.

This paper distills those observations. We do not claim that LLM authorship is uniformly beneficial; we encountered places where the model confidently produced plausible, wrong code, and we are explicit about them. Our thesis is narrower and, we think, more durable: *a simulation framework designed around locality, declarative composition, strong static contracts, determinism, and co-located documentation is markedly easier for an AI agent to extend correctly than one that is not, and these are largely the same properties that make it easier for humans.* The novelty

for an LLM collaborator is that it has no persistent memory of the codebase between sessions, cannot run the system in its head, and pays a steep price for any ambiguity it must resolve by guessing—so the framework's structure has to substitute for the situated knowledge a human contributor would accumulate.

Background and related approaches

The contrasts with HiroSim's predecessors motivate the design choices we later argue are AI-friendly.

NASA Trick.

Trick [4] established the template HiroSim follows most closely: an event-driven scheduler, a variable server for runtime introspection and save/restore, and per-model multi-rate scheduling. Trick's jobs are C/C++ functions registered with the scheduler. HiroSim keeps the scheduler shape and the variable-server protocol but replaces the in-tree C++ model with a dynamically loaded plugin exporting a single C-ABI symbol, and replaces the hand-written controller with a compiled DSL.

Gazebo and ROS 2.

These [5], [6] target the robot-platform problem: a rich ecosystem of sensors, transforms, and message types. HiroSim is deliberately *not* a robot-platform replacement; it is a simulation plus flight-computer framework. It interoperates with ROS 2 by publishing Common Data Representation (CDR)-encoded `std_msgs/Float64` and a `/clock` topic through a bus model, so an external bridge sees the simulation as ordinary topics, but it does not attempt to subsume the robotics middleware.

PX4 and ArduPilot.

These [7], [8] pioneered the "same binary in simulation and on the vehicle" discipline that HiroSim adopts. The differences are in the substrate: the HiroSim flight computer is a Python DSL compiled to native code rather than a hand-written C++ autopilot, the transport is Zenoh rather than MAVLink, and the vehicle model is supplied by the user rather than baked in.

The common thread is that all of these systems were designed to be extended by *human* engineers who would spend weeks internalizing their conventions. The question we found ourselves asking is what changes when the extender is an LLM that arrives with broad but shallow knowledge, no memory of yesterday's session, and an inability to execute code mentally.

Architecture in brief

HiroSim is organized as a set of small processes and libraries rather than a monolith. Three traffic flows connect them: a one-way UDP telemetry push from producers to a receiver that compacts samples into per-channel NumPy column windows; a bidirectional JSON-over-TCP command protocol for `get/set/eval/pause/save/restore`; and a Zenoh pub/sub link carrying the sensor-and-actuator control loop between the simulator and the flight computer.

Inside the simulator, five concepts compose everything else. A **DataStore** is a flat, typed key-value space holding every named scalar or fixed-size array. **Models** are the units of behavior, each owning a state object and a small set of lifecycle callbacks—`init`, `wire`, `advance`, and, for integrated bodies, `getDerivatives`. **Services** are named functions one model exposes for siblings to call, decoupling, say, a gravity source from its consumers. **Force providers** invert that pattern for the integrator: a thruster registers itself as a force on a vehicle, and the vehicle's integrator sums all registered forces each step without knowing their identities. The **Scheduler** drives the tick, dispatching due models onto a thread pool grouped by a declared `order`, committing a double-buffered `nextState` into `prevState`, and emitting telemetry.

Two properties of this core matter for what follows. First, the DataStore is double-buffered: writers target `nextState`, readers see `prevState`, and a commit at the end of each scheduler order level publishes one into the other. Every model therefore sees a consistent snapshot of the previous step, and order-level parallelism is safe by construction. Second, after a `freeze()` call at the end of initialization, the variable layout, service table, and integration state vectors are immutable; the steady-state hot path is a sequence of indirect loads and stores with no allocation.

The flight computer is a separate concern. FCSL takes a JSON program describing a state machine, its algorithm instances, their wiring, and per-state schedules, together with algorithm bodies written in a restricted Python subset, and emits a single Nim source file with static types, allocation-free execute paths, compile-time transition arbitration, and optional foreign-function bindings. The same generated module can be linked into the simulator as an in-process plugin, run as a standalone daemon over Zenoh, or—experimentally—driven through a hardware-abstraction layer against either synthesised or real I/O, defining a four-rung software-in-the-loop-to-hardware-in-the-loop (HIL) validation ladder.

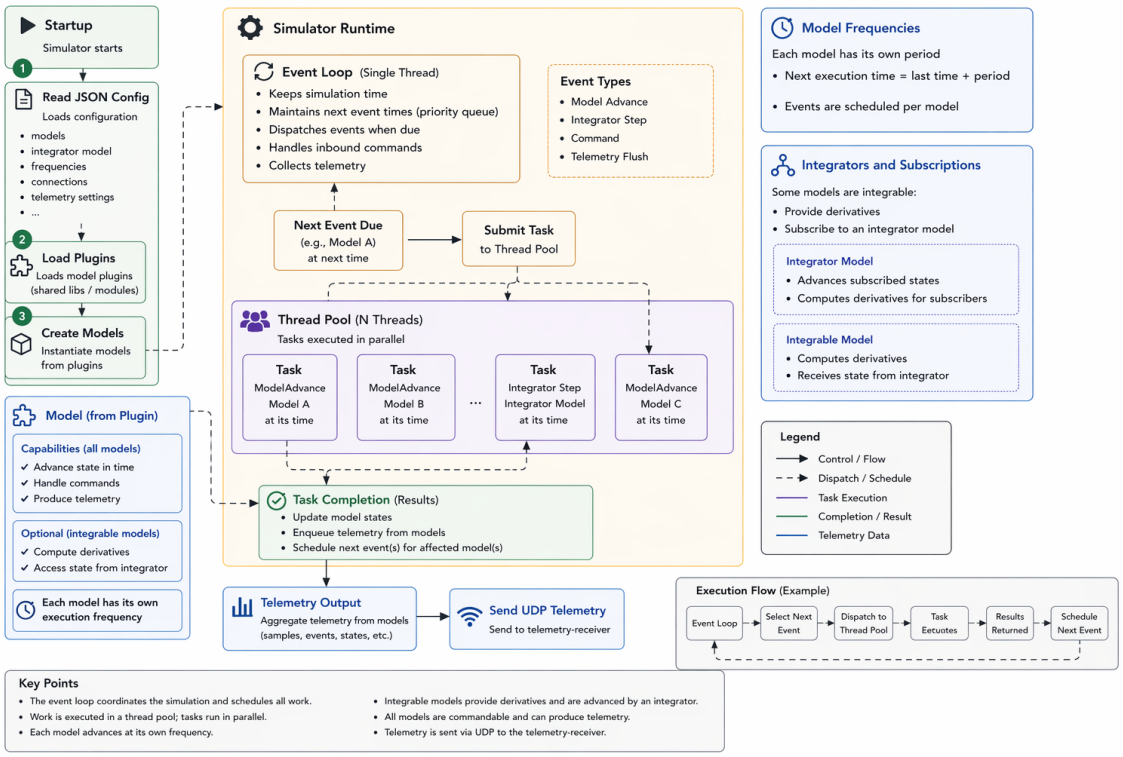

FIGURE 1. The simulator runtime. A single-threaded event loop draws due events from a priority queue and dispatches each model's `advance` onto a thread pool, ordered by dependency. Cross-model coupling happens through subscriptions and registered force providers, so models and vehicles are added via JSON without code changes.

Design properties that make machine extension tractable

Across many sessions of extending HiroSim with an LLM, the same handful of structural properties recurred as the difference between a clean extension and a frustrating one. We present them as design principles, each with the mechanism in HiroSim that embodies it and the reason it helps a machine in particular.

Composition is declarative, not imperative

The most consequential choice is that new physics is assembled by *wiring* rather than by editing source. A thruster is coupled to a vehicle by listing it as a force provider in JSON; a gravity model is consumed by passing its service name as a configuration string. Neither the vehicle's code nor the thruster's code is touched when a new force is added. A controller is bound to a vehicle by an `input_map` and an `output_map` that name DataStore channels.

For an LLM this is decisive. Generating a correct JSON object that references existing, documented primitives is a constrained, well-specified task with a verifiable shape. Generating correct imperative code that mutates a shared object graph—getting the locking, the update order, and the lifecycle right—is an open-ended task with many ways to be subtly wrong. By pushing composition into a declarative layer, HiroSim converts most "add a vehicle" or "add a sensor" requests from program-synthesis problems into configuration problems, which current models handle far more reliably.

Locality and model isolation

Because models communicate only through the DataStore and the service table, they do not hold references to one another. A sensor reads a channel; it does not know which model produces it. This locality means that an agent asked to add a model needs to understand that model's contract and the names of the channels it touches—not the global control flow of the simulator. The relevant context fits in a prompt. The alternative, common in tightly-coupled simulators, is that correctly adding a feature requires reasoning about interactions

scattered across the codebase, which is exactly the kind of non-local reasoning that both exceeds an LLM's working context and is hard to verify after the fact.

Contracts are checked at compile time

We built FCSL to be narrow on purpose: not a general-purpose language but one with "semantics simple enough to reason about, schedule, inspect, and eventually harden." That narrowness is a gift to a machine author. The type system is small (a handful of scalar types plus fixed-size arrays and a few enums); array dimensions are static, so a dimension mismatch in a linear-algebra builtin is a Nim compile error rather than a runtime surprise; transition requests compile to an enum, so a typo in a state name fails the build; and the execute path is statically guaranteed allocation-free by a test that scans generated code for forbidden patterns. An LLM is good at producing code that is locally plausible; a tight static contract is what catches the cases where plausible is not correct, and it does so at the build step rather than three seconds into a flight.

The restriction list is itself a feature: FCSL forbids `while` loops, recursion, dynamic allocation, comprehensions, and arbitrary string manipulation inside algorithms. Each forbidden construct is one fewer way for a generated controller to be non-deterministic or unbounded, and the compiler's rejection message tells the agent precisely what to change.

Determinism makes verification cheap

HiroSim runs are byte-reproducible. Two ingredients combine to make them so: an integer nanosecond `sim_time` shared by the scheduler, integrators, and telemetry timestamps, so there is no floating-point time drift; and a per-model seeded pseudo-random number generator (PRNG), so two runs of the same configuration produce identical noise. Reproducibility is valuable for any engineering workflow, but it is *especially* valuable when a non-deterministic collaborator proposes a change, because it lets the change be verified against a fixed baseline. An agent can assert that a controller reaches a target altitude on a specific deterministic trajectory, and that assertion either holds every time or fails every time. Flaky verification is corrosive to an automated edit-test-review loop; determinism removes the flakiness at the source.

Comprehensive tests prevent regressions

A test suite plays a different role for a machine contributor than for a human one. A human carries a memory of which invariants matter; an agent starting fresh does not, and is therefore prone to "fixing" one behavior while silently breaking another it never knew existed. HiroSim leans on a large regression suite—more than 700 unit tests spanning the simulator, the telemetry pipeline, the command protocol, and the DSL compiler—as the externalized memory of those invariants. Two things make it effective inside the loop. First, the suite is fast and deterministic (by the same integer-clock and seeded-PRNG construction described above), so the agent can run it after every change and read an unambiguous pass or fail rather than chasing flakes. Second, the project's convention is that tests ship with the change that motivates them: when the agent adds a model, a protocol operation, or a DSL construct, it also writes the tests that pin the new behavior, and those tests immediately join the net that guards every subsequent edit. The effect is a ratchet—each generated feature widens the suite, and the widened suite constrains the next generation—so the agent's own output progressively shrinks the space in which it can later regress the system. One caveat is intrinsic and worth stating plainly: a generated test encodes the behavior the agent *believed* correct, so a test written against a misunderstanding locks that misunderstanding in. The suite catches drift from an established baseline far more reliably than it catches a first implementation that was wrong to begin with, which is why the generated tests complement rather than replace human review of new behavior.

The system is observable on a fast loop

A machine collaborator cannot watch a dashboard. HiroSim therefore supports running the simulator directly and printing a per-second status line from a configurable set of `status_vars`, so an agent can confirm what a vehicle actually does without standing up the interactive visualizer. This closed feedback loop—generate a config, run headless, read the status line, adjust—is the mechanism by which the agent corrects its own mistakes.

The same priority on a short loop informed the choice of implementation language. The simulator and the code FCSL emits are Nim [2], [3], which compiles to C and runs at native speed but—unlike Rust, the other obvious candidate for native, memory-safe systems code—compiles very fast. That property matters more for a machine collaborator than for a human one. Because the flight-computer pipeline transpiles and recompiles a native plugin on every change to an algorithm or its wiring (the stale-artifact hazard discussed below), the agent's edit-build-run cycle has a compile step inside it, and that cost is paid on every iteration of

the correction loop. A language with a slow compiler would tax exactly the loop the agent depends on to converge; Nim's quick rebuilds keep each iteration cheap, which in aggregate is the difference between an agent that iterates freely toward a working controller and one throttled by its own toolchain.

The framework's command protocol reinforces this: the same JSON wire format is spoken by the simulator's variable server and by the flight computer's command server, so a single client can introspect either side without branching on which it is talking to. Uniform, scriptable introspection turns "does this work?" into a query rather than a judgment call.

Documentation and code are together

Perhaps the least glamorous and most important property: HiroSim treats documentation as a hard dependency of every change. The project's internal guidance states that "every behavior change ships with its docs in the same edit" and that leaving documentation stale "produces a worse repo than not making the change at all." A table maps each kind of change to the documents it must update; every function must carry a docstring; examples are expected to survive a verbatim grep. The effect on an LLM collaborator is direct and compounding. The model's primary source of situated knowledge about the codebase is its documentation, because it has no memory of prior sessions. When the docs are accurate, co-located, and exhaustive—defining every term in a glossary, specifying the wire protocol byte-for-byte, listing every DSL construct with a worked example—the agent can ground its generation in fact rather than in its priors about how such a system "usually" works. Documentation maintained as a first-class artifact is, in effect, the long-term memory the model lacks.

Case study: a three-body vehicle, end to end

To make the argument concrete we recount a single development session, recorded in the project's development log, in which a complete vehicle, its flight computer, and a test harness were built entirely from natural-language prompts. The prompts were lightly edited for spelling but not for intent.

The first instruction was mechanical:

> Three rigid bodies connected by three rods in the shape of a horizontal triangle, a downward thruster on each body, each rod 1 m, body mass equal to 100 kg minus consumed propellant.

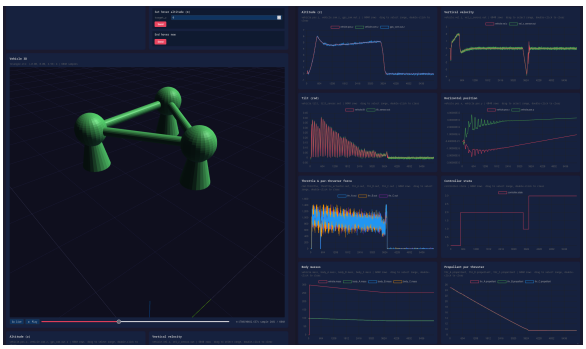

FIGURE 2. The triangular vehicle of the case study, rendered in the telemetry visualizer from the simulator's channels. Three rigid bodies sit at the vertices of an equilateral triangle, joined by three fixed-length rods, each carrying a downward thruster. The vehicle, its four-phase flight computer, and its test harness were produced end-to-end from natural-language prompts. A video of a flight is available at https://www.youtube.com/watch?v=ndIrsdUYUNY.

Because the simulator already shipped every primitive the sketch needed—rigid bodies, fixed-length distance constraints, thrusters, and a propellant-coupled mass model whose interpolation matched the prompt's mass = 100 − consumed rule one-for-one—the work was wiring, not invention. The only genuine computation was geometric: placing the bodies at the circumradius $1/\sqrt{3} \approx 0.577\,\text{m}$ of an equilateral triangle centered at the origin. The remaining configuration—gravity, a fourth-order Runge–Kutta (RK4) integrator listing the bodies and thrusters as dependencies, the constraint solver scheduled immediately after, a variable server, and a set of scripted pilots to register the thrusters' input channels—was scaffolding the operator did not have to specify because the framework's conventions supplied it.

The second instruction added sensors:

> A noisy sensor measuring the vehicle's position and center of mass, and a per-body mass sensor.

This step required more than wiring. The simulator's sensor primitives read existing scalars; they do not compute aggregates. The vehicle, being three bodies, has no single position channel. The resolution was to introduce a vehicle-state aggregator that publishes the arithmetic-mean position and the mass-weighted

center of mass as derived channels, and then to point ordinary GPS-style sensors at those derived bases. This is exactly the kind of two-step composition the declarative layer supports: expose a new derived quantity as a channel, then attach standard primitives to it.

The third step was the flight computer:

> Lift by 5 m, hover for 30 s using sensor position and attitude, then land.

This was the most state-machine-heavy task and the one where the DSL's structure carried the most weight. The controller became a four-state machine—ascent, hover, descent, landed—with an abort edge to "landed" from each non-terminal state guarded on a tilt-limit. Each phase was a small algorithm with explicit inputs, outputs, parameters, and an `execute` body; hover ran a proportional-derivative loop and used an integer tick counter to enforce the dwell exactly; descent switched to an analytic stopping-distance brake near the ground; landed was terminal by construction, with no outgoing edge. On the simulator side the scripted pilots were removed, a throttle actuator was inserted, and the thrusters were rewired to read it—again, a wiring change, not a code change.

The final steps built the test harness: a visualizer dashboard mirroring an existing one, a command button that ends the hover early by writing zero into the hover phase's dwell parameter, and a one-command local testbed that brings up the telemetry service, the visualizer, and the simulation together, starting paused so an operator can inspect the initial state before resuming.

Two things about this session stand out. First, almost every step reduced to either generating a JSON object against documented primitives or writing a small, contract-bounded algorithm—the two tasks the framework is shaped to make safe. Second, the one step that required genuine invention (the vehicle-state aggregator) was invention *within* the model contract: it produced a new channel that downstream primitives consumed unchanged. The framework did not eliminate the hard part; it confined it to a single, well-bounded place.

Where the abstraction leaked

In some cases machine authorship was difficult or dangerous. The three leaks we hit most often share a signature: in each, the system's behavior was not a pure function of the source the agent was editing.

The sharpest is the build-time gap in the flight-computer pipeline. FCSL controllers are not interpreted; the compiler bakes each machine and its algorithms into a shared library, so editing an algorithm, a connection, or even an instance parameter has *no effect* until the plugin is rebuilt, and running a newer configuration against a stale library typically *segfaults at startup* rather than erroring cleanly. This stale-artifact trap is exactly the state an LLM—which reasons about source, not the binary on disk—is prone to fall into. The second is numerical silence: the DSL catches exceptions but not floating-point pathologies, so $\sqrt{-1}$ and $1/0$ yield NaN and $\pm\infty$ that propagate without raising. A generated controller that omits a bounds check before an `asin` compiles cleanly yet can poison an outer loop at runtime. The third is the hot-path allocation discipline: a per-tick allocation introduces tail-latency jitter that a profiler would catch but a unit test would not, so a change can be correct and still wrong. In all three the framework falls back on prose and review rather than a check that fails the build.

The properties that made extension safe were backed by a mechanism that fails loudly at build or test time; the ones backed only by convention were where machine authorship most often went wrong. The lesson generalizes: to make a tool safe for an AI collaborator, convert as many correctness obligations as possible from conventions into checks.

Discussion

The experience of building HiroSim suggests that "AI-friendly" should not be an afterthought but, to a first approximation, a restatement of long-standing good design under a harsher constraint. Every property we found valuable—locality, declarative composition, strong static typing, determinism, scriptable observability, and co-located documentation—is something a thoughtful engineer would already value. What changes with an LLM collaborator is the *penalty for violating them*. A human contributor compensates for a non-local design by building a mental model over weeks; an LLM cannot, and so non-locality that a human would merely find annoying becomes, for the machine, a source of confident error. A human remembers that a particular edit needs a rebuild; an agent starting fresh each session does not, unless the requirement is written down where it will be read. The constraint is harsher, but it points in the same direction.

The highest-leverage investments for AI extensibility are not model-specific tricks; they are the boring disciplines—narrow, typed interfaces; declarative configuration over imperative wiring; reproducible execution; a fast headless feedback loop; and documentation maintained with the same rigor as code. HiroSim

adopted most of these for reasons that predate its AI-assisted development: the integer clock was chosen for determinism in regression testing, the plugin boundary for polyglot model authoring, the DSL's narrowness for eventual hardening toward flight. That they also made the system tractable for an LLM to extend was, in several cases, a fortunate consequence rather than an intention—which is itself evidence that the underlying principles are sound.

The case study describes a favorable scenario: a domain with clean primitives, a declarative composition layer, and an operator able to review every step. The model did not autonomously discover that it needed a vehicle-state aggregator; it was guided there. The build-time and numerical leaks above show that the agent's output required human review precisely where the framework's guarantees thinned out. The right reading is not that the framework made the LLM reliable, but that the framework's structure determined *where* the human's attention was needed—concentrating it on the few genuinely ambiguous decisions and removing it from the many mechanical ones.

Limitations and future work

This is a single-project experience report, not a controlled study; we cannot quantify how much the design choices helped relative to a baseline, and our observations are inevitably colored by having made those choices ourselves. A natural next step is to harden the leaky seams into checks: a staleness guard that refuses to run a configuration against an out-of-date plugin rather than segfaulting, and optional NaN/Inf poison-tracking on the flight-computer execute path that an agent could enable during development. More broadly, we would like to measure whether the same declarative-composition discipline that helped with vehicle and controller authoring extends to the harder task of *debugging*, where the agent must localize a fault rather than synthesise a well-shaped artifact. We expect the same properties—locality and reproducibility above all—to matter even more there, but we have not yet tested it systematically.

Conclusion

We set out to build a deterministic, modular simulation stack for flight-software development, and we built much of it in collaboration with a large language model. The lasting lesson is not about the productivity of that collaboration but about its preconditions. An AI agent extends a framework well when the framework lets it work locally, compose declaratively against documented primitives, lean on static contracts that fail loudly, verify against deterministic baselines, observe the running system through a script, and ground its generation in documentation that is kept honest. Where HiroSim offered these, adding a vehicle, a sensor, or a controller became a constrained and reviewable task. Where it fell back on convention—stale build artifacts, silent floating-point, allocation discipline—machine authorship was at its most fragile. The design prescription that follows is unglamorous and, we suspect, general: build the tool the way you would for a careful human who has no memory and cannot run the code in their head, and turn every convention you can into a check.

Acknowledgments

This paper, like HiroSim itself, was written in collaboration with an AI assistant (Anthropic's Claude, Opus 4 family). The author is solely responsible for all content.

REFERENCES

1. M. Di Pierro, "HiroSim simulation stack." Accessed: May 2026. [Online]. Available: https://hirosim.com
2. "Nim programming language." Accessed: May 2026. [Online]. Available: https://nim-lang.org
3. A. Rumpf, *Mastering Nim: A Complete Guide to the Programming Language*. Self-published, 2022.
4. NASA Johnson Space Center, "Trick simulation environment." Accessed: May 2026. [Online]. Available: https://github.com/nasa/trick
5. Open Source Robotics Foundation, "Gazebo simulator." Accessed: May 2026. [Online]. Available: https://gazebosim.org
6. Open Source Robotics Foundation, "Robot Operating System 2 (ROS 2)." Accessed: May 2026. [Online]. Available: https://www.ros.org
7. "PX4 autopilot." Accessed: May 2026. [Online]. Available: https://px4.io
8. "ArduPilot." Accessed: May 2026. [Online]. Available: https://ardupilot.org
9. Eclipse Foundation, "Eclipse Zenoh." Accessed: May 2026. [Online]. Available: https://zenoh.io

Massimo Di Pierro is a specialist in numerical simulation and scientific computing. For eight years he led the simulation team at SpaceX, and he has taught as a Professor of Computer Science at DePaul University and at the University of California, Santa Cruz. His current interests center on high-fidelity physics simulation and AI-assisted software engineering.