Chaperonin

A Modular, Type-Checked Workflow Engine for Computational Biology

Sihoo Lee (25-083) Hangyeol Lim (25-098)

Korea Science Academy of KAIST

2026-1 Information Science Project Showcase

June 2026

Abstract

Modern protein engineering depends on a sprawl of heterogeneous command-line tools—RFdiffusion, Rosetta, AlphaFold, RoseTTAFold, PyMOL—that disagree on file formats, runtime environments, and parameter conventions. Composing them today means writing throwaway shell glue, which kills reproducibility and gates bench scientists out of the iteration loop. We present **Chaperonin**, a modular workflow engine that lets users assemble these tools either visually, on a node canvas, or textually, as a small domain-specific language (DSL), with full bidirectional translation between the two views. Pipelines are typed dataflow programs: a wire is admissible only when the producer's output type satisfies the consumer's input type, checked at edit time before any GPU-hour is spent. A standard-library-only Python orchestrator topologically sorts the graph and dispatches each node as a sibling Docker container against the host daemon, streaming logs, progress, and rendered 3D structures live to the browser over a hand-rolled WebSocket. New tools are added by dropping a single self-describing Python file into a directory—no registry edits, no scheduler patches. We describe the system's architecture, type system, dual representation, and execution model, and demonstrate three end-to-end pipelines (energy relaxation, iterative worst-of-N selection, and *de novo* binder design) running on a laptop.

1 Introduction

Proteins underlie essentially every biological process, and the design of novel proteins increasingly relies on computational tools such as RFdiffusion, Rosetta, and AlphaFold. Yet three obstacles keep these tools out of the hands of the biologists who would benefit most from them.

(i) **The CS/CLI barrier.** Each tool assumes programming fluency and a comfort with the terminal that most experimental biologists do not have.

(ii) **Format fragmentation.** Every tool ships its own schema; moving data between two tools is a manual, error-prone conversion step.

(iii) **No definite protocol.** There is no standard, shareable way to express "the sequence of steps" that constitutes an analysis.

The purpose of this project is to build a single platform on which a biologist can compose these tools—visually or textually—without programming, run them, and inspect the results, so that biological expertise rather than computer-science fluency sets the limit on what can be attempted. Chaperonin answers each obstacle directly: a drag-and-wire graphical interface removes the need for any terminal knowledge, a hierarchical type system makes edges self-converting and refuses incompatible connections at edit time, and the canvas itself *is* the protocol—a single artifact, readable by expert and novice alike, that both documents and executes the workflow. The name is deliberate: chaperonins (GroEL, HSP60) are proteins whose job is to help *other* proteins reach their correct folded state. A system whose job is to help other tools reach their functional output is the project's mission statement in metaphor.

2 Main Body

2.1 Overall Architecture

Chaperonin is organized as four layers communicating over a single socket (Figure 1).

At the top, a React single-page application renders the node canvas (via ReactFlow), the DSL editor, a streaming log panel, and an inline 3D structure viewer (3Dmol.js). The frontend keeps no business logic beyond a colour lookup for edge types; everything authoritative lives in the orchestrator below it.

The orchestrator is the user-facing process: a standard-library-only Python 3.12 asyncio server with no web framework and no third-party WebSocket library. It exposes a tiny HTTP surface (`GET /api/modules` for the palette, `POST /api/upload` for inputs, `GET /api/outputs/...` for produced files) and a single `/ws` WebSocket endpoint that accepts `run` and `cancel` messages and emits the live event stream; the handshake and RFC 6455 frame coding are implemented by hand in roughly

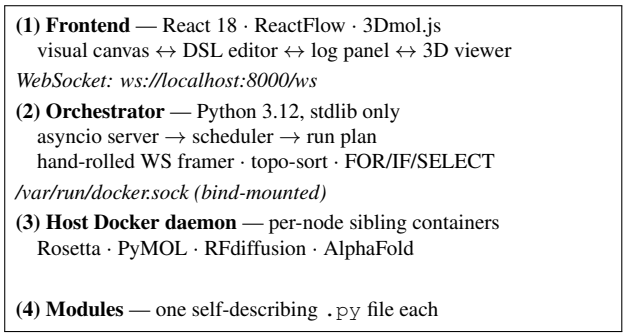

Figure 1: The four-layer architecture. The Chaperonin container talks to the host Docker daemon through the bind-mounted socket; each module node spawns a *sibling* container.

sixty lines.

Beneath that sits the host Docker daemon. Because the orchestrator container mounts `/var/run/docker.sock`, each module dispatches a *sibling* container rather than a nested one, and bind mounts use identical host and container paths—so any path produced by Rosetta resolves to the same string when PyMOL reads it in the next node, with no path rewriting between steps. Each tool itself is a single Python file under `backend/modules/`, so adding or modifying a tool touches exactly one file and never the orchestrator.

The visual canvas and the DSL are not separate features but two projections of the same underlying pipeline AST. Editing either side and pressing *Apply* re-derives and updates the other; the round trip is lossless. The DSL is a small, Python-flavoured language:

```
pdb = input(Structure.PDB, label="scaffold")
relaxed = ROSETTA_RELAX(structure=pdb, nstruct=1)
viz = VISUALIZER(value=relaxed.relaxed)
best = select(from=relaxed.score, mode="min")
```

Here `input(TYPE, label=...)` declares a source, a binding `VAR = MODULE_ID(input=source.handle, param=value)` is a module call referencing producers by handle, and `output(var.handle, name=...)` declares a sink. Offering both a Scratch-style canvas and a textual form means the same protocol is approachable to a novice yet diff-able, copy-pasteable, and version-controllable for an expert.

2.2 Software and Hardware Technologies

The stack (Table 1) is deliberately thin. The frontend is a React single-page app built with Vite, using ReactFlow for the node canvas and 3Dmol.js for in-browser 3D rendering. The orchestrator is written in *standard-library-only* Python 3.12—no web framework and no third-party WebSocket or HTTP library—so it has no dependency to install or pin beyond the interpreter itself. Tools are dispatched through the host's `docker` CLI, and the browser and orchestrator communicate over a single JSON-over-WebSocket channel. The whole system is packaged as one Dockerfile.

Concern	Technology
Canvas / UI	React 18, ReactFlow
3D viewer	3Dmol.js
Build	Vite
Orchestrator	Python 3.12 (stdlib only)
Transport	WebSocket / JSON
Tool dispatch	Docker CLI (sibling containers)
Packaging	single Dockerfile

Table 1: Software stack.

On the hardware side, the only hard requirement is a Docker daemon (Docker Desktop or any Linux daemon) and roughly 30 GB of free disk for the module images. The CPU-only tools—Rosetta Relax, PyMOL, and the host-side converters and visualizer—run on a commodity laptop, including Apple Silicon, where the multi-arch Rosetta image runs natively and amd64-only images run under emulation. The deep-learning modules (RFdiffusion, AlphaFold, RoseTTAFold) are GPU-bound: they are usable at production speed only on a Linux host with an NVIDIA GPU, and their palette entries stay greyed out unless `CHAPERONIN_GPU_AVAILABLE=true` is set.

2.3 Implementation Methods and Algorithms

Several design commitments shape the implementation. There is a single source of truth—each module self-describes through its type annotations, so decorating a class *is* its registration and no separate registry file exists to fall out of sync. Data flows as typed references (*handles*) rather than files, and the engine materializes a concrete path only when a module asks for one. Resource needs are declared honestly per module and respected by the scheduler. Reproducibility is treated as substrate: every output is conceived as content-addressed by `hash(module_version, inputs, params)` under a tiered retention policy. And the core is async-friendly, never blocking the event loop on a subprocess. The remainder of this section describes the algorithms that realize these commitments.

2.3.1 The type system and edit-time checking

Pipelines are typed dataflow programs. Types are hierarchical and dotted: `Structure.PDB` is a subtype of `Structure`. The type module is the single source of truth; the frontend holds only a colour table.

A wire from a producer output of type o into a consumer input of declared type i is admissible iff o *satisfies* i (covariant compatibility). The decision procedure (`is_compatible`) accepts the pair when the wildcard $\star$ is involved (used only by control primitives), when o and i

Namespace	Subtypes
`Structure`	`.PDB`, `.mmCIF`
`Sequence`	`.FASTA`, `.FASTQ`
`Visual`	`.PNG`, `.Web3D` (terminal)
`Text`	`.RawString`, `.Integer`, `.Float`, `.Score`, `.Bool`
`List.`X	elementwise lift of any scalar type

Table 2: The type namespaces.

are equal, when i is a union `A | B` and o satisfies one alternative, when both are `List.`-lifted and their element types are compatible, or when o is a proper sub-namespace of i (i.e. o begins with i`.`), so that `Structure.PDB` satisfies a `Structure` input. Because this check runs at *edit time*, an incompatible wire is refused as the user draws it—the pipeline cannot reach the scheduler ill-typed, and no GPU time is wasted discovering a mismatch at runtime.

2.3.2 Scheduling and container dispatch

When the user hits *Run*, the frontend sends the pipeline payload over the WebSocket and the scheduler (`scheduler.py`) turns it into a run plan. The graph's compute nodes and edges are first topologically sorted (`topo_order`) into a linear execution order that respects every data dependency; the scheduler is deliberately synchronous and runs inside a worker thread, which makes it trivial to unit-test while keeping the asyncio server responsive. Each node then executes through an `ExecutionContext`, either on the host (for pure converters such as `PDB_TO_FASTA`) or in a Docker container whose command is built by a *pure* function (`build_docker_command`)—so the exact `docker run` argv the system will issue is verifiable without a running daemon. Inputs are mounted read-only, the per-run scratch directory read-write, and the mount-at-the-same-path convention guarantees that an argv built from `ctx.path(handle)` resolves identically inside the container.

While a node runs, it emits log lines, progress fractions, scalar metrics, and outputs through the context object; these are bridged from the worker thread back onto the event loop and framed onto the WebSocket, so logs, progress bars, and rendered structures appear in the canvas as they happen, and pressing *Stop* raises a cancellation that kills in-flight containers immediately. Each module also declares a `retention` tier and a version, from which outputs are intended to be content-addressed by `hash(module_version, inputs, params)` for instant resume; in the current build this metadata is declared and surfaced through introspection, while the cache short-circuit and eviction policy remain the next engine milestone rather than a shipped feature.

2.3.3 Control flow

Beyond straight-line dataflow, Chaperonin supports loops and conditional selection through a set of *control primitives*. Crucially, these are not `@module`-decorated tools that dispatch containers; they are operators the orchestrator applies inline, which keeps the module contract clean. The primitives include:

- `START_FOR` / `END_FOR` — a counted loop; the nodes reachable between them form the body, executed once per iteration;
- `SAVE` / `GET` — named variables whose scope is derived from graph topology (a `SAVE` inside a loop accumulates a list that is promoted to the enclosing scope when the loop closes);
- `IF` — gates a subgraph on a boolean;
- `COMPARE` / `SELECT` — utilities that pick an element of a list by an associated key (e.g. minimum score).

This is what makes a worst-of-N search expressible as a single pipeline:

```
scaffold = input(Structure.PDB, label="scaffold")
n = input(Text.Integer, label="n")

f = start_for(count=n, loop_label="trials")
r = ROSETTA_RELAX(structure=scaffold, nstruct=1)
sd = save(value=r.relaxed, name="designs")
ss = save(value=r.score, name="scores")
ef = end_for(paired_start=f.gate, body_out=sd.
    value)

designs = get(name="designs")
scores = get(name="scores")
best = select(from=designs.value,
            by=scores.value, mode="min")
viz = VISUALIZER(value=best.value)
```

With `n = 3`, three Rosetta runs execute in sequence, each emitting a `total_score`; `SELECT` returns the lowest-energy structure and the Visualizer displays it. Any tool can be wrapped this way.

2.3.4 Adding a module

The strongest practical claim of the system is that adding a tool is "create file, decorate, restart." A module is a plain class whose annotations self-describe its inputs, params, and outputs; the `@module` decorator introspects them into a `ModuleSpec` and registers it.

```
from chaperonin import (module, Input,
                  Param, Output)
from chaperonin.types import Structure, Text

@module(name="RFDIFFUSION", category="design",
  container="ghcr.io/.../rfdiffusion:1.5.0",
  resources={"gpu": 1, "memory_gb": 24})
class RFDiffusion:
   pdb_file: Input[Structure.PDB]
   hotspot: Param[Text.RawString]
   designed_pdb: Output[Structure.PDB]

   def execute(self, ctx):
      ctx.run(["rfdiffusion", ...])
      ctx.publish("designed_pdb",
```

```
                ctx.workdir / "designed.pdb")
```

No registry edits and no scheduler patches are required: the palette, the DSL parser, and the canvas all pick up the new module automatically on the next restart, because they are all generated from the same introspected registry.

2.3.5 Codebase and testing

The orchestrator is roughly two thousand lines of dependency-free Python 3.12, split into a type system, a decorator/registry, an introspection layer, the WebSocket protocol, the docker-command builder, an execution context, the scheduler, the scope analyzer, and the control-node table. The frontend is a React/Vite application; a separate `demo/` build runs the same UI against a `simulation.js` stub so the project can be shown without a Docker daemon. A `CHAPERONIN_SIMULATE` flag does the same on the backend, faking module execution with placeholder outputs so the entire pipeline—type checking, scheduling, control flow, streaming—is testable before the heavy container images exist.

The backend ships eleven `unittest` suites (run with `python3 -m unittest discover`) covering the type rules, the decorator and discovery, the docker-command builder, the WebSocket framing, the scheduler, the control-flow primitives, the execution context, simulation mode, and the real module specs. Because the docker-command builder and the type checker are pure functions, the bulk of the engine's behaviour is verified without any external service.

3 Execution Results

We exercised the system end-to-end on a laptop with three pipelines of increasing biological ambition.

The simplest is energy relaxation. The three-node pipeline `Input` → `ROSETTA_RELAX` → `VISUALIZER` takes a PDB structure, energy-minimizes it, and renders the relaxed result inline in roughly 8–10 seconds on a small protein. Relaxation is a crucial preprocessing step for downstream design tools such as RoseTTAFold, yet here it is performed without ever touching a terminal or an external viewer.

The second pipeline adds iteration and choice. The worst-of-N graph of Section 2.3.3 runs Rosetta three times and uses `SELECT` to return the lowest-energy structure automatically—in other words, to keep the most stable of several candidate conformations. The same `START_FOR`/`END_FOR`/`SELECT` wrapper generalizes to any module, letting a researcher systematically sample a range of outputs and keep the best inside a single graph.

The third is a complete *de novo* design pipeline analogous to a real drug-discovery workflow: a target protein structure is passed through `RFDIFFUSION` to generate a candidate binding protein, the candidate is folded by `ALPHAFOLD`, and the prediction is scored by its iPAE—a measure of how confidently the interface geometry is positioned. The graph mirrors the process of developing a binder against a disease target directly, demonstrating that the dispatch path composes the GPU tools, though production-scale RFdiffusion and AlphaFold require a CUDA host rather than Apple-Silicon emulation.

4 Conclusion

Proteins carry out and sustain nearly every process in a living organism, and the ability to design *de novo* proteins is reshaping how we study disease and develop new drugs: a well-designed binding protein can neutralize a pathogen, inhibit a misbehaving enzyme, or serve as the starting point for a therapeutic. The computational tools that make such design possible—RFdiffusion, AlphaFold, Rosetta, and their kin—are extraordinarily powerful, but they were built by and for computer scientists, and the biologists who understand the underlying biology best are too often locked out by the command line, by incompatible file formats, and by the absence of any shared way to describe a design protocol. The cost is not merely inconvenience; it is good biological questions left unasked because the tooling stands in the way.

Chaperonin is our attempt to remove that barrier. By presenting these tools as blocks on a canvas that a researcher can wire together with a mouse, by letting the type system absorb the format conversions that used to be manual, and by adding loops and conditional selection so that whole search-and-select experiments—directly analogous to real drug-development workflows—fit inside a single shareable graph, the system lets a biologist's expertise, rather than their programming fluency, set the limit on what they can attempt. A beginner can follow an established protocol without writing a line of code, while an expert can combine the same blocks in new ways and extend the palette with one Python file. Our hope is that, like the chaperonin protein it is named for, the system quietly does the work of helping other tools—and the people who wield them—reach their functional output.

5 Others

5.1 Repository

The full source, poster, and project proposal are available at `https://github.com/sihooleebd/chaperonin`, released under the MIT license. The backend test suite runs with `python3 -m unittest discover`; a no-Docker demo build is under `demo/`.

5.2 Current Status and Future Work

The system is an honest `v0.x`. The canvas, DSL, bidirectional translation, control flow, GPU gating, and the

Module	Category	Container	Notes
ROSETTA_RELAX	refinement	rosettacommons/rosetta	CPU; multi-arch (native on Apple Silicon). Emits relaxed PDB + Rosetta total_score.
PYMOL	visualization	pegi3s/pymol	amd64 only; under emulation on arm64. Renders PNG + viewable PDB scene.
RFDIFFUSION	design	rosettacommons/rfdiffusion	GPU; CPU-emulation on Apple Silicon is demonstration-only (30 min–3 hr/design).
ALPHAFOLD, ROSETTAFOLD	prediction	(GPU image)	Greyed out unless the GPU-available flag is set.
PDB_TO_FASTA	converter	none (host)	Extracts sequence from PDB. Always available.
VISUALIZER	visualization	none (host)	Renders PDB / PNG / WRL inline (3Dmol.js). Terminal sink.

Table 3: Modules shipped in the current build.

Rosetta + PyMOL + Visualizer triple all run end-to-end. The content-addressed cache and tiered-retention eviction are designed and their metadata is plumbed through introspection, but the live short-circuit is not yet implemented—it is the principal engine milestone remaining. RFdiffusion runs under CPU emulation on Apple Silicon slowly enough to be a proof of dispatch rather than a usable workload, and AlphaFold/RoseTTAFold require a GPU host to leave the palette's greyed state. Future work centres on the cache, opt-in auto-converters between adjacent types, frozen pinned-version pipelines for publication, and a broader module library.

5.3 Reflections

The decision that paid off most was making modules self-describing. Once the `@module` decorator and the introspection layer were in place, the palette, the DSL parser, and the type checker all derived from one source, and adding a tool stopped touching the engine—a small up-front investment that removed a whole category of synchronization bugs. Committing to a standard-library-only backend was initially a constraint we imposed for portability, but it turned out to sharpen the design: hand-writing the WebSocket framing and the `docker run` builder forced us to understand exactly what crossed each boundary, and because those pieces are pure functions they became the easiest part of the system to test. The hardest lesson was about hardware honesty—our first pipelines assumed a GPU was always present, and discovering how slowly the deep-learning tools run under Apple-Silicon emulation pushed us to build resource declaration and palette gating in early rather than bolt them on. More broadly, working at the boundary between biology and software taught us that the real engineering problem was not running any single tool but making the *composition* of tools legible to someone who is an expert in proteins and a novice at the terminal.

Acknowledgments

We thank Prof. An Jeong-Hun (Dept. of Chemistry & Biology) for his assessment of the system's practical value: *"Being able to connect different computational programs and run them on a single platform would be enormously convenient. If Chaperonin really lets a researcher wire the tools together with a mouse and execute them, it will be an exceptionally practical bioinformatics tool."* This work was carried out for the 2026-1 Information Science Project Showcase at the Korea Science Academy of KAIST.