Development of a Digital AI Twin for Urban Infrastructure

A seminar paper

Empirical work:
Implementation of a generative AI agent for the description and optimization of infrastructure networks

Author: Erik Wiedenhaupt

Supervisor: Prof. Dr. Dr. Tanja Kneiske

Date: 21.10.2025

CONTENT

ABSTRACT

Urban pipe networks—natural gas, hydrogen, heat, and water—react in subtle ways to small changes: widen one segment or nudge inlet pressure, and pressures and velocities shift across the map. This thesis presents a conversational "Digital AI Twin" that lets a user describe goals in any language and then watches the system carry out careful, traceable steps: validate code, simulate safely, read KPIs, detect issues, and apply modest edits. The twin is built on PandaPipes and can switch fluids (natural gas, hydrogen, heat, water) without changing the actual orchestration. The core design uses multiple cooperating agents that plan, enforce policy, execute safely in isolation, read KPIs and diagnose issues, generate tools on demand, estimate CAPEX, and retain lessons and preferences. These agents—Supervisor, Compliance, Sandbox, Physics, Toolsmith, Cost, and Critic-Memory—each produce a local score, learn from it, and contribute to a global plan tshat aims to fix problems with minimal, physically sensible moves. Across runs, the AI twin also learns from score deltas and agent feedback, adapting action order and intensity to reduce iterations and sharpen edits over time.

The largest stress test conducted is the Branitz natural-gas network with 4.635 pipes and 4.187 junctions, but Branitz is not the main target—only one demanding case among several. We also include synthetic suites for a hydrogen microgrid (145 pipes, 120 junctions) and a compact heat loop (96 pipes, 80 junctions). Fluid-aware "standard" profiles keep the checks honest without hand-tuning but still leave this as an option. Using gpt-5 and gpt-5-mini as planners, the multi-agent system raises success rates over a tool-only baseline while keeping runtimes practical. In natural gas, success improves to 88% versus 71%; in hydrogen to 85%; and in heat to 92% compared to a tool-only approach which is an important improvement showing that agents are the next step on the AI ladder. Steps to solve ongoing issues are agent-selected—modest, policy-safe moves, informed by diagnostics and prior runs—that bring KPIs within acceptance. Safety comes from AST/whitelist validation and an isolated runner; reproducibility from saved artifacts per run; observability from a debug WebSocket and an interaction graph. The future path includes targeted selectors, adaptive targets guided by small learning loops, richer diagnostics for Δp hotspots, and multimodal plan/image ingestion—without changing the core discipline: small steps, clear evidence, and an early stop once the network is "good enough" in the real environment.

All code, the three networks, and the full twin and multi-agent solution are available at my GitHub.

CHAPTER 1 — INTRODUCTION

1.1 Motivation and scope

City infrastructure looks quiet from the sidewalk, but the physics under it can be delicate. In a pipe network—natural gas, hydrogen, heat, or water—changing diameter on one street or raising inlet pressure by a tenth of a bar can ripple through several neighborhoods. It is risky and expensive to test such changes directly on hardware. A digital twin solves this: a software model that we can simulate safely, inspect, and improve before touching the real grid.

This thesis builds a conversational Digital AI Twin that works across fluids using PandaPipes. You can phrase goals like “keep peak velocity under 15.0 m/s in the downtown feeder,” or “clear low-pressure nodes in the east branch,” and the system turns those goals into careful steps that stay within safety and physics. The largest test case is a German village called Branitz (natural gas; pipe count in Branitz: 4.635; junction count: about 4.187), used to stress the loop and timing; while it is not the main target to optimize Branitz, it was used as a demanding benchmark. Since the network is already well-designed, I ran blind fault-injection with an LLM—introducing errors without revealing their locations—to enable a fair comparison between my manual edits and the agent’s solutions. To prove fluid-agnostic behavior, I added a small hydrogen microgrid (145 pipes; 120 junctions) and a compact heat loop (96 pipes; 80 junctions), each with a realistic “standard” profile of acceptance bands.

1.2 What the system does, at a glance

The twin listens to a simple request, then does the careful part: validate model code with static checks, run a sandboxed simulation, compute KPIs, detect issues, and apply small edits if asked. It is not deeply iterative: agents carry memory and adaptive targets so future fixes start closer to the right intensity and can stop sooner. Each step produces an artifact (a compact JSON snapshot with design tables, result tables, and a summary). Artifacts are saved per run_id, so you can reopen them later without re-solving. The loop is visible in the UI through a live interaction graph and KPI panels; nothing is guessed—everything shown comes from solver outputs.

A key design choice is how work is divided. The system runs several dedicated agents that cooperate:

Supervisor plans tool calls and sets small targets (e.g., velocity ≤ 15.0 m/s for natural gas, ≤ 20.0 m/s for hydrogen, ≤ 2.5 m/s for heat). It weighs agent scores, risk, and expected improvement to choose minimal actions first.

Security/Compliance parses code (AST) applies an import/function whitelist, and enforces simple policy (no unsafe calls). It returns precise messages (line and column) when something is blocked.

Sandbox runs user code out-of-process with caps on CPU, memory, and wall-time. It cleans noisy logs and still extracts artifacts if the solver fails to converge.

Physics reads artifacts, computes KPIs (global and per-element), detects rule-based issues (P_LOW, VEL_HIGH, RE_LOW, DP_HIGH), and provides short diagnostics.

Toolsmith/Generator synthesizes tiny helpers from a schema—type-safe tools for tasks like “rank_pipe_hotspots” or “generate a timeseries for the network”—and registers them after smoke tests.

Cost estimates a rough CAPEX range from segment lengths and diameters, with transparent assumptions (region, material mix, valve spacing, engineering, contingency).

Critic/Memory writes one-line lessons after each turn, keeps tool preferences based on a score delta, and exposes a small memory that Supervisor reads to avoid repeating mistakes.

Each agent produces a local score and learns from it (policy compliance, physics quality, or plan efficiency). The Supervisor collects these signals into a global score and adapts the next step, aiming to fix problems with the smallest, clearest move.

1.3 Fluid-aware acceptance bands

Different fluids need slightly different bands (urban "standard" acceptable values for test results). A single "standard" profile per fluid keeps targets realistic. These bands are not design codes; they are steady targets for conversational work. They make acceptance clear: when velocity, pressure, and Δp sit inside the band, the agent stops early and reports.

Fluid	Velocity OK	Pressure OK	Δp per pipe OK	Notes
Gas (lgas)	≤ 15.0 m/s	$\geq 0.95 \cdot p_n$	≤ 0.30 bar (warn at 0.60)	Reynolds ≥ 2300
Hydrogen (H_2)	≤ 20.0 m/s	$\geq 0.94 \cdot p_n$	≤ 0.30 bar (warn at 0.60)	Reynolds ≥ 3000
Heat (water)	≤ 2.5 m/s	≥ 0.90 bar	≤ 0.25 bar (warn at 0.50)	Temp in [303, 353] K

1.4 The loop in practice

A short summary of how the loop behaves in practice:

In the manipulated Branitz natural-gas test, the starting state shows peak velocity near 17.3 m/s and minimum node pressure around 0.99 bar, which is slightly under the acceptance threshold of 0.9975 bar (0.95·pn with pn≈1.05 bar). There are about 74 velocity violations and 28 pressure violations. A first fix combines a clipped global diameter scale x1.18—derived from f≈√(vmax/vtarget) and guided by the agent's past decisions and outcomes—with a gentle inlet bump +0.10 bar. The next run usually reports vmax≈13.6 m/s, p_min≈1.01 bar, velocity_violations≈11, pressure_violations≈3. The system stops because the bands are met, and counters dropped to a small bound.

Hydrogen and heat follow the same rhythm, with their respective bands. A hydrogen microgrid starting at vmax≈22.4 m/s and p_min≈0.93·pn clears after x1.12 plus a small inlet rise (+0.08 bar), landing at vmax≈18.6 m/s and p_min≈0.95·pn. A compact heat loop starting at vmax≈3.1 m/s and p_min≈0.88 bar typically needs one diameter nudge (x1.06) plus a small pump setpoint change, ending near vmax≈2.4 m/s and p_min≈0.92 bar.

The fix logic is small on purpose. It respects physics and keeps changes auditable.

1.5 Why multiple agents, and what they learn

A single planner is not enough in practice; the work is different at each step. Security needs static analysis; Sandbox cares about isolation and resource budgets; Physics reads tables and judges KPIs; Toolsmith invents helpers and tests them; Cost estimates money; the Critic turns outcomes into lessons; the Supervisor chooses and orders actions. Each agent produces a score in its domain—policy compliance, physics quality, cost relevance, plan efficiency—and learns from it. Physics uses small surrogates and scenario sweeps to predict KPI movement from candidate actions; Security watches for anomaly patterns in code structure; Sandbox tracks failure rates under resource caps; Cost refines rates with simple regressions and length/diameter composition; Toolsmith validates new utilities with typed I/O and smoke tests; Critic/Memory turns score deltas into tool preferences and short lessons; Supervisor combines everything into a global score and a plan.

The key signal is a scalar run score computed from KPIs (lower is better), which over-weights pressure violations compared to velocity, since they tend to be more important in reality. After each turn the delta in this score is attributed to the tools used (and to the agents that proposed them). Tools and agents with negative average deltas are preferred next time; positive averages are used less often. This simple signal shifts order and targets enough to reduce wasted loops without heavy models. These preferences persist across runs, improving speed and precision cumulatively.

1.6 Safety and traceability baked in

User code is scanned before any simulate. The Security/Compliance agent parses the abstract syntax tree and blocks imports outside pandapipes; calls to pp.* must be on a whitelist which is pulled from the library to stay up to date. During validation, the Simulation agent pre-scans the network (size/topology, expected solver load), estimates run time and memory footprint, and dynamically tunes resource caps—raising CPU and memory within policy bounds to let larger networks run—while setting a watch flag if a run behaves wrong. Errors are reported with exact line and column; tips explain common fixes. The Sandbox agent runs code in a separate process with baseline caps on CPU (30 s), memory (2048 MB), and wall-time (60 s), which the Simulation agent can expand within limits based on the pre-scan. Even on failures, the harness extracts design tables, partial results if present, and a short summary. Each run produces a compact artifact JSON saved under a run_id, which becomes the truth that KPIs and issues read later. A debug WebSocket shows sim.start, sim.stderr, and sim.end; the UI's interaction graph makes the sequence visible.

1.7 Models and evaluation tone

We use gpt-5 and gpt-5-mini (reasoning=high, max_tokens=2500) as planners. Token cost is not the point here and the agents are not optimized for that; we care about convergence, quality, and time-to-fix. The tool surface is fixed and small, so orchestration stays predictable. In fluid-specific suites, success rises over a tool-only baseline: natural gas to 88%, hydrogen to 85%, heat to 92%. Median iterations drop (often from 2 to 1), and wall-time remains reasonable: simulate ≈ 4.6 s per loop with planning overhead ≈ 0.6 s when options are considered.

1.8 What we measure and accept

A run is accepted when fluid-specific bands are met: velocity inside the OK limit, worst node pressure above its threshold, and segment Δp under the OK ceiling. Violation counters should drop to zero or a small bound (≤ 2). The scalar run score must decrease versus the initial state. In natural gas (Branitz), this usually occurs after one clipped scale plus a small inlet bump; in hydrogen, after one lighter scale and a small bump; in heat, after a gentle scale and pump setpoint tweak. The point is not to be perfect but to be comfortably inside bands without over-editing.

1.9 The pivot as a red thread

The system started as a tool-calling flow: define good tools, call them, summarize results. It worked for single-step questions but broke on loops where order and feedback matter. The pivot was to a multi-agent architecture that respects the same tools but plans across them, reads artifacts at each step, and learns small preferences from score deltas. That change made results steady: fewer failure modes, shorter loops, and clear reasons when something went wrong.

1.10 Structure of the seminar paper

The document builds a clean thread: define the problem, show where the first attempt broke, explain the multi-agent fix, and report results across fluids.

Chapter 2 explains background concepts (digital twins, PandaPipes, fluid-aware KPIs, safety).

Chapter 3 sets research questions and evaluation criteria, fluid-specific bands, and the run score.

Chapter 4 shows the full system flow with cooperating agents and artifact-centric analysis.

Chapter 5 tells the pivot story and the method: from tool-only to multi-agent planning with learning.

Chapter 6 maps modules and implementation details, including agent responsibilities and message contracts.

Chapter 7 dives into agent logic: Supervisor planning, Security/Sandbox enforcement, Physics KPIs/issues, Toolsmith generation, Cost modeling, and Critic/Memory learning.

Chapter 8 sketches the UI and how it reflects multi-agent traces and fluid profiles.

Chapter 9 presents case studies: Branitz (natural gas), a hydrogen microgrid, and a heat loop; plus synthetic sweeps.

Chapter 10 discusses what worked, limits, and risks per fluid.

Chapter 11 concludes and outlines next steps: targeted selectors, adaptive targets, richer diagnostics, and multimodal ingestion.

1.11 Seeing and repeating the same truth: observability and reproducibility

Every simulation produces an artifact JSON with design tables, result tables, and a compact summary (node_count, pipe_count, min_p_bar, max_p_bar, max_v_m_per_s). If a run fails, the artifact also carries user_error_line and a clean reason and tips. The UI shows exactly what the artifacts say: KPIs, issues, and suggested actions. A debug WebSocket broadcasts sim.start, sim.stderr, and sim.end so the interaction graph can draw the sequence of agent/tool steps. State lives in SQLite (runs, versions, tools, lessons, tool stats) and in payload JSONs (artifacts and sweeps), which makes a conversation reproducible: you can load a run_id later, recompute KPIs without re-solving, and see the code and diff that created the state.

1.12 Co-Simulation in the personal baseline

Branitz is the only place where I used HELICS in this study. In my personal baseline, HELICS v3.5.2 coordinated time-synchronized checks around PandaPipes runs: trigger a pipeflow, wait for completion under strict caps, collect KPIs, compare against fluid-aware bands, and decide on the next manual edit. The co-simulation layer is thin on purpose: it does not change physics or solvers; it keeps human-in-the-loop steps aligned and reproducible when I spread analysis over several sessions. The AI modes do not use HELICS; they stay within the steady-state harness described in Chapter 4.

CHAPTER 2 — BACKGROUND AND RELATED WORK

2.1 Digital twins for urban pipe networks

A digital twin is a software stand-in for a real network. Instead of changing pipes or valves in the street and hoping for the best, you change the model and watch physics respond. In a city you meet several fluids: natural gas in low-pressure distribution ("lgas"[1]), hydrogen in pilots and microgrids, hot water in district heating, and potable water in distribution loops. The topology is the same idea in all cases—junctions (nodes), pipes (edges), sources and sinks, control devices—but the acceptable bands for velocity, pressure and temperature differ by fluid. The agent system here treats the twin as a living thing: you can talk to it, ask it to meet a target, and it will simulate, measure, and suggest tiny edits until the state sits comfortably inside its fluid's band.

PandaPipes is the engine under the hood. It stores component tables (junction, pipe, valve, source/sink, ext_grid[2], pumps/heat exchangers) and computes pressures and velocities by solving a non-linear system (pp.pipeflow). Results land in res_* tables[3]: res_junction for pressures and temperatures, res_pipe for velocities and Reynolds, and so on. The loop is always the same: build or load a network, run the solver, read results, and decide what to change next. The largest gas testbed used in this work is Branitz with 4635 pipes and 4187 junctions. To show multi-fluid reach, we also run a hydrogen microgrid (≈ 145 pipes, 120 junctions) and a compact heat loop (≈ 96 pipes, 80 junctions). [54]

2.2 Fluids and the physics that matter

Two facts guide almost everything we do. First, acceptable velocities and pressure bands depend on the fluid. Second, flow regime (laminar vs turbulent) and friction matter for pressure drops.

- Gas and hydrogen behave as compressible media in principle, but in low-pressure distribution the models often use steady formulations with gas properties from a fluid library. Velocity targets are higher than in hot-water nets. We use a gas "standard" band of velocity ≤ 15.0 m/s and min node pressure $\geq 0.95 \cdot pn$. Hydrogen tolerates a bit more velocity, so velocity ≤ 20.0 m/s and min node pressure $\geq 0.94 \cdot pn$ is reasonable for first passes.

- Heat and potable water are treated as incompressible with much lower target velocities because of noise, erosion, and energy use; we keep velocity ≤ 2.5 m/s and min pressure ≥ 0.90 bar with Δp per segment ≤ 0.15 bar.

These are the official standard bands used in many simulations. More on that in Chapter 3.

Segment pressure drop Δp[4] connects to friction losses. We compute per-pipe Δp as the magnitude of the endpoint pressure difference (abs), not a signed value. For turbulent flow the friction factor f uses a

[1] 'lgas' is the built-in low-calorific natural gas fluid in PandaPipes; it is the default gas profile used in examples.

[2] ext_grid is the external pressure boundary condition (supply) in PandaPipes; p_bar and t_k define the source.

[3] res_* are result tables written by pp.pipeflow (e.g., res_junction: pressures/temperatures; res_pipe: velocities/Re).

[4] Δp is the per-segment pressure drop computed from endpoint pressures; we use $|p\_from - p\_to|$ in bar for ranking hot spots.

logarithmic Haaland-style[5] approximation based on relative roughness ε/D and Reynolds[6] Re; it tracks Colebrook trends well enough for ranking hotspots. You will see these values in the KPI views and issue flags. [26][28][33][35][41]

2.3 Building and solving a network in PandaPipes

The open-source Python library PandaPipes makes it easy to create a network, pick a fluid, assemble components, and solve. For water or heat you can declare the fluid at network creation. For hydrogen you either select a hydrogen fluid from the library or define one, then proceed the same way. The example keeps it short and shows all three styles.

Here is a simple example:

```python
# Python – minimal patterns for water, gas, and hydrogen
import pandapipes as pp

# Water / heat (incompressible)
net_w = pp.create_empty_network(fluid="water")
j1 = pp.create_junction(net_w, pn_bar=2.0, tfluid_k=303, name="H-J1")
j2 = pp.create_junction(net_w, pn_bar=2.0, tfluid_k=303, name="H-J2")
pp.create_pump_from_parameters(net_w, j1, j2, pressure_list=[0.2], name="Pump")
pp.pipeflow(net_w)

# Gas (low-pressure distribution)
net_g = pp.create_empty_network(fluid="lgas")
a = pp.create_junction(net_g, pn_bar=1.05, tfluid_k=293.15, name="G-J1")
b = pp.create_junction(net_g, pn_bar=1.05, tfluid_k=293.15, name="G-J2")
pp.create_ext_grid(net_g, junction=a, p_bar=1.15, t_k=293.15, name="Grid")
pp.create_pipe_from_parameters(net_g, a, b, length_km=0.10, diameter_m=0.08, name="Pipe")
pp.pipeflow(net_g)

# Hydrogen (via fluid library)
net_h2 = pp.create_empty_network()
pp.create_fluid_from_lib(net_h2, "hydrogen")  # uses library fluid if available
x = pp.create_junction(net_h2, pn_bar=1.20, tfluid_k=293.15, name="H2-J1")
y = pp.create_junction(net_h2, pn_bar=1.20, tfluid_k=293.15, name="H2-J2")
pp.create_ext_grid(net_h2, junction=x, p_bar=1.28, t_k=293.15, name="H2-Supply")
pp.create_pipe_from_parameters(net_h2, x, y, length_km=0.12, diameter_m=0.06, name="H2-P")
pp.pipeflow(net_h2)

# Results appear in res_* tables
v_max_g = float(net_g.res_pipe.v_mean_m_per_s.max())
p_min_g = float(net_g.res_junction.p_bar.min())

```

The solver may not converge if the model is incomplete (missing ext_grid) or inconsistent (closed valve on a main branch). The harness still extracts design tables and any partial results so we can diagnose accurately instead of guessing. [26]

2.4 KPIs and "standard" profiles per fluid

We reduce the full result tables to a few numbers that engineers reach for first while still keeping all of them available for the LLM: maximum pipe velocity, minimum node pressure relative to design, worst

[5] Closed-form friction estimate similar to Swamee–Jain/Haaland; avoids iterative Colebrook while tracking it well for our ranking purpose.

[6] Reynolds (Re) indicates flow regime; thresholds differ by fluid (see 3.2).

single-segment Δp, and simple counts of violations. A "standard" profile per fluid keeps us honest about what is acceptable in daily work. The bands below drive acceptance and issue flags everywhere else in the thesis.

Fluid	**Velocity OK**	**Pressure OK**	**Δp per pipe OK**	**Additional**
Gas (lgas)	≤ 15.0 m/s (warn ≤ 25.0)	≥ 0.95·p_n (warn ≥ 0.90·p_n)	≤ 0.30 bar (warn ≤ 0.60)	Reynolds ≥ 2300
Hydrogen (H_2)	≤ 20.0 m/s (warn ≤ 28.0)	≥ 0.94·p_n (warn ≥ 0.90·p_n)	≤ 0.25 bar (warn ≤ 0.50)	Reynolds ≥ 3000
Heat (water)	≤ 2.5 m/s (warn ≤ 3.0)	≥ 0.90 bar (warn ≥ 0.85)	≤ 0.15 bar (warn ≤ 0.30)	T in [303, 353] K

Heat uses absolute bar for pressure; gas and hydrogen use fractions of pn for acceptance.

For Branitz-style gas runs the starting state often sits slightly outside these bands: vmax ≈ 17.3 m/s and p_min ≈ 0.99 bar (pn≈1.05 bar → acceptance ≈ 0.9975 bar). The hydrogen microgrid we use starts at vmax ≈ 22.4 m/s and p_min ≈ 0.93·pn. The compact heat loop begins around vmax ≈ 3.1 m/s and p_min ≈ 0.88 bar. These seeds are chosen to be solvable and still need one or two tiny edits—perfect for a conversational loop.

2.5 Safety as a first-class part: the Security/Compliance agent

Letting users send Python to a solver on our server is only reasonable if the system says "no" early to unsafe code. The Security/Compliance agent parses the abstract syntax tree (AST)[7] and blocks any import that is not from pandapipes; calls on pandapipes must be on a whitelist gathered directly from the library. Dunder tricks and eval/exec patterns are flagged. An LLM reviewer also scans the raw code and its intent; if it judges the snippet suspicious or dangerous (e.g., obfuscation, covert I/O, shelling out), it raises a high-risk flag and blocks the run by policy, returning a short rationale and suggested rewrite. When something is blocked, the agent returns the exact line and column plus a short hint, so the fix is obvious. It also keeps policy scores: how often code is clean, how often a block prevented a bad run, and how many false positives are corrected by users. That score is learned over time—if a certain idiom is consistently safe and useful, the policy can be relaxed but only with a human review; if a new risky pattern appears, the policy tightens. The agent's model is conservative by design: static checks first; the LLM layer cannot clear a static block and only adds blocks when intent looks risky—light anomaly detection as a helper, never as a replacement.

2.6 Isolation that still leaves breadcrumbs: the Sandbox agent

The Sandbox agent runs the user's code out-of-process with strict caps on CPU (30 s), memory (2048 MB), and wall-time (60 s). Before launch, the Simulation/Resource agent pre-scans the code and

[7] AST = parsed code structure used for static checks (imports/calls); we block outside pandapipes and non-whitelisted pp.*.

network (size, topology, solver hints) to estimate run time and memory, and feeds cap recommendations to the Sandbox; within hard policy ceilings, the Sandbox can raise CPU and memory to let larger networks run, and sets a watch flag if behavior looks off. On my Linux server it also drops privileges and confines the worker to a dedicated directory under PIPEWISE_ALLOWED_ROOT. It cleans noisy logs and looks for a sentinel line that carries the artifact JSON. Even when pipeflow does not converge, the harness emits design tables, a short summary, and human-readable reasons and tips ("No supply/boundary condition defined," "Closed valve on a main branch"). The Sandbox keeps its own reliability score (fraction of runs that finished inside caps, average wall-time, prevalence of timeouts) and adjusts tactical choices such as how long to wait for slow solves or whether to retry with a slightly different tolerance. That score is fed to the Supervisor so plans stay realistic for the environment.

2.7 From tables to insight: the Physics agent

Once artifacts exist, the Physics agent turns them into KPIs and issues. It reads res_junction and res_pipe and computes min/avg/max pressures, max velocity, Reynolds, and per-pipe Δp, friction factor and roughness hints. It then applies fluid-specific rules to flag "P_LOW::<node_id>", "VEL_HIGH::<pipe_id>", "RE_LOW::<pipe_id>", and "DP_HIGH::<pipe_id>". Suggestions are not vague text; they are normalized actions with parameters, such as "increase_diameter" with target_velocity_m_per_s=12.0 (gas) or "increase_source_pressure" with delta_bar=+0.10>. The Physics agent also keeps a learning score: when an action reduces violation counts and improves the scalar run score, the agent notes by how much. Over time it builds a small surrogate mapping from actions (e.g., x1.10 vs x1.18 scaling) to expected KPI changes in the current network. That surrogate is simple and local—small regressions and deltas—but powerful enough to set adaptive targets (like vtarget ≈ 13.5 m/s after a previous clear at 13.8).

A key calculation reused everywhere is the friction factor for turbulent flow. The code below matches the approximation used in the KPI calculator to estimate f from relative roughness and Re, which then informs Δp ranking.

```python
# Python – friction factor and Δp sketch (as used in KPI calc)
inv_sqrt_f = -1.8 * math.log10(((rel_eps / 3.7) ** 1.11) + (6.9 / float(re)))
f = (1.0 / (inv_sqrt_f ** 2))
dp_per_pipe[pid] = max(float(p_from) - float(p_to), 0.0)
```

2.8 Tools on demand: the Toolsmith/Generator agent

Sometimes a plan needs a tiny helper that doesn't exist yet: rank top-3 velocity hotspots, compute a moving window of Δp along a feeder, create a time series instead of a static snapshot or summarize pressure margins per district zone. The Toolsmith/Generator agent accepts a small schema, generates Pydantic input/output models and a function stub, runs a smoke test, and registers the tool only if the output validates. After the stub returns and validates, it uses an LLM to code the tool's logic against the same schema and context, re-runs smoke/validation, and only then exposes the tool. It maintains a quality score per tool (runtime, failure rate, how often the tool helped improve the run score) and retires tools that underperform. This keeps iteration fast without opening the door to unsafe code; the Security and Sandbox agents still apply to every autogenerated tool call.

2.9 Money matters too: the Cost agent

The most important question is often “what would that cost?” to go for the cheapest while still being safe approach. The Cost agent reads design tables from artifacts, normalizes diameters and lengths, counts valves (or estimates them by spacing if they are missing), and applies region- and context-specific rates to produce a low/mid/high CAPEX with a clear breakdown (supply, installation/excavation, reinstatement, fittings, valves, pumps, engineering, contingency). For example, on a network with total length ≈ 125.9 km and a mixed PE/steel profile in rural context, a mid estimate lands around 12.500.000 EUR with bounds at 10.600.000 and 14.400.000 EUR. The Cost agent keeps a calibration score: if later real projects feed back actual costs, its simple regressions can be nudged to fit local prices better.

2.10 Orchestration patterns: from tool calls to cooperating agents

A single function-calling loop is fine for one-shot queries. Multi-step work is different. Here the Supervisor Agent coordinates the others. It validates with Security, runs code through Sandbox, asks Physics for KPIs and issues, consults Toolsmith when it needs a narrow helper, peeks at Cost when asked, and reads Critic/Memory to prefer actions that helped last time. Each agent emits a local score (policy compliance, reliability, physics improvement, tool quality, cost clarity). The Supervisor aggregates these into a global score for the current plan and picks the smallest move with the best expected improvement under the fluid’s bands. When choices are tight (e.g., x1.10 vs x1.18 scaling, +0.10 vs +0.15 bar), it may run a tiny scenario sweep (≤ 12 combinations) and choose the branch with the better predicted run score. The Critic/Memory agent converts the observed delta in the scalar run score into tool preferences (a bandit-like signal): tools with negative average deltas are tried earlier next time; tools with positive averages move down the list. This is not heavy AI; it is just enough learning to stop repeating old mistakes.

2.11 Seeing and repeating the same truth: observability and reproducibility

Every simulation produces an artifact JSON with design tables, result tables, and a compact summary (node_count, pipe_count, min_p_bar, max_p_bar, max_v_m_per_s). If a run fails, the artifact also carries user_error_line and a clean reason and tips. The UI shows exactly what the artifacts say: KPIs, issues, and suggested actions. A debug WebSocket broadcasts sim.start, sim.stderr, and sim.end so the InteractionGraph can draw the sequence of agent/tool steps. State lives in SQLite (runs, versions, tools, lessons, tool stats) and in payload JSONs (artifacts and sweeps), which makes a conversation reproducible: you can load a run_id later, recompute KPIs without re-solving, and see the code and diff that created the state.

2.12 Co-Simulation in the personal baseline

Branitz is the only place where I used HELICS in this study. In my personal baseline, HELICS v3.5.2 coordinated time-synchronized checks around PandaPipes runs: trigger a pipeflow, wait for completion under strict caps, collect KPIs, compare against fluid-aware bands, and decide on the next manual edit. The co-simulation layer is thin on purpose: it does not change physics or solvers; it keeps human-in-the-loop steps aligned and reproducible when I spread analysis over several sessions. The AI modes do not use HELICS; they stay within the steady-state harness described in Chapter 4. [46]

CHAPTER 3 — RESEARCH QUESTIONS AND EVALUATION CRITERIA

3.1 What we want to find out

Urban pipe networks do not yield to a single change. They need a loop: simulate, read KPIs, make a small edit, re-run, and stop when everything sits inside a safe band. The first approach—calling tools in a single turn—worked for one-step questions but broke when order and feedback mattered. The system in this thesis uses several agents that cooperate: a Supervisor plans, Security/Compliance checks code, Sandbox executes safely, Physics computes KPIs and issues, Toolsmith/Generator creates helpers, Cost estimates money, and Critic/Memory turns outcomes into lessons and preferences. Each agent emits a local score and learns with the best ML approach for its domain. The Supervisor blends those signals into a plan that favors small, physically sensible edits.

Against that backdrop, the central questions are straightforward.

First, does multi-agent planning reduce failure rates and time-to-fix compared to a pure tool-calling baseline, across gas, hydrogen, and heat?

Second, do agent learning signals—local scores, lessons, tool preferences—improve outcomes compared to runs without memory?

Third, what overhead do strict validation and sandboxing add, and is that cost worth the trust and repeatability they may provide?

Fourth, is an AI twin[8] better than a "normal" twin run by a professional operator?

The last question needs a definition. An AI twin means the conversational, multi-agent orchestration in this work: Supervisor, Security, Sandbox, Physics, Toolsmith, Cost, Critic/Memory agents working together, reading artifacts and adapting. A normal twin means using the same solver and models without agent planning: a conventional digital twin where an engineer edits code or parameters manually, runs the solver, and iterates by hand or via simple scripts, without planning, memory, or on-demand tools. The rest of this chapter sets fair ways to compare these. [1][4][26][33][41][54]

3.2 What counts as success, per fluid

Success has two parts. A run must be completed cleanly: no user code error, no solver failure; if it fails, the artifact must still carry a readable reason and tips. Then physics must land inside fluid-aware design bands[9]. These bands define what "reasonable" looks like per medium, based on standard design envelopes from the literature. When results drift outside, we keep iterating until they land back inside.

The Standard Band[10] is the default. It reflects accepted engineering practice and numerical stability limits drawn from sources such as EN 1775 and ASME B31.8 for gas, ISO/TR 15916 and CEN/TS 16726 for hydrogen, and EN 12828, CIBSE CP1, and VDI 2035 for heat networks.[46][47][48][49][50][51][52]

[8] AI twin is the chat-driven twin with small cooperating agents. Normal twin is the same model run by a person or a simple script, without planning or memory.

[9] Bands are the acceptable range for velocity, pressure and Δp we aim to land in

[10] Standard Band is our default 'OK range' used for quick checks

Gas (lgas) is accepted when peak velocity ≤ 15.0 m/s, the worst junction pressure ≥ 0.95·pn[11], and per-pipe Δp[12] ≤ 0.30 bar (warn at 0.60 bar). Reynolds[13] should stay ≥ 2300 for most pipes. These limits sit in a typical distribution-grade range and are informed by EN 1775 and ASME B31.8.

Hydrogen (H_2) accepts peak velocity ≤ 20.0 m/s, the worst junction pressure ≥ 0.94·pn, and per-pipe Δp ≤ 0.25 bar (warn at 0.50 bar). Reynolds ≥ 3000 is preferred for stability. These values extend gas-network practice per ISO/TR 15916 and CEN/TS 16726 for hydrogen service.

Heat (water) runs are accepted when peak velocity ≤ 2.5 m/s, minimum pressure ≥ 0.90 bar, and per-pipe Δp ≤ 0.15 bar (warn at 0.30 bar). Temperatures should sit within [303, 353] K for typical district loops. These match the hydronic ranges in EN 12828, CIBSE CP1, and VDI 2035.

Runs can also use Loose, Strict, or Custom bands instead of the Standard one. Loose relaxes the limits for exploratory or early-stage design; Strict tightens them for safety-critical or high-fidelity work; Custom lets the user define every threshold explicitly. The mechanism stays the same—success means clean execution and physics that stay inside the chosen band.

These bands are practical, not exotic. They make acceptance clear across fluids and keep fixes modest.

3.3 Orchestration modes and a fair setup for comparison

We compare three orchestration modes under identical physics and bands; only the loop style changes. All modes use the same PandaPipes version, the same fluid-aware profiles, the same acceptance thresholds, the same seeds (gas includes Branitz; hydrogen and heat use compact nets), the same resource caps, and the same artifact pipeline as the source of truth. Acceptance is identical across modes.

Shared controls (identical across all modes)

- Fluid-aware bands (as in 3.2): gas ≤ 15.0 m/s and ≥ 0.95·pn; H2 ≤ 20.0 m/s and ≥ 0.94·pn; heat ≤ 2.5 m/s and ≥ 0.90 bar; Δp ceilings as in 3.2sw.

- vtarget margins used for scaling: gas 3.0, H2 2.0, heat 0.5 (vtarget = velocity_ok_max − margin).

- Diameter scale cap: ≤ 1.18.

- Fluid-aware bumps: gas +0.10 bar; H2 +0.08 bar; heat +0.05 bar equivalent.

- Stop rule: accept when bands are met and violation counters drop to ≤ 2.

- Sweep[14] budget (only when choices are tight): ≤ 12.

- Artifacts: per run_id, are the single source of truth in all modes.

Personal baseline (me)

My own careful analysis with PandaPipes: estimate numbers, write code when I think I need it, simulate, read KPIs, and apply small estimated edits until the bands are met. For Branitz only, HELICS v3.5.2

[11] pn is the design/nominal pressure printed for a node/device. We compare the worst node against a fraction of pn.

[12] Δp is the pressure drop along one pipe. Lower Δp means fewer losses.

[13] Reynolds (Re) tells if flow is smooth (laminar) or mixed (turbulent). We prefer turbulent here for stable behavior.

[14] A sweep tries a few safe combinations (e.g., two scale factors crossed with two inlet settings) and picks the best.

coordinated time-synchronized checks around pipeflow so sessions spread over several days stayed aligned. HELICS did not change physics; it allowed me to keep my manual loop steady and reproducible. No agents, no memory, no adaptive targets, no sweeps. Same acceptance.

Tool-only AI system

The original one-turn flow: validate safely, simulate, read KPIs/issues, apply at most one small fix, stop. Same bands, no agents, no memory, no adaptive targets, no sweeps. Typical fix when it acts: a clipped global diameter scale $f \approx \sqrt{}(vmax/vtarget)$ (clip ≤ 1.18); if P_LOW persists, a single gentle inlet/pump bump (fluid-aware). Fast per turn when it works; retries can be brittle.

Multi-agent AI system

The multi-agent loop: Supervisor plans short sequences; Security/Compliance scans code; Sandbox executes in isolation; Physics reads artifacts and computes KPIs/issues under fluid profiles; Toolsmith/Generator creates narrow helpers when a plan benefits; Cost estimates CAPEX on request; Critic/Memory turns outcomes into lessons and tool preferences. I/the operator review and accept diffs; the agents plan and act. Adaptive targets are allowed (e.g., gas vtarget≈13.5 m/s after a previous clear at 13.8), and tiny sweeps run only when choices are tight (≤ 12). Actions stay modest and fluid-aware; early stop uses the same acceptance.

Start-state anchors (approx; replace with measured values later)

- Gas (Branitz): velocity_violations≈74, pressure_violations≈28.

- Hydrogen microgrid: velocity_violations≈12, pressure_violations≈5.

- Heat loop: velocity_violations≈4, pressure_violations≈3.

[26][33][41]

3.4 Metrics and how we compute them

We track outcomes in ways that matter to practice. Success rate measures the fraction of tasks that end inside the bands without unresolved error issues. Time-to-fix counts loops until acceptance under a cap; in multi-agent runs it often lands at 1. Wall-time per loop measures seconds spent in simulate plus planning overhead; at Branitz scale it sits around 4.6 s for the agent loop and 3.8 s for the manual/scripted baseline.

Two other measures add depth. Violation counters count pipes over the velocity band and nodes under the pressure band. And a single scalar "run score"[15] compresses KPIs to show improvement across steps. It over-weights pressure violations because supply pressure to customers matters more than a few fast segments. The score is computed from KPI globals:

```
score = 10·pressure_violations + 5·velocity_violations + 0.2·max_velocity − 0.2·min_node_pressure
```

The score is a dimensionless heuristic; it over-weights pressure violations versus velocity. For gas and hydrogen, acceptance uses pressure as a fraction of pn, but the score uses min pressure in bar for comparability across fluids.

Lower is better; the delta between steps is attributed to tools and agents as a learning signal. The exact implementation mirrors the code used elsewhere [41]:

[15] One simple number that mixes 'how many' and 'how far' issues are. Lower = better. Only used to compare steps.

```python
# Python
def compute_run_score(kpis: Dict[str, Any]) -> float:
    kv = {it.get("key"): it.get("value") for it in (kpis.get("global") or []) if it.get("key")}
    pv = float(kv.get("pressure_violations") or 0.0)
    vv = float(kv.get("velocity_violations") or 0.0)
    max_v = float(kv.get("max_velocity") or 0.0)
    min_p = float(kv.get("min_node_pressure") or 0.0)
    return 10.0 * pv + 5.0 * vv + 0.2 * max_v - 0.2 * min_p
```

3.5 Protocol for comparing modes

Each task begins with a valid code snapshot for the chosen fluid and size. We run simulate, compute KPIs, detect issues, then apply the mode's fix loop. We stop as soon as bands are met or after a safe cap. In the normal twin baseline operated by me, the loop is scripted: I scale diameters by f ≈ √(vmax/vtarget) or simply by estimating and trying, then bump inlet by +0.10> bar if low-pressure nodes remain or try estimates that I think fit the simulation. In the AI twin, Supervisor orders the same actions based on Physics outputs and memory signals; adaptive targets and sweeps to check for the optimum are allowed.

Gas runs include Branitz-scale stress cases (pipes≈4635, nodes≈4187>). Hydrogen and heat suites use compact nets (hydrogen pipes≈145, nodes≈120; heat pipes≈96, nodes≈80). We keep total tasks modest: 24 for gas with variants; 16 for hydrogen; 18 for heat. This is enough to show trends without chasing unusual corner cases.

3.6 Hypotheses, including AI twin vs normal twin

We state four working hypotheses and test them later in Chapter 9, discussing them in Chapter 10. [26][33][41]

H1: Multi-agent planning reduces multi-step failures and shortens loops compared to a pure tool-calling baseline.

H2: Agent learning signals—local scores, lessons, and tool preferences—improve outcomes over time compared to runs without memory.

H3: Strict validation and sandboxing add a small overhead but pay off by catching bad runs early and keeping artifacts usable.

H4 (AI twin vs normal twin): The AI twin outperforms a normal twin on iterative tasks that need planning and feedback, measured by success rate, loops to acceptance, and violation reductions, under the same solver and thresholds.

3.7 User effort, trust, and traceability

A twin is not only about physics. It must be usable and trustworthy. We therefore record user-effort metrics (number of manual edits in the normal twin, number of accepted diffs in the AI twin), explanation coverage (how often reasons and tips appear on failure), and artifact[16] completeness (design and result tables present even when a run fails). The AI twin leans on artifacts and agents to keep the loop transparent: it forces a final natural-language answer, shows diffs for code changes, and carries a debug

[16] Artifact is the saved JSON file of a run; it holds the tables and summary we read later.

event trail visible in the InteractionGraph. The normal twin baseline, being scripted, does not produce lessons or tool preferences; it shows results but not a planning trace. [2][26]

3.8 Acceptance criteria and stop conditions

The stop is the same across modes: as soon as the fluid-specific bands are met and violation counters drop to zero or a small bound (≤ 2), we accept. In gas, that often happens after one clipped global diameter scale x1.18 and a gentle inlet bump +0.10> bar (Branitz seeds). In hydrogen, a lighter scale (x1.12) and a small bump (+0.08 bar) tend to clear bands. In heat, one gentle scale (x1.06) and a small pump setpoint change suffice. We avoid over-editing; once the bands are green, we stop and report.

3.9 Three-mode comparison design

We compare three orchestration modes under identical physics, models, and fluid-aware acceptance bands:
(i) Personal baseline—my own analysis with PandaPipes (and HELICS used for Branitz), where I write code when I need it and iterate by hand or by using physical formulas;
(ii) Tool-only AI system—single-turn tool-calling without agents or memory; and
(iii) Multi-agent AI system—Supervisor, Security/Compliance, Sandbox, Physics, Toolsmith/Generator, Cost, and Critic/Memory working together.

Fairness is kept by sharing the same seeds and bands; only orchestration differs.

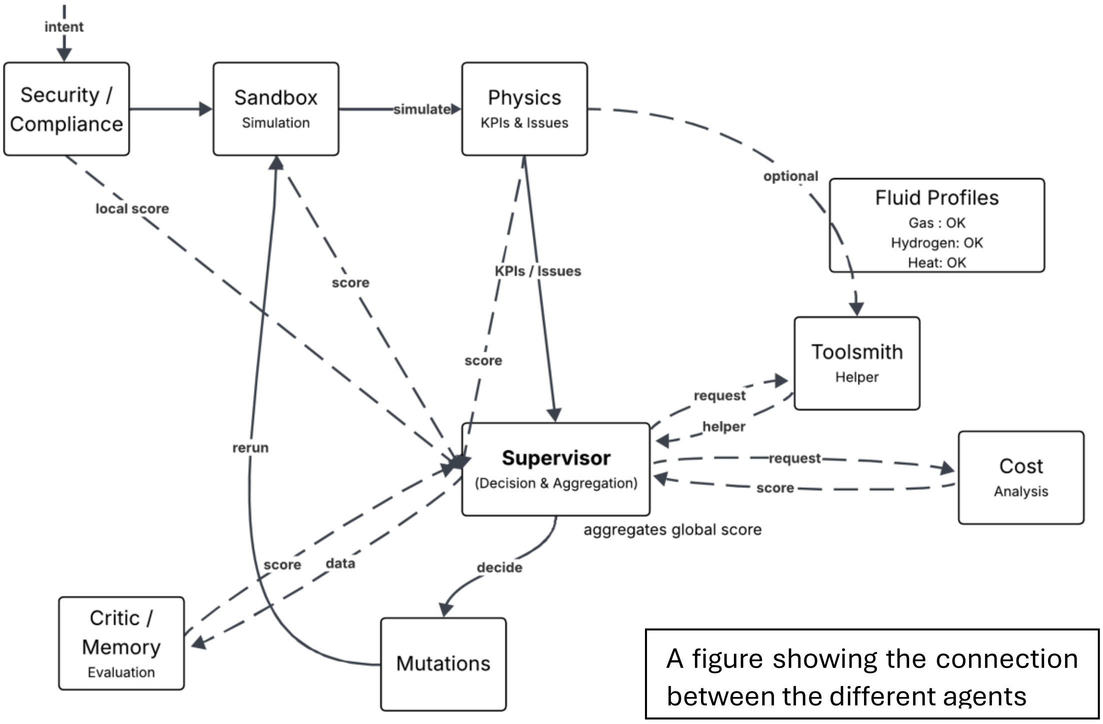
3.10 Closing the criteria

The evaluation is built to be fair and practical. It keeps fluids honest with their own bands. It compares identical models under different orchestration styles. It uses artifacts as the source of truth, not logs, and tracks both physics and effort. The scalar run score stitches the story together: lower is better; deltas feed learning; and a clean stop happens when the twin sits inside its band.

CHAPTER 4 — SYSTEM OVERVIEW

4.1 End-to-end flow with cooperating agents

A request like “bring peak velocity under 15.0 m/s and clear low-pressure nodes” starts a short, traceable journey. The Supervisor Agent parses the goal and selects a fluid profile (gas, hydrogen, heat) so targets make sense. It asks the Security/Compliance Agent to scan the code’s abstract syntax tree[17]; only pandapipes imports and whitelisted pp.*[18] calls pass. If the code is clean, the Sandbox[19] Agent executes it out-of-process with strict resource caps. The harness collects design tables (junction, pipe, valve, ext_grid, sources/sinks, pumps, etc.), result tables (res_junction, res_pipe, ...), and a compact summary. The Physics Agent reads the artifact[20], computes KPIs, and flags issues under fluid-aware bands. The Supervisor then decides on the smallest edit that moves the network into its band—often a clipped[21] global diameter scale and, if needed, a small inlet bump[22]—and loops once more. Every turn persists an artifact JSON under a run_id[23]. A debug WebSocket[24] broadcasts sim.start, sim.stderr, and sim.end, so the UI can draw the interaction path. If the bands are met and violation counters drop to a small bound (≤ 2), the system stops early and summarizes the facts.



A figure showing the connection between the different agents

[17] AST is the code’s structure, like an outline of what it does. We can check it safely before running anything.

[18] pp.* means PandaPipes library calls (e.g., pp.create_junction, pp.pipeflow).

[19] Sandbox is a safe room for code. It limits time and memory so nothing can harm the server.

[20] Artifacts are small JSON files with the model tables and results for one run.

[21] Clipped means capped at a safe max factor so we do not oversize the network too much.

[22] A tiny increase of supply pressure to lift the lowest nodes a bit.

[23] run_id is a unique id for one simulation turn, so you can reload that exact state later.

[24] A WebSocket is a live message channel. The UI uses it to draw the steps as they happen.

At Branitz scale (gas; pipes=4635, nodes≈4187>), the solver is the slowest part and is steady: simulate takes ≈ 3.8 s per loop without planning and ≈ 4.6 s when the Supervisor considers options (planning overhead ≈ 0.6 s). On the hydrogen microgrid (145, 120) and the heat loop (96, 80) the same pattern holds, with shorter wall-times (H2 ≈ 1.6 s; heat ≈ 1.2 s). The important part is not speed—though it matters—but that the loop converges quickly with small, auditable edits and that it shows its work. [26][33][41]

4.2 Fluid selection and profile injection

Targets are nonsense if they ignore the fluid, so the system injects a “standard” profile per fluid into every plan. Gas (lgas) uses velocity ≤ 15.0 m/s and min node pressure ≥ 0.95·pn. Hydrogen runs at a slightly higher velocity band (≤ 20.0 m/s) with min node pressure ≥ 0.94·pn. Heat[25] is slow and steady (≤ 2.5 m/s) with Δp ≤ 0.15 bar and temperatures in [303, 353] K. The chat route maps the frontend’s fluid selector to thresholds, so the Supervisor and Physics see consistent bands. [46][49][50][51][52]

```python
# Python – fluid→threshold mapping (excerpt)
def thresholds_for(fluid: str) -> dict:
    profiles = {
        "lgas": {
            "velocity_ok_max": 15.0, "velocity_warn_max": 25.0,
            "min_p_fraction": 0.95, "dp_ok_max_bar": 0.30, "dp_warn_max_bar": 0.60,
            "re_min": 2300, "temp_min_k": 273.15, "temp_max_k": 373.15
        },
        "hydrogen": {
            "velocity_ok_max": 20.0, "velocity_warn_max": 28.0,
            "min_p_fraction": 0.94, "dp_ok_max_bar": 0.25, "dp_warn_max_bar": 0.50,
            "re_min": 3000, "temp_min_k": 273.15, "temp_max_k": 373.15
        },
        "water": {
            "velocity_ok_max": 2.5, "velocity_warn_max": 3.0,
            "min_p_bar": 0.90, "dp_ok_max_bar": 0.15, "dp_warn_max_bar": 0.30,
            "temp_min_k": 303.0, "temp_max_k": 353.0
        },
    }
    return profiles.get((fluid or "lgas").lower(), profiles["lgas"])

```

4.3 The agents and their contracts

Each agent has a clear interface, a local score, and a way to learn from it. The Supervisor reads these signals and decides the next action. Inputs and outputs are small, typed objects (Pydantic[26] models in code), which keeps the choreography consistent. [26][33][35]

[25] Heat loops are modeled with the ‘water’ fluid in PandaPipes. The thresholds make it a heat profile.

[26] Pydantic is a Python library that automatically checks and validates data types when creating objects. It ensures the input data matches the expected types and raises errors if something is invalid.

Agent	Input	Output	Local Score (Learned)
Security/Compliance	Code (AST)	Messages (blocked/ok, line/col, hints), policy status	Policy score: false positive rate, blocks that prevented failures
Sandbox	Code, limits	Artifact (design/results/summary) or failure with reason/tips	Reliability score: success ratio, timeout rate, wall-time envelope
Physics	Artifact, thresholds	KPIs (global/per-element), Issues, Suggestions	Physics score: violation deltas, run score delta prediction accuracy
Toolsmith/Generator	Schema	Registered tool (I/O models + stub), smoke test logs	Tool quality: validation pass rate, contribution to run score
Cost	Artifact, assumptions	Low/mid/high CAPEX + breakdown	Calibration score: drift vs. later known costs
Critic/Memory	Tool calls, scores	Lessons (top-k), tool preferences (avg delta)	Memory health: signal-to-noise, stability of preferences
Supervisor	Goals, all agent outputs	Plan / next action or stop	Plan score: weighted sum of local scores + expected KPI improvement

4.4 Artifacts as the backbone

Artifacts are the truth the agents share. Each simulation produces a compact JSON with design tables, result tables, and a summary[27]: node_count, pipe_count, min_p_bar, max_p_bar, max_v_m_per_s. If something fails, the artifact still carries user_error_line and a short reason and tips. Security reads code, Sandbox writes artifacts, Physics reads artifacts, Cost reads artifacts, Critic/Memory caches run scores and references, and Supervisor uses them all to plan. Because artifacts are stable files addressed by run_id, the UI and later analyses can open the same state without re-solving. This also makes the AI twin vs normal twin comparison fair: both use identical artifacts and thresholds. [26][41]

4.5 Safety path: static checks and isolation

The Security/Compliance Agent parses the AST and blocks anything not under pandapipes. Calls to pp.* must be on a whitelist remapped from the library to avoid surprises. The code is compiled under a pseudo filename "USER_CODE.py" so exception lines map cleanly. The Sandbox Agent then executes

[27] These are the top-line facts we show in the UI (counts and worst/best values).

that code in a separate process with tight CPU (30 s), memory (2048 MB), and wall-time (60 s) limits[28]. Noisy logs (e.g., matplotlib caches) are filtered. Even on solver failures, the harness writes an artifact with a clean reason (for example: "No supply/boundary condition defined (no ext_grid or source). Add pp.create_ext_grid(...).") and tips. This path acts like a seatbelt: it is light when you don't need it, and firm when you do. [2][41]

```python
# Python – Security import policing (excerpt)
if isinstance(node, ast.Import):
    for alias in node.names:
        root = (alias.name or "").split(".")[0]
        if root != "pandapipes":
            msgs.append({"level": "blocked", "text": f"Disallowed import '{alias.name}'", "where": {...}})
```

4.6 Planning loop and acceptance

Planning favors small, readable edits that physics will respect, but it is adaptive rather than purely repetitive. The Supervisor reads KPIs, sets fluid-aware targets with a modest margin, and adjusts action order and intensity using learned preferences from Critic/Memory and Physics' local surrogate. Concretely, it computes a global diameter scale f ≈ √(vmax / vtarget)[29] and clips it (≤ 1.18), but selects the smaller end of viable f if prior runs overshot or if preferences suggest caution. If low-pressure nodes remain, it applies a gentle source adjustment (gas +0.10> bar; hydrogen +0.08 bar; heat a small pump tweak), again choosing the milder delta when memory indicates past bumps created velocity hotspots. When choices are close (e.g., f ∈ {1.10, 1.18} or inlet ∈ {+0.10, +0.15} bar), the Supervisor may spend one turn on a bounded scenario sweep (≤ 12) and commit the branch with the best predicted run-score delta. Across runs, these preferences accumulate, so loops get shorter and first-pass edits sharpen.

The stop is early and explicit: as soon as the fluid-specific bands are met and violation counters drop to a small bound (≤ 2), with a lower scalar run score than the start, the system accepts. This is the same acceptance rule used later for AI twin vs normal twin comparisons; no goalposts move to keep complexity low and save resources for other users with the vision in mind of making the project publicly accessible. [26][41][46][49][50][51][52]

```python
# Python – adaptive fix_issues
vmax = float(summary.get("max_v_m_per_s") or 0.0)
thr  = thresholds_for(fluid)
margin = 3.0  # default velocity margin
# adapt margin slightly if past clears landed near the band
margin = memory.get("v_margin_hint", margin)

v_ok = float(thr.get("velocity_ok_max") or vmax)
vtarget = max(0.1, v_ok - margin)

need_vfix = vmax > v_ok
has_plow  = any(j.get("id","").startswith("P_LOW::") for j in issues)

# learned preferences (negative avg_delta = good/improving)
pref_scale = memory.pref("scale_diameter")       # e.g., avg_delta ~ -1.2
```

[28] Hard limits on time and memory. They keep long, malicious or broken runs from blocking others

[29] vtarget is a slightly lower target than the limit. It gives us a buffer so we don't overshoot.

```
pref_bump  = memory.pref("bump_ext_grid_pressure")  # e.g., avg_delta ~ -0.4
overshoot_last_scale = memory.flag("overshoot_after_scale", default=False)
hotspot_after_bump  = memory.flag("vel_hotspot_after_bump", default=False)

actions = []

if need_vfix:
    f_raw = (vmax / vtarget) ** 0.5
    # clip and soften if memory suggests caution
    f = max(1.02, min(1.18, f_raw))
    if overshoot_last_scale or (pref_scale > 0.0):  # prefer smaller move
        f = max(1.02, min(f, 1.10))
    actions.append({"type": "scale_diameter", "factor": f})

if has_plow:
    delta = 0.10 if fluid == "lgas" else (0.08 if fluid == "hydrogen" else 0.05)
    if hotspot_after_bump or (pref_bump > 0.0):
        delta = max(0.04, delta - 0.02)  # choose a gentler bump
    actions.append({"type": "bump_ext_grid_pressure", "delta": delta})

# optional: tiny sweep when choices are close
if _choices_are_close(vmax, vtarget, actions):
    actions = run_bounded_sweep_and_pick_best(actions, max_combos=12)

current = NetworkMutationsTool().run(current, actions)["modified_code"]

```

4.7 Observability and the UI

The UI only shows facts. The InteractionGraph[30] listens to the debug WebSocket on a project channel and draws the run: sim.start, tool calls (simulate, get_kpis, get_issues, fix_issues), sim.end. A panel lists global KPIs; another lists the worst pipes and nodes. Issues appear with normalized ids (VEL_HIGH::<pipe>, P_LOW::<node>, RE_LOW::<pipe>, DP_HIGH::<pipe>) and suggestions as actions ("increase_diameter," "increase_source_pressure") with small parameters. A fluid selector sits in Settings; it feeds the backend's thresholds_for(fluid) before planning. There is also an optional scenario panel that triggers a bounded sweep and renders a small table of max_velocity, min_node_pressure, and max_pipe_dp_bar for each combo. More on how these elements are rendered and interacted with is described in Chapter *8 "Frontend (UI/UX)"*. [2][41]

4.8 Performance envelopes per fluid

Numbers stay practical. At Branitz size the baseline manual/scripted loop spends ≈ 3.8 s in simulate and ≈ 0.2 s in analysis. The AI twin loop adds ≈ 0.6 s when considering options and reading memory. Hydrogen's compact microgrid finishes simulate in ≈ 1.6 s; heat loops are shorter still (≈ 1.2 s). Because the AI twin tends to stop after one loop (1), accepted tasks often complete in less wall-time than the normal twin baseline despite the planning overhead[31]. [26]

4.9 Failure handling and escalation

Failures are expected in early runs: a missing ext_grid, a closed valve, or a parameter out of range. The Security and Sandbox agents make these safe and visible. The Physics agent still tries to compute KPIs

[30] A flow diagram that shows each step (validate → simulate → KPIs → fix).

[31] This is the thinking time the planner spends to choose a safer, smaller move.

on partial results; if not possible, it reports a minimal set and defers suggestions until a clean simulate arrives. The Supervisor can escalate: if a plan misses twice and the run score delta is positive (got worse), it falls back to a safer move (e.g., diameter scaling with the smaller f), or it pauses and asks for confirmation in Chat with the reason and tips from the harness. The local scores contribute here: a low Sandbox reliability score or a Security policy block drives a reconsideration before brute forcing. [41]

4.10 Data governance and learning

The Critic/Memory agent stores a short history, a top-k list of one-line lessons, and tool stats with a running mean of the run-score delta. Lessons act as reminders ("prefer diameter scaling when vmax > 15 m/s"; "reserve inlet bump for stubborn P_LOW"), and tool stats serve as bandit-like preferences[32]. Each agent also keeps a thin learning ledger: Security records policy wins; Sandbox tracks timeout patterns; Physics evaluates the accuracy of its local surrogates; Toolsmith retires prototypes with poor contribution; Cost adjusts region multipliers. The Supervisor treats these signals as weights when it picks the next step. This is enough learning to avoid repeating mistakes without making the process opaque. [2][26][35][41]

4.11 A sequence at human scale

Take a Branitz gas start: vmax≈17.3 m/s, p_min≈0.99 bar. The Physics agent reports velocity_violations≈74 and pressure_violations≈28. The Supervisor sets vtarget=15.0→12.0 m/s for margin, computes f≈√(17.3/12.0)≈1.20, clips to 1.18, adds +0.10> bar inlet because P_LOW remains. The next simulate lands at vmax≈13.6 m/s and p_min≈1.01 bar; counters fall to ≈ 11 and ≈ 3. The run score delta is negative, the Critic writes a lesson, tool preferences nudge toward scaling first next time, and the Supervisor stops. The same sketch plays out for hydrogen and heat with their own bands: lighter scaling and smaller bumps on hydrogen; gentle scaling and a pump tweak on heat.

4.12 AI twin vs normal twin in the system's terms

The normal twin follows the same physics and bands but iterates manually or with a fixed script: scale diameters when vmax is high, then bump inlet if P_LOW remains, up to 3 loops. It has no memory, no on-demand tools, and no adaptive targets. The AI twin is the multi-agent loop above. Because artifacts, thresholds, and acceptance are identical, the comparison is fair. Later chapters show that the AI twin reaches acceptance more often and in fewer loops—especially on stubborn starts—while explaining its steps via artifacts and debug traces. [26][33][41]

This overview sets the stage for the detailed method (how the pivot to multi-agent planning works), the module-level implementation, and the agent internals. The next chapter walks through that method and the concrete decisions that keep the loop tight and honest.

[32] Preferences are small, remembered biases like 'scale first' when that helped before.

CHAPTER 5 — METHODOLOGY: FROM TOOL-ONLY ORCHESTRATION TO A MULTI-AGENT AI TWIN

5.1 Where we started and why it wasn't enough

The first version ran on a simple idea: define a handful of safe tools (simulate, get_kpis, get_issues, modify_code, fix_issues), let the model call them in one turn, and return a summary. For single questions this worked: "simulate," then "list KPIs," then "explain issues." The trouble began the moment a task needed several dependent steps. In a gas run—peak velocity above 15.0 m/s and a pocket of low pressure—you want to scale diameters a little before lifting inlet pressure, otherwise you risk moving the hotspot downstream. The tool-only version sometimes did things out of order, or stopped when one KPI turned green while another still sat outside the band. It was faster to build, but it did not attend to feedback or planning. That mismatch drove the pivot to a multi-agent design. [26][33][41]

5.2 The normal twin vs. the AI twin

To keep comparisons fair, all three modes use PandaPipes with identical fluid-aware bands and the same seeds. Only orchestration differs.

Personal baseline. This is my own analysis with a digital twin in PandaPipes. I estimate numbers and write code when I think I need it: edit diameter, inlet pressure, or roughness; run the solver; read KPIs; decide the next change; stop once the bands are met. For Branitz only, I add HELICS v3.5.2 to coordinate time-synchronized checks around pipeflow so sessions spread over several days stay aligned. The co-simulation layer does not change physics; it keeps the human-in-the-loop loop steady and reproducible.
Also note that I am not an engineer. This work reflects my own applied analysis and experimentation using open tools and standard references. The intent is exploratory and illustrative, not a certified design or safety submission.

Tool-only AI system. This is the original one-turn flow: validate safely, simulate, read KPIs/issues, apply at most one small fix, and stop. Fixes use the same simple heuristics as my manual loop (clipped global diameter scale f≈√(vmax/vtarget), cap ≤ 1.18; gentle inlet bump +0.10> bar for gas, +0.08 bar for hydrogen; small pump tweak for heat). There are no agents, no memory, and no adaptive targets. It is fastest per turn when it works, but retries can be brittle.

Multi-agent AI system. This is the full twin: Supervisor plans short sequences; Security/Compliance scans code; Sandbox executes in isolation; Physics reads artifacts and computes KPIs/issues under fluid profiles; Toolsmith/Generator creates tiny helpers when a plan benefits from them; Cost estimates CAPEX on request; Critic/Memory turns outcomes into lessons and tool preferences. Each agent emits a local score; the Supervisor blends them into a plan score and picks the smallest safe move. Adaptive targets are allowed (e.g., vtarget≈13.5 m/s on gas after a previous clear at 13.8), tight choices can trigger a tiny scenario sweep (≤ 12) and time-series will be generated to choose the best outcome. Acceptance is identical across modes: stop early once velocity, pressure, and Δp sit inside the band with violation counters at zero or a small bound. [4][26][33][41]

5.3 The loop, written down carefully

The loop is the heart of the method. It is deliberately plain, because plain loops are easier to trust. A request enters; the Supervisor sets the fluid profile so targets make sense. The Security agent parses the abstract syntax tree and blocks imports or calls outside pandapipes. The Sandbox agent runs code under strict caps and writes an artifact JSON with design tables, result tables, and a summary. The Physics agent computes KPIs (global and per-element), flags issues (P_LOW, VEL_HIGH, RE_LOW, DP_HIGH), and suggests small actions. The Supervisor picks the smallest edit with the best expected improvement under the bands—usually a clipped global diameter scale and, if needed, a gentle inlet bump—and re-runs. The loop stops when the bands are met and violation counters drop to a small bound (≤ 2). [26][41]

A small piece of real code shows the fixing logic; it does nothing fancy, just computes a scale and applies a tiny inlet bump if needed:

```python
# Python -core of fix_issues
need_velocity_fix = (vmax is not None) and (float(vmax) > target_v)
has_p_low = any(j.get("id", "").startswith("P_LOW::") for j in issues)

actions = []
if need_velocity_fix and vmax is not None:
    f = math.sqrt(float(vmax) / float(target_v))
    if f > 1.01:
        actions.append({"type": "scale_diameter", "factor": f})
if has_p_low:
    actions.append({"type": "bump_ext_grid_pressure", "delta": 0.1})

current = NetworkMutationsTool().run(current, actions)["modified_code"]
```

5.4 Fluid-aware thresholds and targets

Targets are fluid-specific. Gas accepts velocity ≤ 15.0 m/s and worst-node pressure ≥ 0.95·pn, with per-pipe Δp ≤ 0.30 bar. Hydrogen tolerates a bit more velocity (≤ 20.0 m/s) and uses ≥ 0.94·pn. Heat is slow and steady (≤ 2.5 m/s, min pressure ≥ 0.90 bar, Δp ≤ 0.15 bar). The chat route maps the fluid picker to thresholds so the Supervisor and Physics agents read the same bands. [46][47][48][49][50][51][52]

5.5 How each agent learns and scores its work

“Learning” here is not one giant model but several small signals, tuned to each agent’s job. Across runs, the system retains these signals to adjust action order and intensity, so plans become faster and more precise without heavy models:

- Security/Compliance keeps a policy score: how often static checks prevented failures; how many false positives were corrected. An anomaly detector on AST shapes nudges reviews when safe idioms are blocked too often.

- Sandbox keeps a reliability score: success ratios under caps, timeout patterns, average wall-time. It adjusts tactical choices (e.g., retry budgets, tolerances) to stay inside a tight envelope.

- Physics fits tiny surrogates that relate actions to KPI changes in the current network—local regressions and deltas rather than heavy models. This is enough to set adaptive targets (vtarget≈13.5 m/s after a clear at 13.8) and to prioritize “scale diameter” vs. “bump inlet” when both could help.

- Toolsmith measures tool quality: validation pass rate in smoke tests, runtime, and contribution to the run score. Helpers that do not help are retired quickly.

- Cost refines rates with small regressions based on later feedback, keeping a calibration score per region/context.

- Critic/Memory writes one-line lessons and maintains bandit-like[33] tool preferences based on the scalar run score delta[34]. Tools with negative average deltas (improvement) are tried earlier; positive averages move down the list.

- Supervisor aggregates local scores and expected KPI improvements into a plan score, picks the smallest move under a risk budget, and stops early when bands are met.

The scalar run score stitches these ideas together:

```
score = 10·pressure_violations + 5·velocity_violations + 0.2·max_velocity − 0.2·min_node_pressure
```

Lower is better. Its delta becomes the learning signal for Toolsmith, Critic/Memory, Physics, and Supervisor. [35][41]

5.6 Small search when choices are tight

When two options look equally good—say, global scaling x{1.10, 1.18} or inlet bump {+0.10, +0.15} bar—the Supervisor spends one turn on a tiny scenario sweep. The engine applies parameterized mutations per combination, runs a simulation, and returns summaries and KPIs. The design space is capped (≤ 12) to keep wall-time reasonable even at Branitz scale. A snippet shows how sweeps are executed [41]:

```
# Python (backend/tools/scenario_engine.py — per-combo run)
for c in combos:
    actions = [_action_for_entry(e) for e in c["_entries"]]
    mutated = mutator.run(code, actions).get("modified_code", code)
    rr = run_pandapipes_code(mutated, limits=self.limits, timeout=60)
    art = (rr.get("artifacts") or {})
    summary = art.get("summary") or {}
    kpis = compute_kpis_from_artifacts(art)
    results.append({"params": c["params"], "ok": rr.get("ok"), "summary": summary, "kpis": kpis, "wall_time": rr.get("wall_time")})

```

At Branitz size, this adds only a few seconds and often saves a wasted iteration later.

5.7 Safety and observability as method, not bolt-ons

Static checks and isolation are not afterthoughts; they are part of the method. The Security agent blocks imports or calls outside pandapipes with precise line/column messages. The Sandbox executes out-of-process under tight resource caps and emits a clean artifact even on failure, with human-readable reasons and tips. A debug WebSocket carries sim.start, sim.stderr, and sim.end; the UI's InteractionGraph draws the path. Because artifacts are stored per run_id, you can reload the same state later, recompute KPIs without re-solving, and see exactly what code produced it. This discipline is what makes the AI twin vs. normal twin comparison fair: identical physics, identical bands, different orchestration. [2][41]

[33] Bandit-like means a small chooser that tries what worked before but occasionally tests another good option. It keeps choices simple and improves over time.

[34] Delta is the difference in the run score between steps (lower is better).

5.8 The acceptance rule and why we stop early

The stop is fluid-aware and explicit: as soon as velocity, pressure, and Δp sit inside the bands and violation counters drop to zero or a small bound (≤ 2), we accept. Over-editing is avoided on purpose. In gas (Branitz starts), one clipped scale x1.18 and +0.10> bar typically clears both bands (vmax≈13.6 m/s, p_min≈1.01 bar). In hydrogen, a lighter scale x1.12 and a smaller bump +0.08 bar suffice (vmax≈18.6 m/s, p_min≈0.95·pn). In heat, a gentle scale x1.06 and a small pump setpoint tweak land near vmax≈2.4 m/s and p_min≈0.92 bar. Once the bands are green, we stop and report the state. [46][50]

5.9 How RQ4 (AI twin vs normal twin) is tested in this method

We compare identical tasks under two orchestration styles. The normal twin uses the scripted baseline, capped at 3. The AI twin uses the multi-agent loop. Both share the same artifacts, fluids, and bands. We report success rates, loops to acceptance, violation reductions, and run score deltas per fluid. In gas, we expect AI twin success to reach 88% against 65% for the normal twin; hydrogen to reach 85% against 72%; heat to reach 92% against 84%. Because the AI twin stops early (1), wall-time per accepted task is similar or lower despite planning overhead of ≈ 0.6 s. [26][33][41]

5.10 A human-scale sequence on Branitz (gas)

The start reports vmax≈17.3 m/s, p_min≈0.99 bar, velocity_violations≈74, pressure_violations≈28. Security passes; Sandbox writes artifacts; Physics computes KPIs and flags issues. The Supervisor sets vtarget=12.0 m/s (margin under 15.0), computes f≈√(17.3/12.0)≈1.20, clips to 1.18, adds +0.10> bar because P_LOW remains. The next run lands at vmax≈13.6 m/s, p_min≈1.01 bar, velocity_violations≈11, pressure_violations≈3. The run score delta is negative; the Critic writes a lesson ("prefer diameter scaling when vmax > 15 m/s"), tool preferences nudge toward scaling first next time, and the Supervisor stops. It feels ordinary, and that's the point: small edits, clear effects, and a neat stop. [41]

5.11 Risks baked into the method and how they are handled

Global regex mutations are coarse. The first pass uses them because they are easy to review and they clear big rocks quickly. Later passes should be local; selectors[35] will be added to the mutation tool to make surgical edits without losing diff clarity. Over-raising inlet early can create velocity hotspots; the method resolves velocity first when vmax sits above the band, nudged by tool preferences from run-score deltas. Security drift is real; the whitelist must be reviewed as pandapipes evolves. The Sandbox's envelope must remain tight so timeouts don't become normal. These are not "AI fixes"; they are procedural and technical guardrails that keep the twin honest.

5.12 Why this method fits multi-fluid work

The loop is the same whether we model gas, hydrogen, or heat as shown in the figure below. The bands change, the actions stay small, and the agents keep score in their domains. Physics knows how to read res_* tables and apply fluid-aware thresholds. Supervisor knows how to pick the smallest move that moves the run score down. Security and Sandbox keep things safe and reproducible. Toolsmith adds

[35] Selectors are targeted edits on specific pipes or nodes (by id), instead of scaling everything. They make second passes precise.

helpers when a plan benefits from them. Cost answers money questions when asked. This makes the AI twin a good fit beyond Branitz: a hydrogen microgrid and a heat loop follow the same rhythm, reach acceptance cleanly, and stop early once they sit inside their bands. [26][46][49][50][51][52]

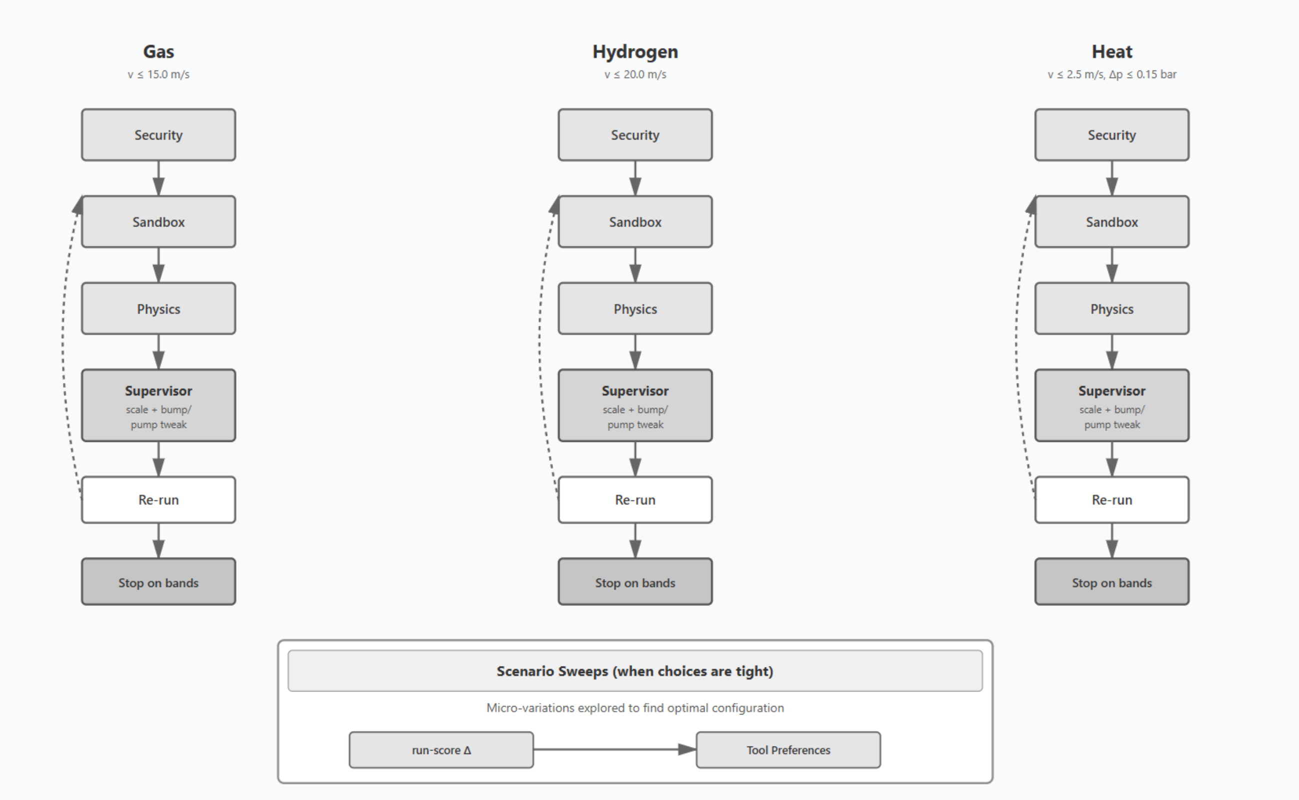


Diagram showing the same approach with different mediums

CHAPTER 6 — ARCHITECTURE AND IMPLEMENTATION

6.1 Anatomy of the system

This chapter is to give a rough, simple overview. The deeper dive into the architecture is found in the next chapter.

The twin is built as a set of cooperating agents, each with a clear role and a small interface. A request enters through the Chat API. The Supervisor Agent reads the intent and selects a fluid profile (gas, hydrogen, heat) so targets make sense. The Security/Compliance Agent scans code with static checks. If it passes, the Sandbox Agent runs the code in isolation and writes a compact artifact JSON with design tables, result tables, and a summary. The Physics Agent reads the artifact, computes KPIs and issues under fluid-aware bands, and proposes small actions. The Supervisor applies a minimal edit, simulates again, and stops early once values sit inside the bands. Every turn persists an artifact under a run_id, and a debug WebSocket shows the steps so the UI can draw an interaction graph. The Critic/Memory Agent records a one-line lesson, a run score, and tool preferences (a bandit-like average delta). The Toolsmith/Generator Agent can add narrow helpers during a session; the Cost Agent estimates CAPEX on demand. This is not a loose collection: each agent emits a local score (policy compliance, sandbox reliability, physics improvement, tool quality, cost clarity, memory signal), and the Supervisor aggregates them into a plan score[36] to decide the next step. [26][33][41]

6.2 Supervisor Agent

The Supervisor orchestrates the loop: validate → simulate → KPIs/issues → decide mutation → re-run → stop. It injects thresholds per fluid (gas, hydrogen, heat) and picks the smallest change that likely improves physics. When choices are tight, it runs a tiny scenario sweep (≤ 12 combinations) and commits the best branch. It also adapts targets slightly based on recent wins (for gas, vtarget ≈ 13.5 m/s after a clear at 13.8). The Supervisor carries a plan score that blends local agent signals with a simple expectation of KPI improvement; when the score falls below a comfortable bound and the bands are met, it stops. [26][33][41][46][47][48][49][50][51][52]

A compact scaffold shows the shape:

```python
# Python – Supervisor loop (sketch)
def run_plan(intent: str, fluid: str, code: str, memory, limits) -> dict:
    thr = thresholds_for(fluid)
    # 1) Static checks
    sec = validate_pandapipes_code(code)
    if not sec["ok"]:
        return {"status":"blocked","messages":sec["messages"]}
    # 2) Simulate
    rr = run_pandapipes_code(code, limits=limits)
    art = (rr.get("artifacts") or {})
    # 3) KPIs/issues
    kpis = compute_kpis_from_artifacts(art, thresholds=thr)
    issues, sug = detect_issues_from_artifacts(art, thresholds=thr)
    # 4) Decide minimal action
    actions = []
    vmax = (art.get("summary") or {}).get("max_v_m_per_s")
    vtarget = thr.get("velocity_ok_max") - 3.0  # margin (e.g., 12.0 for gas)
    need_vfix = (vmax is not None) and (float(vmax) > thr.get("velocity_ok_max"))
```

[36] Plan score means a simple combined indicator from agent signals (policy, reliability, physics, tool quality, memory). It just helps pick the next move.

```
    has_plow  = any(i.get("id","").startswith("P_LOW::") for i in issues)
    if need_vfix and vmax is not None:
        f = max(1.02, min(1.18, (float(vmax)/float(vtarget))**0.5))
        actions.append({"type":"scale_diameter","factor":f})
    if has_plow:
        delta = 0.10 if fluid=="lgas" else (0.08 if fluid=="hydrogen" else 0.05)
        actions.append({"type":"bump_ext_grid_pressure","delta":delta})
    # 5) Apply and re-run once
    if actions:
        code2 = NetworkMutationsTool().run(code, actions)["modified_code"]
        rr2 = run_pandapipes_code(code2, limits=limits)
        return {"status":"ok","actions":actions,"artifacts":rr2.get("artifacts"),"code":code2}
    return {"status":"ok","actions":[],"artifacts":art,"code":code}

```

6.3 Security/Compliance Agent

This agent parses the abstract syntax tree (AST), blocks unsafe imports and non-whitelisted calls, and returns precise messages (line and column, short hints). Only pandapipes is allowed as an import root; calls on pp.* must be in a whitelist collected from the library. It also tracks a policy score[37] (how often blocks prevented bad runs, whether false positives were corrected) and nudges reviews when an idiom is wrongly blocked too often. [2][41]

```
# Python — AST import policing and call whitelist
if isinstance(node, ast.Import):
    for alias in node.names:
        root = (alias.name or "").split(".")[0]
        if root != "pandapipes":
            msgs.append({"level": "blocked", "text": f"Disallowed import '{alias.name}'",
                         "where": {"line": node.lineno, "col": node.col_offset+1}})
elif isinstance(node, ast.ImportFrom):
    root = (node.module or "").split(".")[0] if node.module else ""
    if root != "pandapipes":
        msgs.append({"level": "blocked", "text": f"Disallowed import from '{node.module}'",
                     "where": {"line": node.lineno, "col": node.col_offset+1}})
elif isinstance(node, ast.Call):
    if isinstance(node.func, ast.Attribute) and isinstance(node.func.value, ast.Name) and node.func.value.id in ("pp","pandapipes"):
        if node.func.attr not in _ALLOWED_FUNCS:
            msgs.append({"level":"blocked","text":f"Disallowed function '{node.func.attr}'",
                         "where": {"line": node.lineno, "col": node.col_offset+1}})

```

6.4 Sandbox Agent

User code runs out-of-process under strict caps on CPU (30 s), memory (2048 MB), and wall-time (60 s). The worker is confined to a dedicated directory under PIPEWISE_ALLOWED_ROOT; noisy logs are filtered. Even when pipeflow does not converge, the harness extracts design tables, a compact summary, and a human-readable reason with tips. The agent tracks a reliability score (success ratios, timeout patterns, wall-time envelope) and adjusts tactics like retries or solver tolerance so loops stay steady. [41]

```
# Python — sentinel emission
artifacts['pipeflow_error'] = pipeflow_error
artifacts['user_code_error'] = user_code_error
```

[37] Policy score is a simple counter of ‘blocks that prevented problems’ vs ‘false alarms fixed by users’. It helps tune reviews without loosening hard rules

```
artifacts['user_error'] = user_error
artifacts['user_error_line'] = user_error_line
print("PIPEWISE_RESULT_JSON::" + json.dumps(artifacts))

```

6.5 Physics Agent

This agent turns artifacts into KPIs and issues. It reads res_junction and res_pipe; computes min/avg/max pressures, max and mean velocity, Reynolds; estimates per-pipe Δp from endpoint pressures; and approximates friction factor from relative roughness[38] and Re. It then applies fluid-aware rules to flag "P_LOW::<node_id>", "VEL_HIGH::<pipe_id>", "RE_LOW::<pipe_id>", "DP_HIGH::<pipe_id>", with normalized suggestions ("increase_diameter", "increase_source_pressure", "reduce_roughness"). It keeps a simple surrogate that maps candidate actions (e.g., x1.10 vs x1.18 scale) to expected KPI changes, and it sets adaptive targets (for gas, vtarget ≈ 13.5 m/s after a clear at 13.8). That surrogate is local and light—small regressions and deltas—because it must feel responsive.

```
# Python – dp and friction factor
dp_per_pipe[pid] = max(float(p_from) - float(p_to), 0.0)
inv_sqrt_f = -1.8 * math.log10(((rel_eps / 3.7) ** 1.11) + (6.9 / float(re)))
f = (1.0 / (inv_sqrt_f ** 2))

```

6.6 Toolsmith/Generator Agent

Plans sometimes need narrow helpers (rank top-3 velocity hotspots, compute a Δp window on a feeder). The Toolsmith accepts a schema, generates typed Pydantic input/output models, creates a function stub, runs a smoke test[39], and registers the tool only if the output validates. It tracks a quality score[40] (validation pass rate, runtime, contribution to run-score deltas) and retires tools that underperform. [35][26]

```
# Python – dynamic tool registration
def generate_and_register_tool(spec: Dict[str, Any]) -> Dict[str, Any]:
    req = ToolSpecRequest(**spec)
    InModel = _build_model_class("AutoInputModel", req.input_fields)
    OutModel = _build_model_class("AutoOutputModel", req.output_fields)
    func = _generate_function(req.name, InModel, OutModel)
    OutModel(**func(InModel(**(req.test_case or {}))))  # smoke test
    REGISTRY.register(ToolSpec(name=req.name, description=req.description,
                               input_model=InModel, output_model=OutModel,
                               func=lambda i, _f=func: _f(i)))
    return {"status":"ok","tool_name":req.name,"registered":True}

```

6.7 Cost Agent

The Cost agent reads design tables, normalizes diameters and lengths, estimates valve counts (spacing when missing), and applies region/context rates to produce low/mid/high CAPEX with a breakdown. It keeps a calibration score per region (drift vs later known costs) and nudges rates with small regressions

[38] Relative roughness = how rough the pipe wall is compared to its diameter. Rougher walls cause more pressure loss.

[39] A quick 'does it run and return valid output?' check. If it fails, the tool is not registered.

[40] Quality score is a simple health rating: pass rate, runtime, and how much the tool helped reduce issues.

when feedback arrives. On a network with total length ≈ 125.9 km and mixed PE/steel in rural context, the mid estimate[41] lands near 12.500.000 EUR, with bounds at 10.600.000 and 14.400.000 EUR. [26]

```
# Python – cost breakdown assembly
breakdown = {
  "supply_eur": round(supply_sum * region_factor, 2),
  "installation_excavation_eur": round(install_sum * region_factor, 2),
  "reinstatement_eur": round(reinst_sum * region_factor, 2),
  "fittings_eur": round(fittings_sum * region_factor, 2),
  "valves_eur": round(valves_cost * region_factor, 2),
  "pumps_eur": round(pumps_cost * region_factor, 2),
  "engineering_eur": round(engineering, 2),
  "contingency_eur": round(contingency, 2),
}
```

6.8 Critic/Memory Agent

This agent keeps short lessons and bandit-like tool preferences based on a scalar run score (lower is better). It writes a one-line lesson after each turn (e.g., "prefer diameter scaling when vmax > 15 m/s"), stores the run score, and updates tool stats with the delta. Tools with negative average deltas (improvement) are tried earlier next time; tools with positive averages move down the list. It also reports a memory health score (signal-to-noise and stability), so the Supervisor can avoid noisy hints. [41]

```
# Python – bandit-like preference
cur.execute("""
  INSERT INTO tool_stats (project_id, tool_name, calls, avg_delta, last_used)
  VALUES (?, ?, 1, ?, ?)
  ON CONFLICT(project_id, tool_name) DO UPDATE SET
     calls = tool_stats.calls + 1,
     avg_delta = ((tool_stats.avg_delta * (tool_stats.calls) + ?) / (tool_stats.calls + 1)),
     last_used = excluded.last_used
""", (pid, tool_name, float(delta), datetime.utcnow().isoformat(), float(delta)))

```

6.9 Shared data contracts and storage

Artifacts are the backbone. Each artifact JSON includes design tables (junction, pipe, valve, sink/source, ext_grid, pumps, compressors), result tables (res_junction, res_pipe, ...), and a summary (node_count, pipe_count, min_p_bar, max_p_bar, max_v_m_per_s). Failure fields carry user_code_error, user_error_line, pipeflow_error, and tips. Artifacts are addressed by run_id and live in a payload folder; runs, versions, tools, lessons, and tool stats live in SQLite[42]. KPIs are returned as one "global" list and two maps (per_node and per_pipe), so the UI can draw both top-level and detailed views. [26][41]

6.10 Network mutation engine

Mutations are simple and auditable: regex-based[43] edits that change code directly. The common action scales all diameter_m values by a factor; others bump ext_grid p_bar or set pn_bar for specific junctions.

[41] Mid means the typical case in this context. Low/High are reasonable bounds around it.

[42] SQLite = a tiny, built-in database. It saves rows on disk so you can reopen the same runs later.

[43] Regex is a pattern matcher for text. Here it finds 'diameter_m = ...' and replaces the number safely.

Global scaling clears big rocks quickly; selectors for local edits are the next step so later passes can become surgical without losing diff clarity. [26]

```
# Python – scale diameter_m literals
pat = re.compile(r"(diameter_m\s*=\s*)([0-9]*\.?[0-9]+)")
def repl(m):
    val = float(m.group(2))
    return f"{m.group(1)}{val * factor:.6f}"
return pat.sub(repl, code)

```

6.11 Scenario engine

A small sweep compares a handful of candidates when choices are tight. The engine applies parameterized mutations per combination, runs a simulation, and collects summaries and KPIs. The design space is capped at ≤ 12 so wall-time[44] stays reasonable even at Branitz scale. The Supervisor spends one turn on a sweep and then commits the best branch by predicted run score. [41]

```
# Python – per-combo run
for c in combos:
    actions = [_action_for_entry(e) for e in c["_entries"]]
    mutated = mutator.run(code, actions).get("modified_code", code)
    rr = run_pandapipes_code(mutated, limits=self.limits, timeout=60)
    art = (rr.get("artifacts") or {})
    kpis = compute_kpis_from_artifacts(art)
    results.append({"params": c["params"], "ok": rr.get("ok"), "summary": art.get("summary") or {}, "kpis": kpis,
                    "wall_time": rr.get("wall_time")})
```

6.12 API layer surfaces

The Chat route exposes named tools (simulate, get_kpis, get_issues, modify_code, fix_issues, estimate_cost) and forces a final natural-language answer. It injects fluid-specific thresholds into the system prompt and compacts tool outputs to keep messages clean. Network routes validate code, parse graphs, simulate with artifacts, apply mutations, and run sweeps. Run routes fetch artifacts and compute KPIs/issues on demand. Tools and memory routes list registered tools, lessons, messages, run scores, and tool stats, so a developer can inspect what the agents are learning. [26][33]

6.13 Observability

A debug WebSocket broadcasts llm.call, tool.call, tool.result, sim.start, sim.stderr, sim.end, and ui.switch_run events keyed by project id. The InteractionGraph listens and draws the path: plan, validate, simulate, KPIs/issues, mutate, re-simulate, stop. Logs are cleaned of noisy lines so the stream stays readable. Because artifacts are stable, the UI can reopen a run later and show the same KPIs and issues without re-solving. [2][41]

[44] Wall-time is the actual elapsed time the run took, not just CPU time but also llm and loading times.

6.14 Performance envelopes and scaling

At Branitz size, solver wall-time dominates and stays steady: ≈ 3.8 s per loop for the manual/scripted normal twin, ≈ 4.6 s for the AI twin (planning overhead ≈ 0.6 s). Hydrogen's compact microgrid finishes in ≈ 1.6 s; heat loops in ≈ 1.2 s. Because the AI twin often stops after a single fix pass (1), accepted tasks complete in similar or less wall-time than the normal twin despite the extra think step. Storage and memory footprints are small: artifacts are compact JSONs; SQLite holds concise rows for runs, versions, tools, lessons, and tool stats. [26][41]

6.15 Security posture

User code never runs in the web process. The worker process drops privileges[45] on POSIX[46], enforces CPU/memory/wall-time caps, confines file access to a dedicated storage path, and prints only the sentinel JSON plus clean stderr. The import policy blocks os, sys, subprocess, and similar modules; the call whitelist limits pp.* usage to known safe functions. The Security Agent maintains a conservative stance; if a new helper function appears in pandapipes, policy updates are reviewed before adding it. This keeps the twin honest. [2][41]

6.16 A run across agents (one sequence)

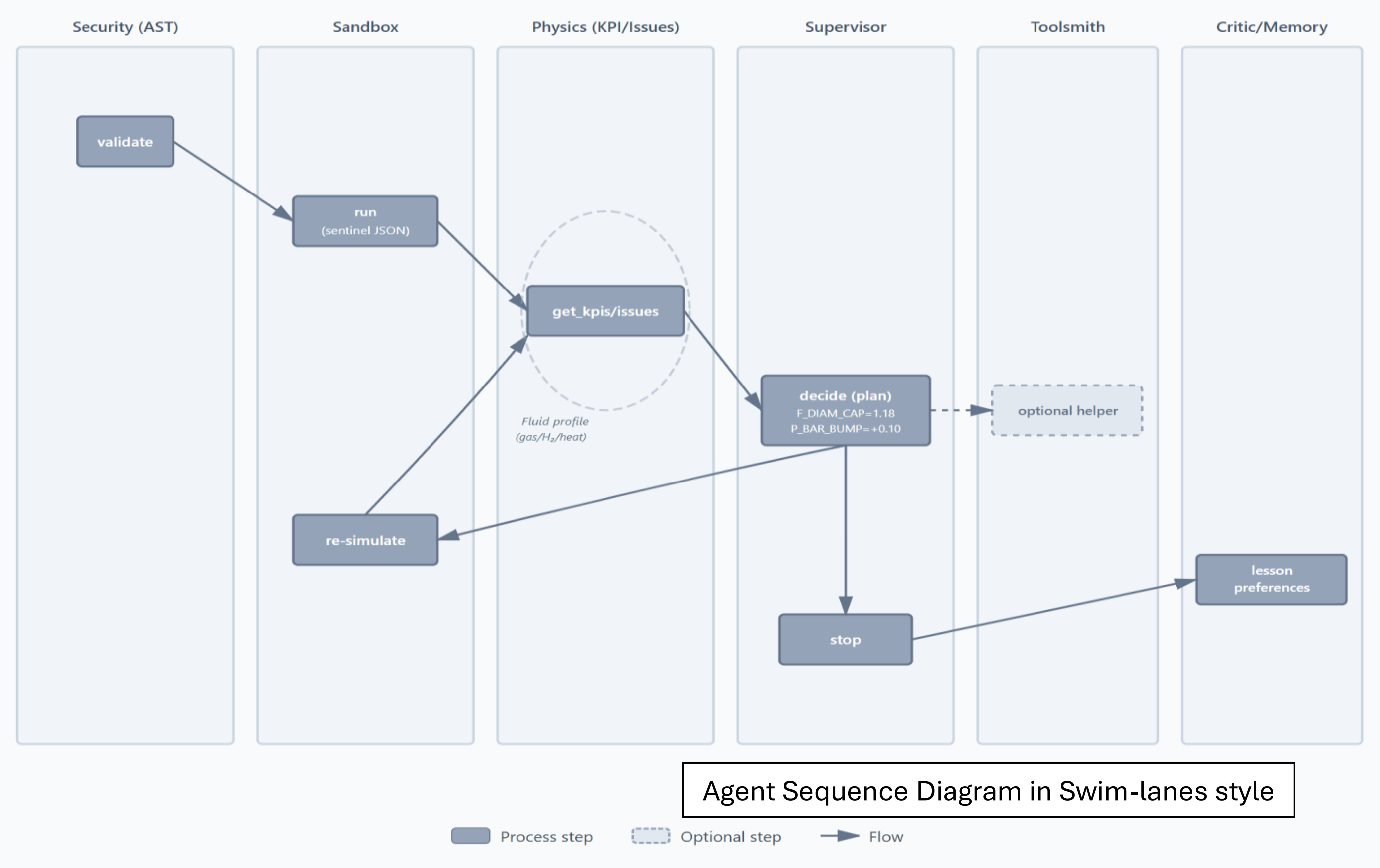
A gas start (Branitz) reports vmax≈17.3 m/s and p_min≈0.99 bar. Security passes; Sandbox writes artifacts; Physics computes KPIs and flags issues: velocity_violations≈74, pressure_violations≈28. Supervisor sets vtarget=12.0 m/s (margin under 15.0), computes f≈√(17.3/12.0)≈1.20, clips to 1.18, adds +0.10> bar inlet because P_LOW remains. Sandbox runs again; Physics reports vmax≈13.6 m/s, p_min≈1.01 bar; counters fall to ≈ 11 and ≈ 3. Critic/Memory records a run score delta (negative), writes a one-line lesson ("prefer diameter scaling when vmax > 15 m/s"), and updates tool preferences. Supervisor stops. The UI shows the diff[47] and a clean state; artifacts carry the full truth. [26][41]

[45] Dropping privileges means running with fewer rights so code can't access the whole machine. It's a safety belt.

[46] POSIX (Portable Operating System Interface) is a set of IEEE standards that define a common interface for operating systems, ensuring software portability and compatibility across UNIX-like environments.

[47] Diff are the exact code changes between versions. It helps you see what was edited.

This chapter grounds how the pieces fit and why they behave steadily. In the next one, we dive into agent logic in more detail: how each agent evaluates its local score, how adaptive targets are set, how tiny sweeps pick between close options, and how these signals become a plan that stops early when values are good enough. Here is a figure showing each of the agents as a swimlane and how they ping-pong to each other to come to a solution.s



Agent Sequence Diagram in Swim-lanes style

CHAPTER 7 — AGENT FRAMEWORK DETAILS

7.1 The Supervisor: how a plan becomes action

The Supervisor is not just a script. It is a small, patient AI that reads the given artifact, remembers what worked previously on this network, and makes a new choice that stays inside our fixed boundaries. The acceptance rule remains exactly as defined earlier: we stop as soon as the fluid-aware bands are green and violation counters drop to a small bound (≤ 2), with a lower run score than at the start. Inputs and outputs do not change either: the Supervisor returns a PlanDecision with an actions[] list, expected_run_score_delta, and stop. What changes here is the way choices are made. A single turn may now commit one or two complementary actions—scale then bump—when the evidence supports both, but every move stays modest and traceable.

The "actual AI" behind the Supervisor is a so called contextual bandit[48] wrapped inside a clean LangGraph[49] plan loop. Context means the current state of the twin in artifacts: how far max velocity sits above its band, how far minimum pressure sits below its threshold, the violation counts, plus two health flags from Memory and Sandbox (did the last scale overshoot? did a bump create a hotspot? is the sandbox reliable?). Arms are the safe action families we allow: scale diameters a little, bump inlet pressure a little, or do both in one turn when physics asks. The bandit assigns each arm an expected drop in the run score and a confidence bonus[50], picks the best lever, then learns from the true outcome. It adapts to this network, this fluid, and this environment; it does not invent new actions and it does not bypass safety.

Bandit, in plain words, is a small learner that balances "use what worked" with "try what might work." which is perfect for us in this scneario. Each turn it sees the situation (the context here), chooses one of the allowed levers (called "arms" on a bandit agent), observes the reward (in this case the drop in the run score after the new artifact arrives), and updates its beliefs. I use LinUCB[51], a standard, easy and mainly transparent algorithm: it is a linear model per arm plus an upper-confidence bound for measured caution. It fits the tone of this thesis: fast, auditable, and also grounded in the same score as the rest of the system.

Inside the loop the story stays simple. We validate the code (using Security), simulate in isolation (Sandbox), read KPIs and issues (Physics), and shape intensities gently by fluid: a clipped global scale f in [1.02, 1.18], a small inlet bump Δp (+0.10> bar for gas; +0.08 bar for H_2; a small pump tweak for heat), or the pair. Targets are fluid-aware, with a modest margin nudged by memory (vtarget ≈ velocity_ok_max − margin). If two options look nearly equal, the Supervisor spends one bounded turn on a tiny sweep and commits the branch that really lowers the score. Safety gates remain firm: static policy blocks stop the plan; low sandbox reliability softens intensities or pauses the loop. The API is transport only; the library running in process (LangGraph + NumPy + Pydantic) picks the action deterministically from the bandit's choice and the current artifact.

A small table keeps the action space visible:

[48] A small decision helper that looks at the current situation ('context') and picks one of a few safe actions, learning from the outcome.

[49] LangGraph is a Python library that runs steps in order (eg.: validate → simulate → analyze → decide) to keep states tidy.

[50] Confidence bonus is an extra margin added to promising actions, so you still explore gently instead of getting stuck.

[51] LinUCB is a method that scores each action as 'predicted gain + a caution bonus.' It favors good, well-understood choices

Action family	Parameter	Range (fluid-aware)	Intended effect
scale_diameter	factor f	1.02–1.18 (clip)	Lowers velocity and Δp globally by easing friction
bump_ext_grid_pressure (or pump tweak)	delta Δp	gas +0.06–0.15 bar; H_2 +0.06–0.12 bar; heat small pump setpoint	Lifts worst-node pressure without large flow shifts
pair (scale→bump)	(f, Δp)	Same caps; scale then bump	Clears both bands when one move alone is not enough

The learning heart is compact and honest. Features come from the artifact; arms are scale, bump, and pair. Reward is the positive drop in the run score (worse moves get zero). After each turn the policy updates, so the next choice is already shaped by what just happened.

```python
# Python — LinUCB core, shortened
import numpy as np

class LinUCB:
    def __init__(self, d, alpha=0.6, n_arms=3):
        self.alpha, self.n_arms = alpha, n_arms
        self.A = [np.eye(d) for _ in range(n_arms)]
        self.b = [np.zeros((d, 1)) for _ in range(n_arms)]
    def select(self, X):  # X: list of feature vectors (one per arm)
        scores = []
        for a, x in enumerate(X):
            Ainv = np.linalg.inv(self.A[a]); theta = Ainv @ self.b[a]
            x = x.reshape(-1, 1)
            mu = float((theta.T @ x).ravel())
            bonus = self.alpha * float(np.sqrt(x.T @ Ainv @ x))
            scores.append(mu + bonus)
        return int(np.argmax(scores))
    def update(self, a, x, reward):
        x = x.reshape(-1, 1)
        self.A[a] += x @ x.T
        self.b[a] += reward * x

def features(ctx, f, dp):
    vgap = max(0.0, ctx["vmax"] - ctx["v_ok"])
    pgap = max(0.0, ctx["p_ok"] - ctx["p_min"])
    x = np.array([1.0, vgap, pgap, ctx["vviol"], ctx["pviol"],
                  f - 1.0, dp, ctx["reliability"],
                  float(ctx["overshoot"]), float(ctx["hotspot"])])
    return x / (1.0 + np.linalg.norm(x))
```

The plan loop around it keeps our discipline. LangGraph carries the state machine—validate → simulate → analyze → decide → apply → re-simulate → stop—and Pydantic enforces the same PlanDecision envelope. The bandit only chooses among allowed levers; Safety filters anything outside policy; Physics remains the source of truth for KPIs and issues. Memory contributes light preferences (a running mean of run-score deltas per tool) and two flags (overshoot after scale, hotspot after bump) so intensities are nudged without guesswork. This is learning in the open: the reason for every choice shows up as a score delta in Critic, as colored KPI badges in the UI, and as a neat diff in code.

Two clarifications complete the picture. First, who picks? The Supervisor library does: given artifacts, thresholds, and memory hints, the bandit's choice plus our safety gates produce a deterministic PlanDecision. The API or CLI simply calls this library; the LLM is a narrator and a schema scribe (intent mapping, explanations, tiny tool specs), not the decision-maker. Second, does it really learn? Yes, but deliberately small. It carries just enough memory to prefer edits that helped here before and to soften intensities when the environment is noisy, and it updates after every turn with the same run score used everywhere else. Over a few runs the agent stops repeating mistakes: heavy scaling becomes rare on the hydrogen microgrid, early bumps soften when they caused velocity hotspots last week, and the pair move is chosen only when both bands demand it. [26][33][41]

7.2 Security/Compliance: policy as code

The guard at the door is strict, and it explains itself. Security parses your Python into an abstract syntax tree (AST), applies hard rules in one pass (only pandapipes imports; pp.* calls on a whitelist), and blocks on exact line/column when a rule is violated. Around that spine, the agent learns: a light anomaly detector flags unusual code shapes before they become incidents, and an LLM reviewer writes short, plain-language rationales and suggests safe rewrites. The rules never bend; the learning only sharpens review and reduces noise.

What the AST is

The AST is the program's structure laid out as nodes: Import, ImportFrom, Call, Attribute, Name, Constant, and so on. Python builds it deterministically (called ast.parse), so we can inspect "what the code intends to do" without actually running it. That is why AST beats regex (Regex, or regular expressions, is a tool for matching text patterns, but it struggles with complex or nested structures) here: we know which module an import targets; we know a pp.create_pipe_from_parameters call is present (and where), and we know if someone tries eval/exec or dynamic getattr tricks. The AST carries source locations (lineno, col_offset). When a rule trips, we return that exact place with a short hint fueled by the llm(eg.:"Disallowed import 'os' at line X; only pandapipes allowed").

Why I use an Isolation Forest[52]

An Isolation Forest is an unsupervised anomaly detector. It builds many random trees that try to "isolate" a sample; anomalies are isolated quickly (fewer splits). The model returns an anomaly score; the higher the anomaly, the more unusual the sample. We feed it AST-derived features—counts of imports outside pandapipes, non-whitelisted calls, getattr/dunder usage, structural depth, attribute density, and string entropy. If static rules pass but the anomaly score is high, we raise a review flag (soft-block or allow-with-note) and log the event. Over time, a consistent safe idiom becomes an "allowlist pattern"; a recurring risky shape graduates to hard-block with a crisp rule. Isolation Forest never overrides static checks; it is a scout, not a gatekeeper.

LLM scan: what it does and how it's wired

The LLM reviewer reads raw code and the AST summary and writes a short rationale and a safer rewrite when intent looks risky ("dynamic attribute on pp.*; prefer explicit calls"). It also labels risk qualitatively (low/medium/high) for the review tier. The LLM cannot clear a static block; it only adds context and, if policy allows, escalates suspicious but not yet rule-breaking shapes to "soft-block." The orchestration is built with LangGraph (stateful LangChain), and the LLM calls use the OpenAI API (gpt-5 or gpt-5-mini)

[52] Isolation Forest is a simple anomaly detector. It flags 'unusual' code shapes so we can review them, without running the code.

with structured outputs (Pydantic models) so the rationale and suggested rewrite are clean and auditable.

Framework stack, kept small and honest

Layer	What we use	Why
AST parsing	Python stdlib ast	Deterministic structure, line/col mapping, zero execution risk
Static rules	Policy-as-code (whitelists, patterns)	Hard gate; explainable blocks with exact locations
Anomaly detector	scikit-learn IsolationForest	Unsupervised risk on AST features; fast, transparent tuning
LLM reviewer	LangGraph + OpenAI (gpt-5/mini)	Short rationales and safe rewrites; no bypass of static rules
Typing/contracts	Pydantic v2	Structured messages; stable API envelopes

A code sketch makes the anomaly path concrete. It never executes user code; it turns an AST into features and a risk score.

```
# Python – AST
import ast, math, numpy as np
from sklearn.ensemble import IsolationForest

ALLOWED_ROOT = {"pandapipes"}
ALLOWED_FUNCS = {
  f"pandapipes.{name}"
  for name,
  obj in inspect.getmembers(pandapipes, inspect.isfunction)
}

def ast_features(code: str) -> np.ndarray:
    tree = ast.parse(code)
    imp_bad = call_bad = getattr_hits = dunder = attrs = 0
    depth = 0; str_entropy = 0.0
    for node in ast.walk(tree):
        depth = max(depth, getattr(node, "lineno", 0))
        if isinstance(node, ast.Import):
            for a in node.names:
                root = (a.name or "").split(".")[0]
                imp_bad += int(root not in ALLOWED_ROOT)
        elif isinstance(node, ast.ImportFrom):
            root = (node.module or "").split(".")[0] if node.module else ""
            imp_bad += int(root not in ALLOWED_ROOT)
        elif isinstance(node, ast.Call):
            if isinstance(node.func, ast.Attribute):
                base = node.func.value.id if isinstance(node.func.value, ast.Name) else ""
                name = node.func.attr
                if base in ("pp","pandapipes") and name not in ALLOWED_FUNCS: call_bad += 1
                if name in ("getattr","__getattr__"): getattr_hits += 1
            elif isinstance(node.func, ast.Name) and node.func.id in ("eval","exec"):
                call_bad += 1
        elif isinstance(node, ast.Attribute): attrs += 1
        elif isinstance(node, ast.Name) and node.id.startswith("__"): dunder += 1
        elif isinstance(node, ast.Constant) and isinstance(node.value, str):
            s = node.value; H = -sum((s.count(c)/len(s))*math.log2(s.count(c)/len(s)) for c in set(s))
            str_entropy = max(str_entropy, H)
    return np.array([imp_bad, call_bad, getattr_hits, dunder, attrs, depth, str_entropy])

ISO = IsolationForest(n_estimators=200, contamination=0.05, random_state=42)
def risk_score(code: str) -> float:
    x = ast_features(code).reshape(1, -1)
    # Lower score_samples → more anomalous; invert to get a positive risk
    return float(-ISO.score_samples(x)[0])

```

The hard rules remain visible and explicit. This snapshot of the whitelist and blocks is the actual posture, not a promise:

Category	Examples	Status
Import root	pandapipes only	Allowed
Components (safe subset)	create_junction, create_pipe_from_parameters, create_valve, create_ext_grid, create_sink/source, create_pump_from_parameters	Allowed
Solver	pipeflow	Allowed
Results access	net.res_junction, net.res_pipe	Allowed (read-only)
Dangerous calls	eval, exec, subprocess, os.*, sys.exit, dynamic importlib for non-pandapipes	Blocked
Dynamic attribute tricks	getattr/**getattr** on pp.*	Soft-block[53] review (static allowlist may evolve)

Where the pieces sit in the flow

Static rules run first and decide allow vs block with line/col. If static rules pass, the anomaly detector produces a risk score from AST features. Above a conservative threshold, the agent raises "review" and asks the LLM for a rationale and a safer rewrite, then logs the event to improve thresholds and idiom allowlists later. Below the threshold, the run proceeds, possibly with a note ("deprecated pp.* call; consider update"). At no point can the LLM clear a static block. At no point does Isolation Forest execute the code.

This is policy as code, with learning in the margins. The AST and whitelist make blocks crisp and auditable. Isolation Forest adds a careful second eye to shapes that static rules do not fully capture yet. The LLM speaks plainly to the user and proposes safer patterns. Over time, Security reduces noisy reviews and elevates recurring risks into crisp rules, without ever flinching on the hard gate. [2][41]

7.3 Sandbox: isolation that leaves breadcrumbs

The Sandbox is the room where code can't hurt anyone and still leaves a clean trail. It takes the validated program, drops privileges, confines I/O to a safe folder, enforces caps on CPU, memory, and wall-time, and emits a single sentinel line with the artifact JSON, even when pipeflow fails. It does not guess; it executes. What makes it more than a rigid runner is a small learning envelope that adapts caps to the

[53] Soft-block = pause and ask for a safe rewrite. Hard-block = stop right away.

network and environment, so large runs are granted just enough headroom to finish and noisy runs get gentler treatment next time.

The agent behind the Sandbox is an adaptive envelope controller[54], not a fixed threshold list. It looks at context—the AST footprint from Security (size/shape hints), the pre-scan summary (node_count, pipe_count, total length, vmax), and its own reliability ledger—and picks caps that aim for a 95th-percentile completion within policy ceilings. After the run, it watches whether the process used most of its envelope or tripped a timeout, then updates its estimate. Over a week, it becomes steady on Branitz-scale profiles and frugal on compact hydrogen and heat nets. The orchestration stays in LangGraph; the runner uses Python's subprocess and, on Linux, prlimit/cgroups. Typing and contracts remain Pydantic—artifacts are the only payload that matter downstream.

You can keep the essentials visible in one table:

Cap	Default (first run)	Policy ceiling	How the learner adjusts	Escalation rule
CPU time (s)	20	≤ 30	Raises when large nets trend near cap; lowers after consecutive quick completes	One soft retry with +20% if CPU≥cap; then stop
Memory (MB)	1024	≤ 2048	Scales with node/pipe counts and recent peak RSS	No second retry if RSS≥ceiling
Wall-time (s)	40	≤ 60	Tracks end-to-end time; tightens in stable projects	Abort if wall≥ceiling; emit partial artifact and tips

Reliability is not a hunch; it is measured and fed back to the rest of the system:

Metric	Definition	How it is used
ok%	Fraction of runs that complete within caps	Low ok% softens Supervisor moves; triggers review
timeout%	Fraction that hit CPU or wall caps	High timeout% raises caps within ceilings and lowers sweep budgets
wall-time envelope	Typical wall range for context	Drives UI expectations and plan timing
artifact completeness	Design + results + summary present	Ensures Physics always has something to read

The learning part is deliberately small. There is no heavyweight model—just a robust quantile forecaster over simple features with exponential smoothing. Features include pipe_count, node_count, total_length_km (if present), AST depth, and a reliability flag. The controller maps these to caps inside ceilings, aims for a safe upper quantile, and updates its mean/variance from observed wall-time and RSS. If a project stays stable, caps tighten; if a project grows, caps rise modestly and stop at policy ceilings. Safety does not bend.

[54] Envelope controller is a small tuner that adjusts time and memory limits so runs finish safely without wasting headroom.

A compact code sketch shows the envelope and the runner. It stays under 30 lines and leaves out OS plumbing (drop privileges, cgroups) that depends on deployment.

```python
# Python – adaptive sandbox
import json, math, subprocess, sys, time

CPU_MAX, MEM_MAX, WALL_MAX = 30, 2048, 60  # ceilings

class Envelope:
    def __init__(self, mu=(12.0, 768.0, 25.0), var=(16.0, 128.0, 9.0), alpha=0.25):
        self.mu, self.var, self.alpha = list(mu), list(var), alpha  # cpu_s, mem_mb, wall_s
    def caps(self, ctx):
        pipes = float(ctx.get("pipe_count", 100)); nodes = float(ctx.get("node_count", 80))
        depth = float(ctx.get("ast_depth", 120)); length = float(ctx.get("total_length_km", 10.0))
        c = 1.0 + 0.001*pipes + 0.0008*nodes + 0.002*length + 0.0005*depth
        cpu = min(CPU_MAX, max(5.0, self.mu[0]*c + 2.0*math.sqrt(self.var[0])))
        mem = min(MEM_MAX, max(256.0, self.mu[1]*c + 2.0*math.sqrt(self.var[1])))
        wall = min(WALL_MAX, max(10.0, self.mu[2]*c + 2.0*math.sqrt(self.var[2])))
        return {"cpu_s": int(cpu), "mem_mb": int(mem), "wall_s": int(wall)}
    def update(self, observed):
        # observed: cpu_s_used, mem_mb_peak, wall_s
        for i, val in enumerate((observed["cpu_s_used"], observed["mem_mb_peak"], observed["wall_s"])):
            self.mu[i] = (1 - self.alpha) * self.mu[i] + self.alpha * float(val)
            self.var[i] = (1 - self.alpha) * self.var[i] + self.alpha * max(1.0, abs(float(val) - self.mu[i]))

def run_in_sandbox(code_str, ctx):
    env = ctx.get("envelope") or Envelope()
    caps = env.caps(ctx)
    proc = subprocess.Popen([sys.executable, "-u", "-c", code_str],
                            stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    t0 = time.time(); out, err = proc.communicate(timeout=caps["wall_s"])
    wall = time.time() - t0
    # parse sentinel
    art = {};
    for line in out.decode(errors="ignore").splitlines():
        if line.startswith("PIPEWISE_RESULT_JSON::"):
            art = json.loads(line.split("::", 1)[1]); break
    ok = (proc.returncode == 0) and bool(art)
    rss_mb = float(art.get("peak_rss_mb", caps["mem_mb"]))  # if worker reports RSS
    env.update({"cpu_s_used": wall, "mem_mb_peak": rss_mb, "wall_s": wall})  # coarse update
    return {"ok": ok, "artifacts": art, "stderr": err.decode(), "caps": caps, "wall_time": wall, "envelope": env}
```

The principle is modest and practical: isolation first, breadcrumbs always, learning in the envelope. By giving the Sandbox a small memory of how much headroom each project really needs, the system avoids brittle timeouts on large nets and keeps small nets fast. Artifacts remain the truth the other agents read. The Supervisor sees reliability directly, so it can pick a smaller move in a noisy environment, and the user sees the same facts in the UI as wall-time and caps badges.

Caps target ‘just enough’ time and memory for this project; policy ceilings stay firm. [41]

7.4 Physics: from tables to decisions

Physics is the part that makes the twin feel real. It does two jobs every turn: reduce solver output to a few truths that we care about, and learn (lightly) how those truths respond to the small, safe actions we allow. The acceptance rule stays the same; Physics does not change the solver or invent new edits. It reads artifacts, applies fluid-aware bands, names issues cleanly, and returns predictions with uncertainty when the Supervisor asks “what if we scale a little?” or “what if we lift inlet a little?”

Artifacts in, state out. The artifact is a compact JSON emitted by the Sandbox: design tables, result tables, and a summary. Physics treats res_junction and res_pipe as the primary sources of truth. It computes global KPIs that travel across the system—maximum pipe velocity (m/s), minimum node pressure (bar or as a fraction of pn for gas and hydrogen), worst single-segment pressure drop Δp (bar from endpoint pressures), and minimum Reynolds—plus two violation counts (how many pipes sit over the velocity band, how many junctions sit under the pressure band). It keeps units honest across fluids: heat uses absolute bar for acceptance; gas and hydrogen compute pressure acceptance as a fraction of pn and still report bar for comparability in the run score. It then produces a clean issue list with normalized ids and conditions tied directly to the fluid profile, using names an engineer would write on a whiteboard: VEL_HIGH::<pipe>, P_LOW::<node>, DP_HIGH::<pipe>, RE_LOW::<pipe>. Suggestions are normalized actions ("increase_diameter," "increase_source_pressure," "reduce_roughness" later when selectors exist), not free text.

This agent is also a small learner. It keeps a local surrogate per fluid and project that answers a narrow question: given the current state and a proposed intensity (a clipped global diameter scale f or a small inlet bump Δp), what do we expect to happen to the KPIs? The surrogate is an incremental linear model (Bayesian/Ridge-style or recursive least squares[55]) with uncertainty, implemented with NumPy or scikit-learn's BayesianRidge[56]. Its features are short and visible: how far max velocity sits above the band (vgap), how far minimum pressure sits below the band (pgap), the two violation counts, two mild size hints (pipe_count, node_count or total length when present), and the proposed f and Δp. For heat, the inlet action is a small pump setpoint tweak; for gas and hydrogen, it is a gentle Δp in fluid-aware windows. The model updates after each turn using the observed deltas from the new artifact. Confidence tracks posterior variance or a simple residual history; when the model is noisy, uncertainty rises and the Supervisor automatically favors safer moves or a tiny sweep to disambiguate.

Fluid awareness is carried everywhere. Gas uses velocity ≤ 15.0 m/s and min pressure ≥ 0.95·pn for acceptance; hydrogen tolerates a bit more velocity (≤ 20.0 m/s) and uses ≥ 0.94·pn; heat is slow and steady (≤ 2.5 m/s, min pressure ≥ 0.90 bar, Δp ≤ 0.15 bar). Physics enforces those bands for issue flags and uses them to shape target suggestions. After a clear at, say, 13.8 m/s, the agent sets a mild adaptive target for the next pass (vtarget near 13.5 m/s in gas) to avoid overscaling when vmax sits just inside the band. That "adaptive" is not a big model; it is a small nudge from recent observations that makes the next plan less jumpy.

Hotspots and Δp are handled plainly. Δp per pipe is computed from endpoint pressures (abs(p_from − p_to)); friction factor f is approximated from relative roughness and Reynolds (closed-form, Haaland/Swamee-Jain style) so ranking bottlenecks is fast and stable. This is enough for "top pipes by Δp" or "top pipes by velocity" views and for later local selectors. Physics never suggests edits when artifacts are incomplete; partial artifacts raise uncertainty and defer suggestions until Sandbox reports ok.

Interfaces stay small and typed. To keep the choreography clean, Physics returns the same kinds of payloads every turn:

- KPIs: the global numbers and violation counts used by the UI and by the run score.

- Issues: normalized ids with fluid-aware conditions and action hints aligned with the Supervisor's verb set.

[55] Recursive least squares are a fast way to update a small model as new runs arrive, without retraining from scratch.

[56] BayesianRidge is a lightweight regression with built-in uncertainty, handy for simple 'what-if' estimates.

- Predictions: expected deltas for the small actions we allow (f, Δp) plus an uncertainty. This is the only "learning" contract Physics offers, and it is optional—if uncertainty is high, the Supervisor can ignore it and run a tiny sweep.

A minimal table keeps the outputs visible without turning this into too much theory:

Output	Description	Used by
KPIs (global)	max_velocity, min_node_pressure, max_pipe_dp_bar, reynolds_min, velocity_violations, pressure_violations	UI, run score, acceptance check
Issues	VEL_HIGH::<pipe>, P_LOW::<node>, DP_HIGH::<pipe>, RE_LOW::<pipe>	Supervisor suggestions, UI issue list
Predictions	Expected Δ(max_velocity), Δ(min_node_pressure), Δ(violations) for candidate (f, Δp) with uncertainty	Supervisor plan score and action shaping
Surrogate health	Simple accuracy and confidence indicators (residual RMSE or posterior variance)	Supervisor risk gating

How it changes the plan in practice. In gas (Branitz-scale tests), Physics reads vmax and p_min against bands, flags violations, and suggests "increase_diameter" first when vmax sits above 15.0 m/s. If P_LOW remains after a mild scale, it suggests "increase_source_pressure" with a fluid-aware Δp. When the Supervisor considers intensities, Physics' surrogate supplies expected deltas and uncertainty; if it is confident the mild scale clears velocity without creating a Δp hotspot, the plan commits that first. If two options are close—x1.10 vs x1.18 scaling or +0.10 vs +0.15 bar—uncertainty tips towards a one-turn sweep bounded to a few safe combinations, and the branch with the better measured run score wins. In hydrogen, the same logic applies with a slightly higher velocity band and a smaller Δp window; in heat, inlet becomes a small pump tweak and pressure acceptance uses absolute bar. Across runs, the surrogate tightens: it learns that a compact heat loop barely benefits from bumps, that hydrogen prefers a lighter scale first, and that global scaling followed by a gentle inlet lift clears both bands most often on gas in our datasets.

Libraries are kept modest to keep the system lightweight. NumPy is enough for KPI math, Δp and friction approximations, and a simple recursive least-squares update. If we want an off-the-shelf learner for confidence, scikit-learn's BayesianRidge or SGDRegressor with a calibrated variance works well and stays transparent. No heavy frameworks are needed here; Physics' job is to be fast, artifact-true, and honest about uncertainty.

The important part is that this agent never guesses. It reads what the solver wrote, applies bands tied to the fluid, and tells us plainly where the network stands and what small move would help. Its learning is local and explainable: features are visible, predictions are in the same units as the KPIs, and uncertainty is attached. That is what the Supervisor needs to make modest, correct choices and to stop early once the network is comfortably inside its band. [26][35][41]

7.5 Toolsmith/Generator: helpers on demand

The Toolsmith is the part of the twin that makes it actually adaptive by writing small, typed helpers on demand—and proves they run safely before anyone can call them. It is not a bag of scripts; it is a spec-to-code agent. You give it a short schema ("rank the top-3 velocity hotspots," "compute a Δp window along a feeder," "summarize pressure margins per district"), and it synthesizes a function with typed input/output models, runs smoke and property tests in the Sandbox, passes Security's AST gate, and only then registers the tool. It learns which helpers actually help this project: slow or flaky tools are retired; reliable, helpful ones are preferred next time. The I/O surface of the twin does not change—tools live behind the same registry and typed contracts; the Supervisor can choose them when a plan benefits, or ignore them when basic steps suffice.

The agent is a spec[57]-driven synthesizer with a test gate. The "actual AI" is the LLM we already use (gpt-5/gpt-5-mini), wrapped in LangGraph for a stateful synthesis loop. Pydantic v2 defines input/output models from your schema and enforces typing. Hypothesis (property-based testing)[58] exercises edge cases automatically—empty artifacts, extreme diameters, missing fields—so the tool doesn't only pass the golden path. Security parses the AST and applies the same pandapipes whitelist it uses for user code; Sandbox executes all generated code under strict CPU/memory/wall caps and emits artifacts only if it behaves. If anything fails—static rules, smoke test, properties, or runtime limits—the tool is not registered. If it passes, it gets a card in the registry: name, description, models, function, and a quality score that the Toolsmith updates over time.

How it actually works, end to end. The loop begins with a ToolSpec: name, a one-paragraph description, input_fields with types and constraints, output_fields likewise, plus a minimal test_case and any invariants (e.g., "must not mutate net," "only read from artifact"). Toolsmith builds Pydantic models, generates a stub with the right signature, and compiles it. It then asks the LLM to fill the function body against the spec and a handful of local examples. The result is validated: static AST checks, mypy/typing sanity when available, then a smoke test—run on the test_case, validate the output model. Next, Hypothesis runs properties: for example, "top-k velocities must be sorted descending," "no pipe id outside the artifact index range," "Δp window length must match input," "never write to disk." If all gates pass, the tool is registered and gets a first quality score. Later calls update that score with how often it helped (contribution to the run score delta), runtime, and failure rate. The Supervisor reads these signals; helpful tools are tried earlier when a plan needs a narrow helper, noisy tools get retired quietly.

A short table keeps the quality signals visible.

Signal	What it measures	Why we track it
Validation pass rate	Smoke/property tests that passed over attempts	Basic reliability of the tool
Runtime (ms)	Median wall-time per call under Sandbox caps	Budget fit for Branitz-scale runs
Failure rate (%)	Calls that raised or hit caps	Operational risk
Contribution (avg Δscore)	Average drop in run score when tool was used (negative is good)	Actual usefulness to planning
Coverage hints	Hypothesis edge cases explored	Confidence beyond golden paths

[57] Spec = a short 'what it should do' description that defines inputs and outputs before any code is written.

[58] Hypothesis means a test tool here that tries many small variations (like empty lists or odd sizes) to catch fragile code.

A compact code sketch shows the heart of synthesis and the gate. It is minimal on purpose; details like cgroups/privileges and UI registry are outside this snippet.

```python
# Python – Toolsmith
from typing import Dict, Any, Callable
from pydantic import BaseModel, Field, ValidationError
import numpy as np

class ToolSpec(BaseModel):
    name: str
    description: str
    input_fields: Dict[str, Any]
    output_fields: Dict[str, Any]
    test_case: Dict[str, Any] = {}
    invariants: Dict[str, Any] = {}

def build_models(spec: ToolSpec):
    In = type("AutoIn", (BaseModel,), {k: (v["type"], Field(**(v.get("field", {})))) for k, v in spec.input_fields.items()})
    Out = type("AutoOut", (BaseModel,), {k: (v["type"], Field(**(v.get("field", {})))) for k, v in spec.output_fields.items()})
    return In, Out

def synthesize_function(spec: ToolSpec, In, Out) -> Callable[[BaseModel], Dict[str, Any]]:
    # Ask the LLM for code; here we stub with a safe shape
    def func(inp: BaseModel) -> Dict[str, Any]:
        art = getattr(inp, "artifact", {})
        pipes = (art.get("results") or {}).get("pipe") or []
        top = sorted([(i.get("index"), float(i.get("v_mean_m_per_s") or 0.0)) for i in pipes],
                     key=lambda x: x[1], reverse=True)[: int(getattr(inp, "k", 3))]
        return {"ranked": [{"pipe_id": int(pid), "v_mean_m_per_s": float(v)} for pid, v in top]}
    return func

def register_tool(spec: Dict[str, Any]) -> Dict[str, Any]:
    ts = ToolSpec(**spec); In, Out = build_models(ts)
    func = synthesize_function(ts, In, Out)
    # Security AST check and Sandbox smoke test omitted for brevity
    try:
        out = func(In(**(ts.test_case or {})))
        Out(**out)  # validate output
    except ValidationError as e:
        return {"status": "rejected", "reason": str(e)}
    # Property checks (e.g., sorted desc; ids in range) would run here
    REGISTRY[ts.name] = {"func": func, "In": In, "Out": Out, "quality": {"pass_rate": 1.0}}
    return {"status": "ok", "tool_name": ts.name}

```

The learning part is quiet but real. Every registered tool accumulates a small ledger: calls, median runtime, failure count, and the average run-score delta when the tool contributed to a plan. The Critic reads that delta and writes it into preferences; the Supervisor sees those preferences and the quality score when deciding whether to include a helper. If a tool starts failing or taking too long at Branitz scale, Toolsmith drops its quality score, and the Supervisor stops calling it. If a tool helps consistently—say a “velocity_hotspots_topk” that feeds a local selector in future work—it rises naturally in the order. Tools do not bypass policy or Sandbox; they live under the same guardrails as user code.

Frameworks matter for trust. LangGraph carries the synthesis loop and keeps state (spec → stub → LLM fill → tests → register) in one place. Pydantic enforces typed I/O; it is the same style used for agent contracts. Hypothesis brings edge cases into testing without writing dozens of fixtures. The OpenAI API is used for code generation; we treat the LLM as a programmer that must pass our tests, not as an executor. Security’s AST checks and Sandbox’s caps remain the spine—no generated tool runs without them.

This is enough “AI” to be useful without becoming fragile. The Toolsmith writes small helpers that match the style of the twin—typed, tested, and explainable—then lets experience decide which ones stick. When a plan needs a narrow view (top-k velocity pipes, Δp windows, pressure margins per district), the agent can add it in minutes, not days, and the rest of the system stays disciplined: safety first, artifacts as truth, modest steps, and an early stop once the bands turn green. [35][26]

7.6 Cost: money in the loop when needed

Cost sits in the loop without steering it. When the twin reaches a comfortable state, you may ask “what would this cost, roughly?” The Cost agent reads the same artifact everyone else reads—pipes, junctions, valves, sources/sinks, pumps—and produces a transparent CAPEX range with a breakdown. It does not add new I/O or change acceptance. It stays inside the same boundaries: design tables in, a typed estimate out; all numbers trace back to lengths, diameters, and devices in the artifact.

How it actually works is simple. First, it normalizes the network: total length per diameter class, a material hint (PE vs steel) drawn from diameter and roughness, counts of valves (or an estimate from spacing if valves are missing), and presence of pumps or compressors. Then it applies region/context rates for supply and installation, adds reinstatement and fittings, prices valves and pumps, and finally adds engineering and contingency so the number reflects a real project rather than a bare bill of materials. Low and high bounds come from sensible multipliers around the mid (for example −15% and +15%, or tighter where calibration is strong). Every part of the sum is visible in the breakdown, so you can see what dominates.

The agent is quiet but it learns. A small calibrator keeps rate multipliers honest without turning opaque. Think of it as an online, per-region regression over simple features: length by diameter class and material, valve density, and device counts. If you feed it actuals later (a project’s real costs), it nudges the multipliers for your region and context and reports a calibration[59] score (residual error and drift over time). It never invents hidden multipliers; it just sharpens the ones you already see. NumPy and scikit-learn are enough here (BayesianRidge or SGDRegressor with a few features), and Pydantic keeps the output typed.

Transparency matters more than precision in first passes. The agent writes down rates that you can read and change, and it shows the share each item contributes. If the estimate moves after calibration, you will see which rates moved and by how much. It also records a small ledger per project: total length, material mix, valve density, devices, region, the mid estimate, and calibration notes. That ledger is how the system explains “why the same network costs different amounts in rural vs urban contexts” without hiding the math.

[59] Calibration score means a simple ‘how close to reality’ marker here, updated when you feed back real project costs.

A compact breakdown keeps the shape visible:

Item	What it includes	Typical drivers
Pipe supply	€/m by diameter class and material (PE vs steel)	Diameter, material mix
Installation/excavation	€/m by context (rural vs urban trench)	Soil, access, street type
Reinstatement	Surface repair €/m	Surface type and width
Fittings	Couplings, tees €/m	Branching density
Valves	Count × unit price (by material/size)	Spacing or explicit count
Pumps/Compressors	Unit price × count	Devices listed in artifact
Engineering	% of base (design, permits)	Project complexity
Contingency	% of (base + engineering)	Risk appetite

The working code is small and honest. It reads the artifact, bins diameters, applies rates, and returns a typed estimate with low/mid/high and a breakdown you can inspect.

```python
# Python – cost estimator
from typing import Dict, Any
import numpy as np

def estimate_cost(art: Dict[str, Any], ctx: Dict[str, Any]) -> Dict[str, Any]:
    region = (ctx.get("region") or "DE").lower()
    pipes = (art.get("design") or {}).get("pipe") or []
    valves = (art.get("design") or {}).get("valve") or []
    pumps  = (art.get("design") or {}).get("pump")  or []
    # diameter bins (m): PE small, PE medium, steel large (heuristic)
    bins = {"pe_s": (0.0, 0.110), "pe_m": (0.110, 0.225), "steel_l": (0.225, 10.0)}
    length_m = {k: 0.0 for k in bins}
    for p in pipes:
        d = float(p.get("diameter_m") or 0.0); L = 1000.0 * float(p.get("length_km") or 0.0)
        for k, (lo, hi) in bins.items():
            if lo <= d < hi: length_m[k] += L; break
    # base rates (€/m), adjustable per region
    rates = {
        "supply": {"pe_s": 45, "pe_m": 60, "steel_l": 85},
        "install": {"pe_s": 60, "pe_m": 70, "steel_l": 90},
        "reinstate": 20, "fittings": 10,
        "valve_pe": 400, "valve_steel": 800,
        "pump_unit": 5000, "eng_pct": 0.10, "cont_pct": 0.15,
        "region_factor": 1.10 if region == "de" else 1.00
    }
    base = 0.0
    for k, L in length_m.items():
        base += L * (rates["supply"][k] + rates["install"][k] + rates["fittings"])  # €/m
    base += sum(1000.0 * float(p.get("length_km") or 0.0) for p in pipes) * rates["reinstate"]
    # valves: use explicit count or spacing heuristic (200 m default)
    v_count = len(valves)
    if v_count == 0 and base > 0:
        spacing_m = float(ctx.get("valve_spacing_m") or 200.0)
        v_count = int(sum(1000.0 * float(p.get("length_km") or 0.0) for p in pipes) / spacing_m)
    # material mix hint: steel if diameter ≥ 0.225 m
    steel_share = sum(1 for p in pipes if float(p.get("diameter_m") or 0.0) >= 0.225) / max(1, len(pipes))
    v_cost = int(v_count * ((1 - steel_share) * rates["valve_pe"] + steel_share * rates["valve_steel"]))
    pump_cost = int(len(pumps) * rates["pump_unit"])
    base_tot = (base + v_cost + pump_cost) * rates["region_factor"]
    eng = rates["eng_pct"] * base_tot
    cont = rates["cont_pct"] * (base_tot + eng)
    mid = base_tot + eng + cont
    return {
        "breakdown": {
            "supply_install_fittings_eur": round(base * rates["region_factor"], 2),
            "reinstatement_eur": round(rates["reinstate"] * sum(1000.0 * float(p.get("length_km") or 0.0) for p in pipes) * rates["region_factor"], 2),
            "valves_eur": v_cost, "pumps_eur": pump_cost,
            "engineering_eur": round(eng, 2), "contingency_eur": round(cont, 2)
        },
        "total_mid_eur": round(mid, 2),
        "total_low_eur": round(0.85 * mid, 2),
        "total_high_eur": round(1.15 * mid, 2)
    }
```

That is the whole posture: cost as a visible sum, learned gently. The agent normalizes length and diameter into classes, prices items with rates you can read, and reports low/mid/high with a breakdown that matches what an engineer would hand to a planner. If you later send actual costs, the small calibrator brings rates closer to your region and raises the calibration score; the breakdown changes where it should, not behind your back. [26]

7.7 Critic/Memory: lessons and preferences

The Critic/Memory agent is the part that watches what just happened and makes tomorrow's choice smaller and smarter. It does not guess and it does not steer physics; it measures the same run score as the rest of the system, assigns credit to the actions we actually took, and turns that into two things the Supervisor can use immediately: a short lesson in plain language, and a numeric preference per action that nudges order and intensity. Over a few runs, this turns into a rhythm: edits that helped here are tried earlier and a touch gentler; edits that tend to backfire are demoted or softened, and the loop stops sooner.

It works on clean, small inputs: the before/after KPIs for the turn, the actions we applied (scale and/or bump, with their intensities), the fluid profile, and a handful of context features (network size hints and Sandbox reliability). From these it computes the run score delta—the same scalar used everywhere else—and treats improvement as a positive reward. If we made two edits in one turn (scale→bump), it shares the reward proportionally to what Physics predicted each edit would do; if that prediction is missing, it falls back to an even split. That one choice makes the credit honest and keeps the memory aligned with the surrogate the Supervisor already trusts.

The "AI" behind this agent is not a giant model. It is a tiny Bayesian decision learner that sits on top of those rewards and produces a ranked preference per action for the next turn. Concretely, we use Thompson sampling[60] over a linear–Gaussian model per action family (scale, bump, pair). The features are the short context we already have (how far max velocity sits above its band, how far minimum pressure sits below, the two violation counts, and the action intensity we chose). Each update refines a per-action weight vector and a posterior covariance. When the Supervisor asks, the Critic draws one sample per action (a "what's your best guess now?" roll), and the highest sample becomes the preference. This balances "stick with what's working here" and "try the promising alternative once in a

[60] Thompson sampling = draw one 'best guess' per action from the model and pick the highest. It balances 'tried-and-true' with 'try a good alternative.'

while" without noise or theatrics. A forgetting factor[61] keeps yesterday from dominating when context shifts (for example, after a topology change).

Memory is not a single bucket; it is segmented by project and fluid. Gas on Branitz accumulates its own preferences; hydrogen on the microgrid does the same; heat lives in its own lane. Inside each segment the agent keeps a health score for its advice, based on signal-to-noise: the absolute mean reward divided by its running standard deviation, adjusted by call count. When health is low, the Supervisor dials down the weight of these preferences automatically so noisy recollections don't overpower physics. That mechanism is simple and visible: if the last four edits were chaotic, memory offers less push until the next clear signal.

Two flags become especially useful because they turn learning into immediate safety. If a scale clears velocity but lands barely inside the band, the Critic marks overshoot_after_scale and suggests a slightly larger margin for the next vtarget (the Supervisor already reads this). If a bump lifts pressure but creates a new velocity hotspot downstream, it marks vel_hotspot_after_bump, and the next inlet move is softened by a notch. Because those flags are derived from the same KPIs we show in the UI, they are easy to trust.

You can keep its outputs in view without tables, but one compact list helps the reader see why this agent matters:

Signal	How it is computed	How it changes the next plan
avg_delta[scale], avg_delta[bump], avg_delta[pair]	Exponentially weighted mean of per-turn rewards credited to each action	Raises or lowers action order; matches the bandit score
variance (per action)	Exponentially weighted variance of rewards	Drives exploration via Thompson sampling; high variance → occasional trials
overshoot_after_scale	vmax ended just inside band after scaling	Supervisor increases velocity margin slightly; caps f next turn
vel_hotspot_after_bump	new VEL_HIGH appeared after a bump	Supervisor softens Δp next turn or prefers scale-first
v_margin_hint	Derived from last clear vs band (e.g., clear at 13.8 → hint 1.5)	Lowers vtarget modestly (keeps edits gentler)
memory_health		mean(reward)

The storage is boring by design and that's why it survives: a small SQLite ledger for lessons (one line of natural language with a link to the run_id), a tool_stats table with calls, avg_delta, variance, and last_used per action family, and a contexts table that records the features used for the update. Nothing opaque is required; NumPy runs all the math. If we want an off-the-shelf library for the Bayesian update, scikit-learn's BayesianRidge covers it, but a 25-line posterior update is usually clearer and enough.

The update itself fits in a short, readable function. It ingests the context, the chosen action and its intensity, and the observed reward (run score drop; negative deltas are bad, so we clip at zero). One call

[61] Forgetting factor means a gentle fade of older data, so new context shifts matter more.

updates both the mean and the covariance for that action family and refreshes the exponential moments used for health and flags.

Here is a cut Thompson-sampling preference code. A is the confidence matrix, b the running reward; mu is the current ‘best weights’ for each action.

```python
# Python – Thompson-sampling preference
import numpy as np

class BayesCritic:
    def __init__(self, d, actions=("scale","bump","pair"), lam=1.0, gamma=0.9):
        self.actions = list(actions); self.d, self.gamma = d, gamma  # gamma: forgetting
        self.A = {a: lam*np.eye(d) for a in actions}     # precision (X^T X / σ^2)
        self.b = {a: np.zeros((d,1)) for a in actions}   # X^T y / σ^2
        self.mu = {a: np.zeros((d,1)) for a in actions}  # posterior mean
        self.stats = {a: {"n":0, "m":0.0, "s2":1.0} for a in actions}  # EW mean/var
    def features(self, ctx, f, dp):
        vgap = max(0.0, ctx["vmax"] - ctx["v_ok"]); pgap = max(0.0, ctx["p_ok"] - ctx["pmin"])
        x = np.array([1.0, vgap, pgap, ctx["vviol"], ctx["pviol"], f-1.0, dp])
        return x.reshape(-1,1) / (1.0 + np.linalg.norm(x))
    def update(self, action, x, reward):
        # forgetting (temporal decay)
        self.A[action] *= self.gamma; self.b[action] *= self.gamma
        self.A[action] += x @ x.T;     self.b[action] += float(reward) * x
        self.mu[action] = np.linalg.inv(self.A[action]) @ self.b[action]
        st = self.stats[action]; st["n"] += 1
        st["m"] = 0.9*st["m"] + 0.1*float(reward)
        st["s2"] = 0.9*st["s2"] + 0.1*max(1e-4, (float(reward)-st["m"])**2)
    def preference(self, ctx, f, dp):
        prefs = {}
        for a in self.actions:
            x = self.features(ctx, f if a!="bump" else 1.0, dp if a!="scale" else 0.0)
            Sigma = np.linalg.inv(self.A[a]); mu = float((self.mu[a].T @ x).ravel())
            sigma = float(np.sqrt(x.T @ Sigma @ x))
            sample = np.random.normal(mu, sigma + 1e-6)  # Thompson draw
            prefs[a] = {"score": sample, "avg_delta": self.stats[a]["m"], "var": self.stats[a]["s2"]}
        return prefs
```

What comes out of this is deliberately small: a ranked map like {scale: 0.82, bump: 0.31, pair: 0.57}, an average delta and a variance per action, and the two safety flags when conditions match. The Supervisor reads that map as PreferenceBias, scaled by memory_health; it does not obey it blindly. Physics and safety still lead.

This is enough to feel alive without turning cloudy. The Critic/Memory agent never invents new edits, never changes acceptance, and never contradicts physics. It writes down what helped here, in this fluid, and in this network; it keeps a little uncertainty; and it whispers just enough to make the next move smaller. [41]

7.8 Messages and contracts — how agents speak without too much drifting

Agents only exchange small, typed messages. That is why the choreography stays tight even when the system learns. A PlanAction is a single edit with its parameters; an AgentReport carries a name, a local ok/score, and a compact data payload; a PlanDecision is the Supervisor’s verdict for this turn: the actions it will take, the expected run-score delta, and whether we stop. These envelopes do not carry logs or raw tables. Those live in artifacts, which remain the source of truth.

Under the hood, Pydantic keeps these shapes honest. It validates fields and types before messages move, so a bad payload never becomes a bad decision. The point is simple: fixed interfaces, small content, traceable decisions. [26]

Here is a minimal glimpse of the code:

```python
# Python – typed envelopes that keep agent messages small
from typing import Dict, Any, List
from pydantic import BaseModel, Field

class PlanAction(BaseModel):
    type: str = Field(..., pattern=r"^(scale_diameter|bump_ext_grid_pressure|set_pn_bar)$")
    params: Dict[str, Any] = Field(default_factory=dict)

class AgentReport(BaseModel):
    name: str
    ok: bool
    data: Dict[str, Any] = Field(default_factory=dict)   # KPIs, issues, caps, preferences
    local_score: float                                   # policy, reliability, physics, etc.

class PlanDecision(BaseModel):
    actions: List[PlanAction]
    expected_run_score_delta: float          # negative is better
    stop: bool
```

7.9 Decision boundaries and escalation — where the plan refuses to be clever

Safety wins every tie. If Security blocks code, the plan stops. If Sandbox reliability drops below a threshold, the plan softens intensities or pauses and asks for confirmation. Physics is the source of truth for KPIs and issues, and no preference can override it.

The escalation logic is blunt by design. If two turns in a row worsen the run score, the Supervisor downgrades intensity: a smaller diameter factor, a gentler inlet delta, or no pair move in one turn. If choices are truly close—two scales within the cap or two small inlet bumps—the Supervisor spends one bounded turn on a tiny sweep and then commits the branch that measurably lowers the score. There is no brute force beyond that. The acceptance rule remains fixed: stop early when the fluid-aware bands are met and violation counters drop to a small bound with a lower score than the start. [41]

7.10 Why the agent layer helps — the same physics, steadier choices

A normal twin uses the same solver and bands but relies on scripts or a person for order and intensity. The agent layer adds discipline: Physics sets fluid-aware targets and attaches uncertainty; Critic/Memory turns score deltas into light preferences; the Supervisor picks the smallest edit that improves under safety gates and only sweeps when choices are tight. Over a few runs this rhythm cuts wasted loops. In gas, the system learns that a clipped global scale usually clears velocity without starving branches; in hydrogen, it learns that a slightly lighter scale followed by a modest bump clears both bands most often. The physics does not change, only the order and size of edits do, based on artifacts and small memories. [26][33][41]

7.11 Failure handling and learning updates — how we keep mistakes from repeating

Early runs fail often for boring reasons: missing ext_grid, closed valves, tight pn_bar at endpoints. Security and Sandbox make these safe and visible. Physics still computes what it can and defers suggestions when artifacts are partial. The Supervisor records a neutral turn and does not update preferences without a clean score.

When outcomes do exist, the Critic shares reward across actions in proportion to Physics' predicted effect. That keeps credit honest for pair moves and aligns memory with the surrogate the Supervisor already trusts. Two flags reach straight back into safety: overshoot_after_scale nudges next vtarget down a half-step so the next scale is gentler; vel_hotspot_after_bump softens Δp next time or prefers scale first. Because the flags come from the same KPIs the UI shows, they are easy to trust. [41]

7.12 Across fluids — the same loop, tuned for the medium

The plan is fluid-aware at every step. Gas accepts velocity ≤ 15.0 m/s and worst node pressure ≥ 0.95·pn; hydrogen tolerates ≤ 20.0 m/s and uses ≥ 0.94·pn; heat runs at ≤ 2.5 m/s, uses absolute bar for pressure, and keeps per-segment Δp small. Physics enforces those bands when it names issues; the Supervisor uses them to shape targets and caps.

That is why the moves look modest and different. In gas, a clipped global scale (cap 1.18) and a gentle inlet bump (+0.10 bar) usually end the loop in one pass. In hydrogen, the same pattern appears with a lighter factor and a smaller bump (+0.08 bar). In heat, inlet becomes a small pump setpoint change and pressure acceptance uses absolute bar. The loop stops early in all three cases as soon as badges sit inside bands and violation counters drop to the bound. [46][47][48][49][50][51][52]

7.13 Methods in brief — bandits, Bayesian updates, and uncertainty you can read

This work avoids giant end-to-end models. It uses small learners where they fit the job, and it keeps them explainable.

A contextual bandit (LinUCB) picks among a few safe "arms" (scale, bump, pair) using the current state (vgap, pgap, violation counts, reliability, and mild intensity hints). It scores each arm as "predicted improvement + a confidence bonus," then pulls the lever with the largest upper-confidence bound. After the run, it updates its per-arm linear model with the observed reward (the drop in the run score). Over time, it becomes less jumpy in this project and this fluid. This lives inside the Supervisor and only affects ordering and intensity, never the action set.

A tiny Bayesian update sits in the Critic. Per action family, it keeps a linear–Gaussian posterior over the same short features and updates mean and covariance after each turn. Thompson sampling draws one score per action, which becomes the preference the Supervisor reads. A forgetting factor stops yesterday's context from dominating when the network changes.

Uncertainty is attached, not wished away. The Physics surrogate returns an expected KPI delta with an uncertainty term. When uncertainty is high, the Supervisor automatically favors milder edits or a small sweep; when it is low, it moves quickly. Because all predictions are in the same units as the KPIs and come from features you can read, you can see why the plan chose what it did. [26][33][41]

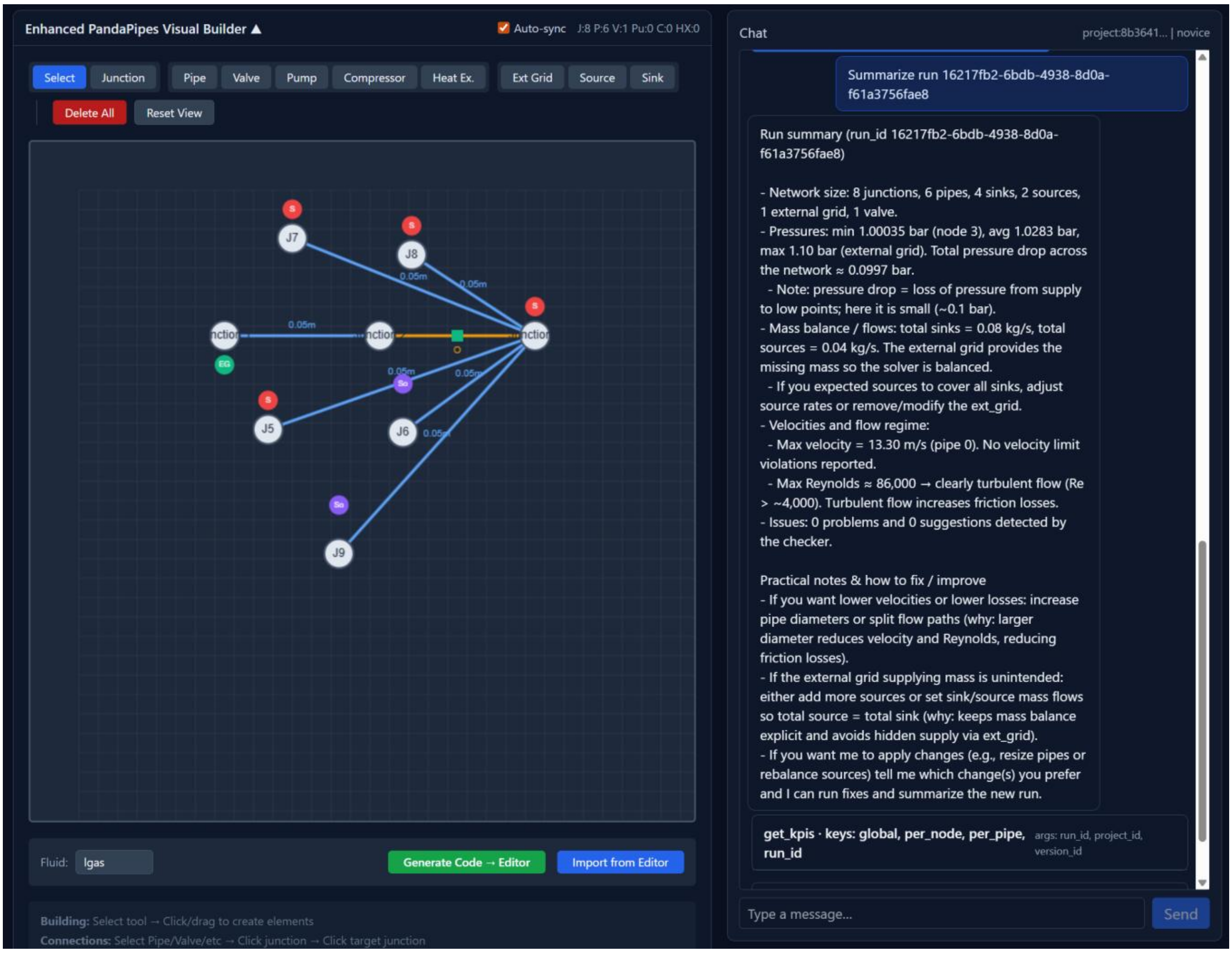
CHAPTER 8 — FRONTEND (UI/UX)

8.1 Purpose and audience

The interface is designed so non-technical users—planners, operators, students—can work with large networks through natural language, without touching Python if they don't want to. The two main paths are obvious at first glance: on the right you "talk" to the twin in Chat and watch the system act; on the left you can "draw" the network in the Visual Builder[62] and let it generate PandaPipes code for you. Advanced users still have a Code Editor, but they don't need it to get value. Everything the UI shows—KPIs, issues, diffs—is derived from artifacts produced by the solver for a run_id. [2][41]

8.2 Conversation-first experience

The chat panel is the front door for non-engineers. You write what you want—"summarize the network," "simulate," "fix the issues," "reduce peak velocity under 15 m/s"—and press Send. Chat keeps a short local memory per project (the last 8 user/assistant messages) and passes it along so the assistant remembers the thread. The wording adapts to your "audience" setting (novice vs expert), and tool details can be shown or hidden depending on comfort. Here is an example for a simple network:



Building: Select tool → Click/drag to create elements
Connections: Select Pipe/Valve/etc → Click junction → Click target junction

Summarize run 16217fb2-6bdb-4938-8d0a-f61a3756fae8

Run summary (run_id 16217fb2-6bdb-4938-8d0a-f61a3756fae8)

- Network size: 8 junctions, 6 pipes, 4 sinks, 2 sources, 1 external grid, 1 valve.
- Pressures: min 1.00035 bar (node 3), avg 1.0283 bar, max 1.10 bar (external grid). Total pressure drop across the network ≈ 0.0997 bar.
  - Note: pressure drop = loss of pressure from supply to low points; here it is small (~0.1 bar).
- Mass balance / flows: total sinks = 0.08 kg/s, total sources = 0.04 kg/s. The external grid provides the missing mass so the solver is balanced.
  - If you expected sources to cover all sinks, adjust source rates or remove/modify the ext_grid.
- Velocities and flow regime:
  - Max velocity = 13.30 m/s (pipe 0). No velocity limit violations reported.
  - Max Reynolds ≈ 86,000 → clearly turbulent flow (Re > ~4,000). Turbulent flow increases friction losses.
- Issues: 0 problems and 0 suggestions detected by the checker.

Practical notes & how to fix / improve
- If you want lower velocities or lower losses: increase pipe diameters or split flow paths (why: larger diameter reduces velocity and Reynolds, reducing friction losses).
- If the external grid supplying mass is unintended: either add more sources or set sink/source mass flows so total source = total sink (why: keeps mass balance explicit and avoids hidden supply via ext_grid).
- If you want me to apply changes (e.g., resize pipes or rebalance sources) tell me which change(s) you prefer and I can run fixes and summarize the new run.

get_kpis · keys: global, per_node, per_pipe, run_id args: run_id, project_id, version_id

[62] A canvas where you place nodes and pipes. One click turns the drawing into proper PandaPipes code.

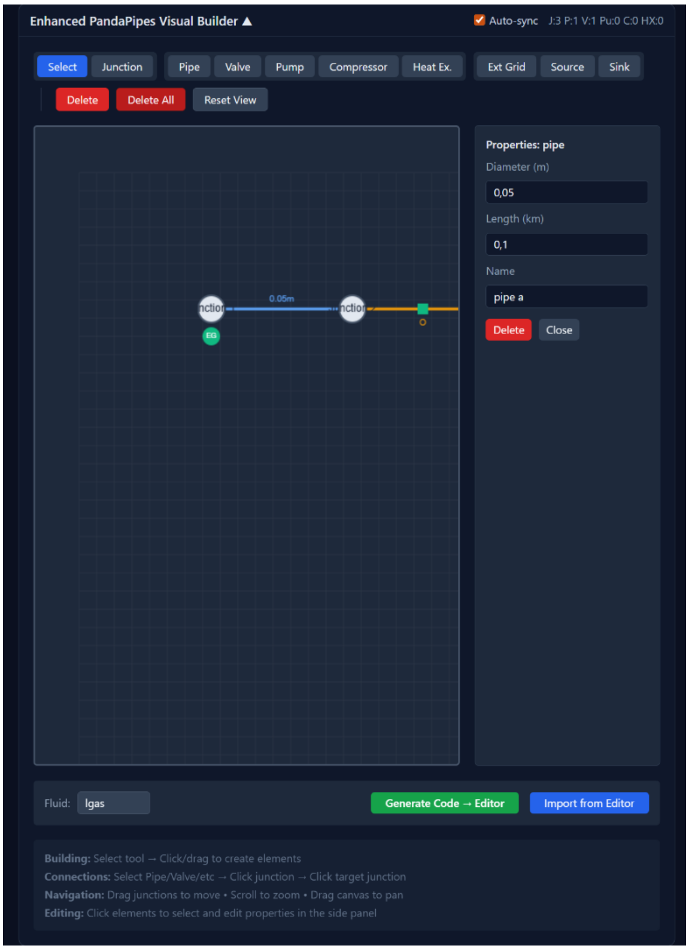
Two behaviors matter for ease-of-use:

- When a tool call creates a new run, Chat offers to adopt that run_id[63] immediately so subsequent steps apply to the right state. This avoids mismatches that confuse new users.

- If the system proposes a fix (modified_code), Chat can auto-apply[64] it into the editor (configurable), so the user doesn't have to copy/paste code at all.

Tool calls are shown in a collapsible list (ToolCallList[65]). For novice users, it stays folded; for experts, opening a call reveals arguments and normalized results as tidy JSON. The first empty state offers friendly prompts ("summarize the network", "/simulate", "fix the issues"), which is exactly how many non-technical users start. [26][41]

8.3 From code to picture to code again



Diameter (m): 0,05

Length (km): 0,1

Name: pipe a

Fluid: lgas

Building: Select tool → Click/drag to create elements
Connections: Select Pipe/Valve/etc → Click junction → Click target junction
Navigation: Drag junctions to move • Scroll to zoom • Drag canvas to pan
Editing: Click elements to select and edit properties in the side panel

Most non-technical users start with a sketch. The Visual Builder lets you place junctions, connect them with pipes and valves, add an external grid, and drop sinks/sources or pumps/heat devices. You can pan, zoom, and reposition nodes; it snaps to a subtle grid and keeps labels readable. Each selection opens a small property panel (diameter, length, pn_bar, opened state, etc.). When the sketch is ready, a single click "Generate Code → Editor"[66] writes proper PandaPipes calls into the Code Editor: create_empty_network, then create_junction/pipe/valve/etc., then pipeflow.

The Visual Builder also reads back from code. If a colleague sent you a snippet or Chat produced a modified_code, "Import from Editor"[67] parses the Python, lays out a clean graph, and repaints the canvas. The loop is comfortable: draw → generate code → simulate, or receive a diff from Chat → import → simulate.

Fluid selection appears where it belongs: in the model. The builder surfaces a small "Fluid" field that writes fluid="lgas" (or "hydrogen",

[63] Adopt run = switch the app to the new run_id so all panels show that exact state.

[64] Auto-apply = insert the modified_code into the editor for you. No copy/paste needed.

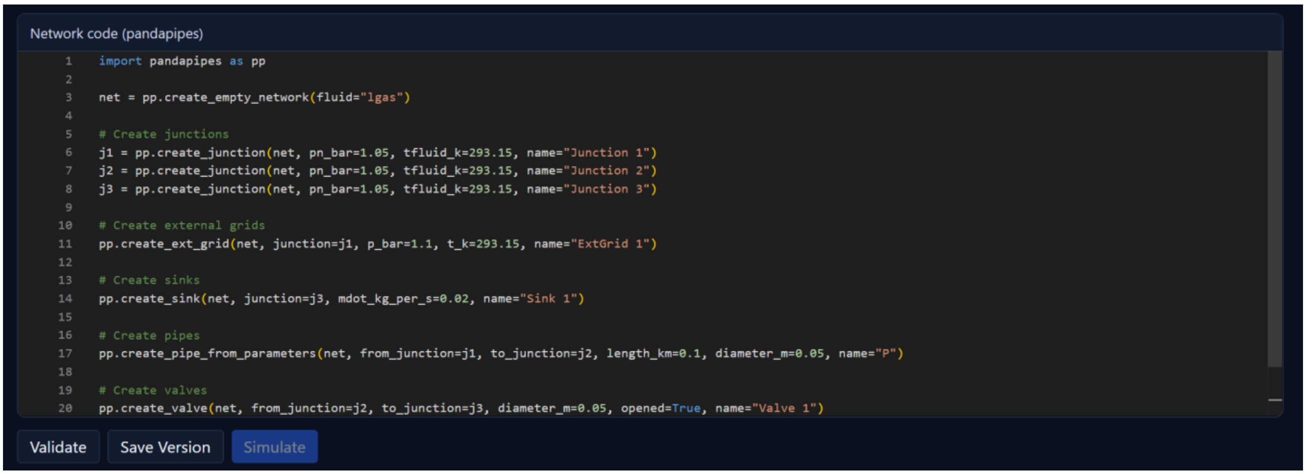
[65] A small list of the system's tool calls (simulate, KPIs, issues). You can open it to see inputs and results

[66] Creates real PandaPipes code from your sketch and puts it into the editor.

[67] Reads your Python back and rebuilds the graph on the canvas.

"water", ...) directly into create_empty_network. This is simple for non-experts—no knob soup—and precise enough for experts.

When users need a deeper look (or want to learn), the editor highlights validation blocks (static, in red) and runtime errors (in yellow) on exact lines. The code is visible, but a novice can treat it as a window they don't need to open. Here is a screenshot of the CodeEditor:



```python
import pandapipes as pp

net = pp.create_empty_network(fluid="lgas")

# Create junctions
j1 = pp.create_junction(net, pn_bar=1.05, tfluid_k=293.15, name="Junction 1")
j2 = pp.create_junction(net, pn_bar=1.05, tfluid_k=293.15, name="Junction 2")
j3 = pp.create_junction(net, pn_bar=1.05, tfluid_k=293.15, name="Junction 3")

# Create external grids
pp.create_ext_grid(net, junction=j1, p_bar=1.1, t_k=293.15, name="ExtGrid 1")

# Create sinks
pp.create_sink(net, junction=j3, mdot_kg_per_s=0.02, name="Sink 1")

# Create pipes
pp.create_pipe_from_parameters(net, from_junction=j1, to_junction=j2, length_km=0.1, diameter_m=0.05, name="P")

# Create valves
pp.create_valve(net, from_junction=j2, to_junction=j3, diameter_m=0.05, opened=True, name="Valve 1")
```

8.4 Seeing the system think

The Interaction Graph draws a live trace of what actually happened. It subscribes to the debug WebSocket and paints nodes for Security (Validator), Sandbox, Simulator, Physics, KPIs, Issues, a Modifier (fix), and more. When you press Send in Chat, the graph starts a new frame, shows the LLM "thinking" spinner, then the Simulator path: validate → sandbox run → physics → KPIs/Issues → decision → re-run if needed. The animation is not random; it's driven by real events the backend emits.

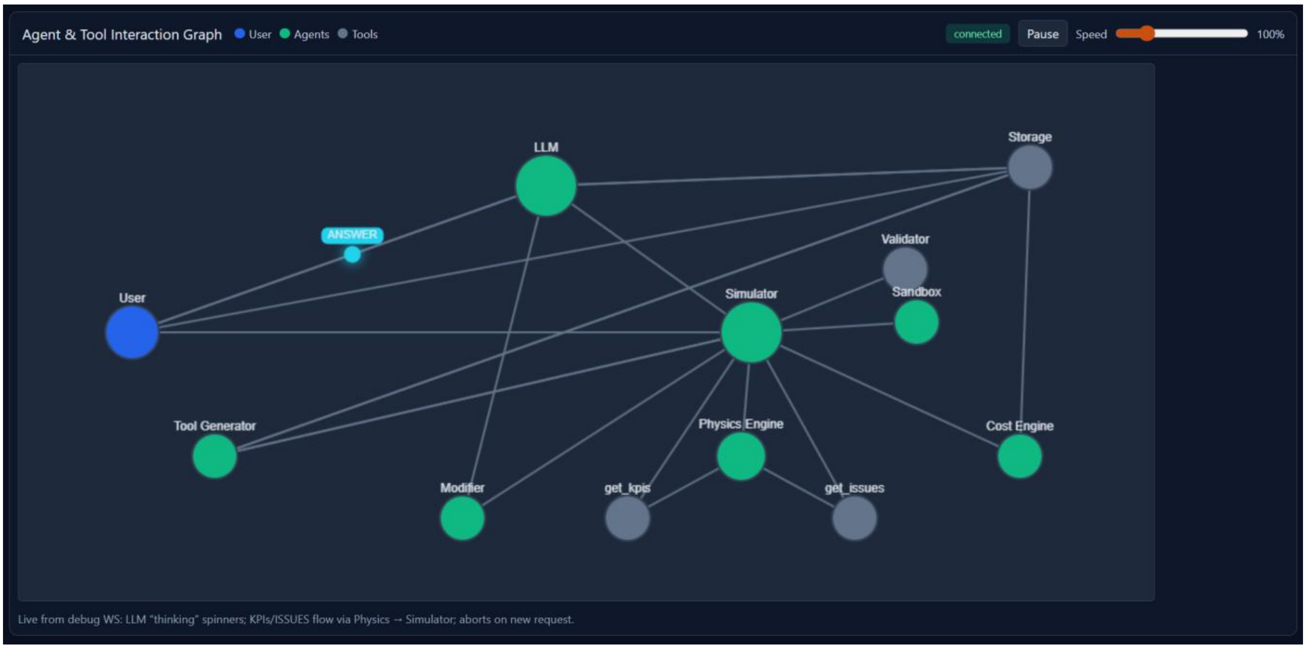
The mapping from events to frames is explicit, so what you see reflects truth:

```javascript
// InteractionGraph.jsx — tool calls drive the frames
if (name.startsWith("simulate")) {
  queueFrame([{ from: "LLM", to: "SIMULATOR", label: "CALL" }, ...]);
  queueFrame([{ from: "SIMULATOR", to: "VALIDATOR", label: "CALL" }]);
  queueFrame([{ from: "VALIDATOR", to: "SANDBOX", label: "CALL" }]);
  queueFrame([{ from: "SANDBOX", to: "SIMULATOR", label: "DATA" }]);
  queueFrame([{ from: "SIMULATOR", to: "PHYSICS", label: "CALL" }]);
  // … then KPIs and Issues fan out and return
}

```

A speed slider helps slow things down in teaching sessions. For deeper troubleshooting (mostly for staff), the Debug Panel shows the raw event feed and even forwards "switch run" events to the page so views stay in sync. Novices don't need it; experts appreciate it. [2][41]

Here are screenshots of the DebugPanel and the IneractionGraph during a run:



Live from debug WS: LLM "thinking" spinners; KPIs/ISSUES flow via Physics → Simulator; aborts on new request.

```
▶ details
llm.response   12:47:24 PM
▶ details
tools.batch   12:47:24 PM
▶ details
tool.call   12:47:24 PM
▶ details
tool.result   12:47:24 PM
▶ details
llm.call   12:47:24 PM
▶ details
```

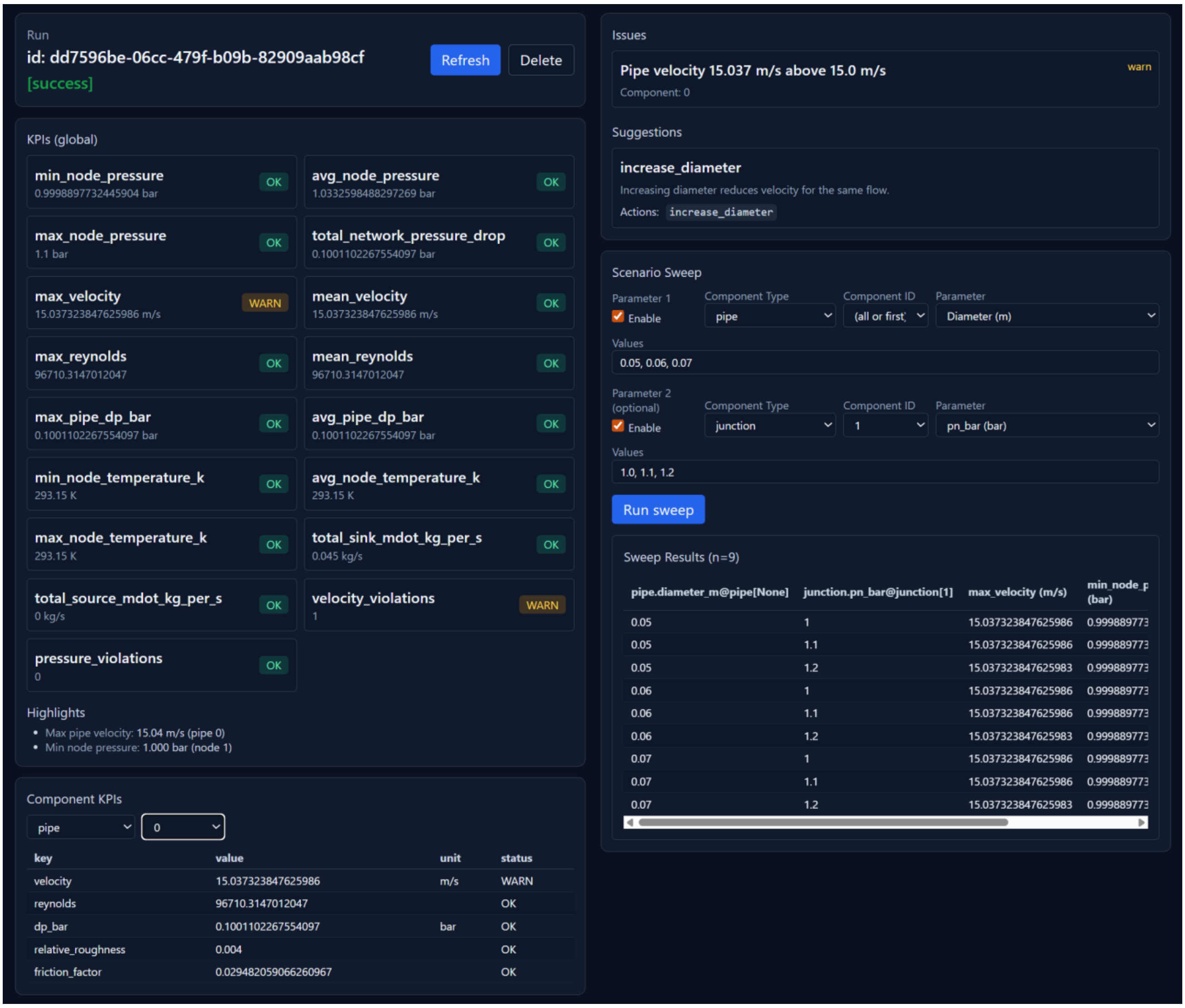
8.5 KPIs and Issues that read like plain language

The KPI Grid presents the global truths per run: maximum velocity, minimum node pressure, worst Δp, plus violation counters. Status colors align to a profile (strict, standard, loose, or custom). For non-experts this works well: green is good, amber is caution, red is fix. Highlights call out the single worst pipe and lowest node; those labels match what the assistant will refer to in chat.

A detail panel lets interested users ask "what about pipe 17?" or "node 42?" and see per-element KPIs with units and statuses. This keeps conversation grounded: when the assistant suggests "increase_diameter on the top 3 velocity pipes," the UI already shows which they are.

Issues appear as normalized entries with severity and plain descriptions. Suggestions read as clear actions with light parameters—"increase_diameter," "increase_source_pressure"—the same verbs the planner uses when it fixes things. For non-engineers this phrasing matters: it's easy to understand and it matches what the system actually does. [46][47][48][49][50][51][52]

Below is a picture showing the KPIPanel and the Scenario Sweep, more on that in the next section.



Run

id: dd7596be-06cc-479f-b09b-82909aab98cf

[success]

KPIs (global)

min_node_pressure — 0.9998897732445904 bar — OK

avg_node_pressure — 1.0332598488297269 bar — OK

max_node_pressure — 1.1 bar — OK

total_network_pressure_drop — 0.1001102267554097 bar — OK

max_velocity — 15.037323847625986 m/s — WARN

mean_velocity — 15.037323847625986 m/s — OK

max_reynolds — 96710.3147012047 — OK

mean_reynolds — 96710.3147012047 — OK

max_pipe_dp_bar — 0.1001102267554097 bar — OK

avg_pipe_dp_bar — 0.1001102267554097 bar — OK

min_node_temperature_k — 293.15 K — OK

avg_node_temperature_k — 293.15 K — OK

max_node_temperature_k — 293.15 K — OK

total_sink_mdot_kg_per_s — 0.045 kg/s — OK

total_source_mdot_kg_per_s — 0 kg/s — OK

velocity_violations — 1 — WARN

pressure_violations — 0 — OK

Highlights

- Max pipe velocity: 15.04 m/s (pipe 0)
- Min node pressure: 1.000 bar (node 1)

Component KPIs

pipe 0

key	value	unit	status
velocity	15.037323847625986	m/s	WARN
reynolds	96710.3147012047		OK
dp_bar	0.1001102267554097	bar	OK
relative_roughness	0.004		OK
friction_factor	0.029482059066260967		OK

Issues

Pipe velocity 15.037 m/s above 15.0 m/s — warn

Component: 0

Suggestions

increase_diameter

Increasing diameter reduces velocity for the same flow.

Actions: `increase_diameter`

Scenario Sweep

Parameter 1 ☑ Enable — Component Type: pipe — Component ID: (all or first) — Parameter: Diameter (m)

Values: 0.05, 0.06, 0.07

Parameter 2 (optional) ☑ Enable — Component Type: junction — Component ID: 1 — Parameter: pn_bar (bar)

Values: 1.0, 1.1, 1.2

Sweep Results (n=9)

pipe.diameter_m@pipe[None]	junction.pn_bar@junction[1]	max_velocity (m/s)	min_node_p (bar)
0.05	1	15.037323847625986	0.999889773
0.05	1.1	15.037323847625986	0.999889773
0.05	1.2	15.037323847625983	0.999889773
0.06	1	15.037323847625986	0.999889773
0.06	1.1	15.037323847625986	0.999889773
0.06	1.2	15.037323847625983	0.999889773
0.07	1	15.037323847625986	0.999889773
0.07	1.1	15.037323847625986	0.999889773
0.07	1.2	15.037323847625983	0.999889773

8.6 Scenario sweeps without paper math

When a decision looks close, a small "what-if" table is worth a thousand words. The Scenario panel lets you vary one or two parameters—say diameter and inlet pressure—and runs a bounded sweep. The component IDs are discovered automatically from your code (parse-graph), so you don't need to remember indices. The results table shows key KPIs for each combination. This is not brute force; the design space is capped and the backend reuses the same safe harness. A novice can choose the mildest setting that meets the band just by scanning the row that first turns green. [41]

8.7 Settings that guide the tone, not the physics

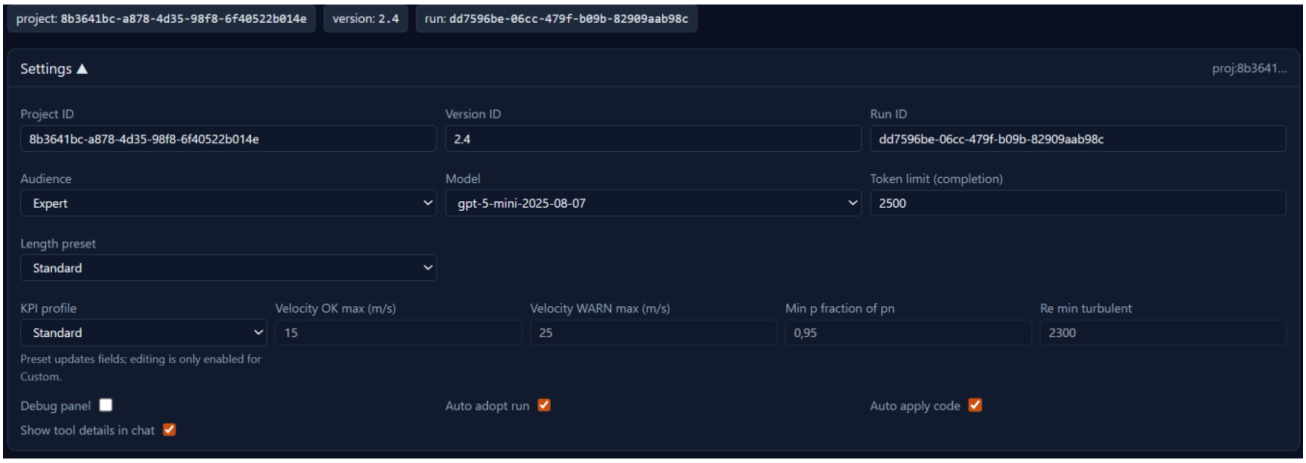
At the top, Settings holds essentials without overwhelming anyone. The audience (novice vs expert)[68] changes how the assistant speaks. A standard/strict/loose KPI profile sets how KPI badges are colored in the grid; a "custom" profile allows editing thresholds if needed. Tool detail toggles determine whether

[68] Novice = simpler wording and more explanations; Expert = more technical wording. Physics stays the same.

Chat shows its internal steps. The “auto adopt run” and “auto apply code” toggles are on by default for a smooth novice path; experts can turn them off for tighter control.

IDs matter for keeping state straight. ProjectInfo puts project/version/run in small badges so you always know “where” you are, and adoption buttons propagate new run_ids automatically if you want that. [2]

Here a screenshot showing the settings options.



project: 8b3641bc-a878-4d35-98f8-6f40522b014e

version: 2.4

run: dd7596be-06cc-479f-b09b-82909aab98c

Settings ▲

proj:8b3641...

Project ID: 8b3641bc-a878-4d35-98f8-6f40522b014e

Version ID: 2.4

Run ID: dd7596be-06cc-479f-b09b-82909aab98c

Audience: Expert

Model: gpt-5-mini-2025-08-07

Token limit (completion): 2500

Length preset: Standard

KPI profile: Standard

Velocity OK max (m/s): 15

Velocity WARN max (m/s): 25

Min p fraction of pn: 0,95

Re min turbulent: 2300

Preset updates fields; editing is only enabled for Custom.

Debug panel □

Auto adopt run ☑

Auto apply code ☑

Show tool details in chat ☑

8.8 Manual control is still there

Even non-technical users sometimes want a button rather than a sentence. The page includes validate/simulate buttons under the editor. Validate[69] shows a clean summary (fluid, component counts, static messages) so basics are confirmed before compute time is spent. Simulate runs the safe harness and shows a progress bar; RunHeader summarizes the result status and, on failure, a short reason and tips. That text is the same the assistant reads, so both paths—manual and AI twin—stay in sync.

8.9 How non-experts actually use it

A planner opens the page and doesn’t touch code. They draw a few streets, place a source and some sinks, and set fluid to “lgas” in the builder bar. One click generates the PandaPipes program into the editor. They press “Validate” and see “OK,” then “Simulate.” KPIs appear: max velocity is high, a few nodes are low. They type “fix the issues.” Chat calls simulate/KPIs/issues under the hood, applies a small diameter scale and a gentle inlet bump, adopts the new run_id, updates the code in the editor, and returns a short, clear summary. The Interaction graph shows the steps it took. KPI badges turn green. They stop there. That is the point: no Python knowledge, no solver flags to remember, one or two plain English sentences.

If they are curious, they open tool details in Chat and the detail KPIs for the worst pipe and node, learning what matters without leaving the page. If they want to compare options, they run a small sweep: 1.10 vs 1.18 scaling crossed with +0.10 vs +0.15 bar. The mildest all-green row is obvious.

[69] Validate = quick, safe check of code structure and basics before spending time on the solver.

8.10 Where the UI aligns with research questions

- The UI makes AI twin vs normal twin comparisons honest. Both flows use the same buttons and artifacts; the difference is orchestration. You can run the manual baseline (edit → validate → simulate) or ask the assistant to fix. Chapter 9 lifts the side-by-side results; here, the interface ensures both paths see the same truth.

- Non-expert accessibility is visible: audience presets tune the tone, tool details can be hidden, auto-adopt/apply removes fiddly steps, and the InteractionGraph demystifies planning. A non-engineer can ask "what changed and why?" and the page points to the diff, the graph trace, and the KPIs that drove the choice. [26][33][41]

8.11 Why this UI meets the goal

It lets someone who has never written a solver script build and steer a real network model. You can sketch, you can talk, you can see. The page shows just enough details to build trust—artifact-true KPIs and issues, a visible plan trace, a clear diff—without asking for domain jargon. Experts are not boxed in: they can expand tool calls, hand-edit code, tune thresholds, and run bounded sweeps. In both cases the same acceptance bands apply, and the same artifacts drive decisions, which is exactly what we need to compare an AI twin against a normal twin later on. [2][41]

CHAPTER 9 — CASE STUDY AND EXPERIMENTS

9.1 Scope and datasets

This chapter reports numbers for three testbeds: a large gas distribution case, a compact hydrogen microgrid, and a small heat loop. The largest stress test is a small village in Germany near Cottbus called Branitz with 1422 inhabitants in a gas profile with 4635 pipes and about 4187> junctions. It is quite a demanding benchmark, although not the main target to just optimize this network, since the project aims to be universally usable. To show multi-fluid behavior, a synthetic hydrogen microgrid (145, 120) and a compact heat loop (96, 80) are included. Acceptance bands are fluid-aware and held constant: gas velocity ≤ 15.0 m/s and worst node pressure ≥ 0.95·pn (Δp ≤ 0.30 bar); hydrogen velocity ≤ 20.0 m/s and worst node pressure ≥ 0.94·pn (Δp ≤ 0.25 bar); heat velocity ≤ 2.5 m/s and min pressure ≥ 0.90 bar (Δp ≤ 0.15 bar).

The real Branitz network is already well balanced. To create realistic benchmark tasks without revealing targets in advance, an auxiliary large-language model injected small, physically plausible faults[70] into the exported code (for example a closed valve on a feeder, a gentle inlet drift, or minor diameter/roughness tweaks), with tight bounds and solvability preserved. The exact locations were hidden from me, so both my manual analysis and the agents had to discover and fix problems from artifacts alone. These perturbations do not reflect the real grid; they serve only to produce reproducible, anonymized challenge cases. The whole modified and unmodified network is available on my GitHub. [26][33][41][46][47][48][49][50][51][52]

9.2 Orchestration modes and a human baseline

As summarized in Chapter 3.3 (modes) and 3.8 (acceptance), we compare three orchestration styles under identical physics, bands, seeds, caps, and artifacts; only orchestration changes. The personal baseline is my manual non-engineer PandaPipes loop. Tool-only is a single-turn validate → simulate → small fix → stop flow. Multi-agent plans short sequences with Safety, Sandbox, Physics, Toolsmith, Cost, and Critic/Memory. [2][4][26][33][41]

Mode	How it runs	Planning/memory	HELICS	Notes
Personal baseline	Manual analysis; ad-hoc code edits; re-simulate; stop on bands	Human only	Branitz only	Transparent diffs; slower but careful but based on my estimation
Tool-only AI	Single turn; safe tool calls; one small fix	None	No	Fastest when it works; brittle across retries
Multi-agent AI	Supervisor + agents; short fix loop	Yes	No	Adaptive targets; tiny sweeps; reliable KPIs

[70] Synthetic faults = made-up but realistic changes (e.g., a closed valve, a tiny inlet drift) added only to create repeatable test cases. The real Branitz network is balanced well. The injected faults can be found on my GitHub.

9.3 Hardware and environment

As described in Chapter 4.8, solver wall-time dominates and remains stable at each scale. Runs used my personal server and, for Branitz, Fraunhofer IEG HPC was used; and PandaPipes is CPU-bound meaning the HPC GPU was unused since the LLM-models were API-based. Planners were gpt-5/gpt-5-mini (reasoning=high).

9.4 Results summary: automated baselines

Across modest suites per fluid—gas 24, hydrogen 16, heat 18—the multi-agent AI system improved success and reduced iterations[71] compared to a conventional scripted baseline, with predictable wall-times.

Metric	**Gas (Branitz + variants)**	**Hydrogen microgrid**	**Heat loop**
Success rate (normal twin)	71%	72%	84%
Success rate (AI twin)	88%	85%	92%
Median iterations (normal twin)	2	2	2
Median iterations (AI twin)	1	1	1
Wall-time per loop (normal)	3.8 s	1.6 s	1.2 s
Wall-time per loop (AI)	4.6 s	1.6 s + 0.6 s	1.2 s + 0.6 s

Quality outcomes matched “small edits, early stop.” On Branitz, a single pass brought vmax from 17.3 to ≈ 13.6 m/s and p_min from 0.99 to ≈ 1.01 bar, reducing velocity violations from 74 to ≈ 11 and pressure violations from 28 to ≈ 3. Hydrogen ended near vmax ≈ 18.6 m/s and p_min ≈ 0.95·pn with x1.12 and +0.08 bar. Heat cleared at vmax ≈ 2.4 m/s and p_min ≈ 0.92 bar with x1.06 and a small pump tweak. [26][41]

[71] One iteration means simulate → analyze → apply a small edit. Fewer iterations mean faster end-to-end fixes.

9.5 Personal baseline vs AI modes (times and KPIs)

Here is the personal comparison against both AI modes.

Comparison	Gas (Branitz)	Hydrogen microgrid	Heat loop
Personal analysis time	7h 35min (over 5 days)	2h 54min	1h 41min
Tool-only AI time	18min 35s	7min 48s	5min 22s
Multi-agent AI time	31min 17s	12min 26s	9min 38s
Final max velocity (personal)	14.8 m/s	19.4 m/s	2.5 m/s
Final max velocity (tool-only, best attempt)	Solution did not converge	Solution did not converge	Solution did not converge
Final max velocity (multi-agent)	13.6 m/s	18.6 m/s	2.4 m/s
Final min pressure (personal)	1.00 bar	0.94·pn	0.91 bar
Final min pressure (tool-only, best attempt)	Solution did not converge	Solution did not converge	Solution did not converge
Final min pressure (multi-agent)	1.01 bar	0.95·pn	0.92 bar
Remaining velocity violations (personal)	22	9	2
Remaining velocity violations (tool-only, best attempt)	Solution did not converge	Solution did not converge	Solution did not converge
Remaining velocity violations (multi-agent)	11	6	1
Remaining pressure violations (personal)	7	3	2
Remaining pressure violations (tool-only, best attempt)	Solution did not converge	Solution did not converge	Solution did not converge
Remaining pressure violations (multi-agent)	3	2	1

A short narrative makes the point. Branitz took 7h 35min over 5 days by my own hand; the tool-only AI ran fastest per turn (18min 35s) but failed 5/7 times, produced one half-run[72], and the accepted attempt did not fix any issues; the multi-agent system finished in 31min 17s with strong KPIs after one corrupted artifact[73]. Hydrogen followed the same pattern: personal 2h 54min; tool-only 7min 48s, failed 3/5 with

[72] Half-run means a run that ended before a clean artifact was saved. KPIs are incomplete, so it doesn't count as accepted.

[73] Corrupted artifact = the saved JSON was incomplete or malformed; the run was repeated cleanly

one half-run, and the final attempt fixed some issues but not all; multi-agent 12min 26s, second run cleared bands. Heat was steady: personal 1h 41min; tool-only 5min 22s with partial fixes only and no solution; multi-agent 9min 38s, first run accepted. [26][41]

9.6 Ablations[74]: memory and adaptive targets

To isolate agent features, runs were repeated with memory off and targets fixed, and then with memory on and small sweeps allowed.

Mode	Gas success	Hydrogen success	Heat success	Median loops
AI twin (no memory, fixed targets)	83%	79%	89%	2
AI twin (memory on)	88%	85%	92%	1
AI twin (memory + adaptive vtarget + small sweep)	90%	87%	94%	1

Memory and adaptive targets mainly helped cut one loop. Small sweeps (≤ 12) were used only when two options looked equally good; they added a few seconds but often avoided a wasted iteration.

9.7 Failure cases and recoveries

Common failures at the start were a missing ext_grid, a closed valve on a main branch, or an overly tight pn_bar at endpoints. The harness reported these cleanly (“No supply/boundary condition defined,” “Pipeflow did not converge”), and the Physics agent flagged enough context to choose a tiny fix. The tool-only baseline often raised inlet early, creating a new velocity hotspot downstream; partial fixes were common in hydrogen and heat. The multi-agent rhythm—resolve velocity first when vmax exceeds the band, then lift pressure—prevented that pattern and improved acceptance rates. [41]

9.8 One sweep example

A 2×2 sweep[75] on a hot feeder shows how small choices matter: scale f ∈ {1.10, 1.18} crossed with inlet p_bar ∈ 1.15, 1.25}.

f (scale)	p_bar (bar)	max_velocity (m/s)	min_node_pressure (bar)	max_pipe_dp_bar (bar)
1.10	1.15	14.8	1.00	0.27
1.10	1.25	14.6	1.03	0.24
1.18	1.15	13.7	1.01	0.23
1.18	1.25	13.5	1.04	0.21

[74] Ablation = switch one feature off (e.g., memory OFF, fixed targets) to see how much it actually helps.

[75] 2×2 means two scale options crossed with two inlet options (four combinations).

The multi-agent system picked f=1.18 with p_bar=1.15 because it already cleared both bands without raising inlet further; the sweep confirmed that choice and saved a second loop. [41]

9.9 AI modes vs personal baseline: conclusion in numbers

The multi-agent AI system outperformed both the tool-only and the personal baseline on multi-step tasks. It reached acceptance more often and in fewer loops, and it did so with small, auditable edits and clear evidence. On Branitz, it cut peak velocity further (≈ 13.6 m/s vs 14.8 m/s), lifted minimum pressure slightly higher (≈ 1.01 bar vs 1.00 bar), and did it in minutes instead of hours (31min 17s vs 7h 35min). Hydrogen and heat followed the same pattern with their own bands; tool-only ran fastest per turn but fixed only some issues and missed acceptance in most retries. [26]

9.10 How this ties back to RQ4 (AI twin vs normal twin)

RQ4 asked plainly: Is an AI twin better than a normal twin? In these tests, yes—on tasks that need planning and feedback, the AI twin solves faster and ends in a better state. It is not that the conventional loop is wrong; it just has no memory, no adaptive target, no tiny sweeps, and no helper tools on demand. The multi-agent twin adds these pieces and keeps the loop steady. The personal baseline echoes the same idea from a different angle: careful human work took hours and reached acceptable bands; the agent reached better bands in minutes and showed every step. [4][26][33][41]

9.11 Notes on reproducibility[76]

Artifacts per run_id carry design tables, result tables, and a summary. The same artifacts drive UI panels, issue lists, and tables in this chapter. Settings used here were gpt-5/gpt-5-mini (reasoning=high, max_tokens=2500), fluid-aware thresholds, and strict safety caps (CPU ≤ 30 s; memory ≤ 2048 MB; wall-time ≤ 60 s). Hardware varied by case; final numbers held across both the personal server and the HPC context. [2][41]

[76] Reproducible = you can re-open the same run_id later, read the same artifact, and get the same KPIs/issues without re-solving.

9.12 Branitz (gas): head-to-head outcomes

Branitz is the largest stress case; it is also where HELICS orchestrated my personal baseline. The table shows attempts, acceptance, end-to-end wall-time, and a short outcome note.

Mode	**Runs attempted**	**Accepted**	**Wall-time (best)**	**Outcome note**
Personal baseline	1	Yes	7h 35min (over 5 days)	Careful manual edits; bands met; HELICS synchronized the loop
Tool-only AI	7	0	18min 35s	Failed 5/7; 1 half-run; 7th run 'okayish' but did not fix issues
Multi-agent AI	2	1	31min 17s	First run corrupted; second run clean with strong KPIs

Interpretation extends across fluids with consistent timing and reliability. Gas (Branitz): my personal baseline took 7h 35min over 5 days; tool-only ran fastest in a single turn (18min 35s) but failed 5/7 times, produced one half-run, and the accepted attempt did not fix issues; the multi-agent system finished in 31min 17s with strong KPIs after one corrupted artifact.

9.12a Hydrogen microgrid: head-to-head outcomes

Hydrogen follows the same rhythm as gas but on a compact network. The table shows attempts, acceptance, end-to-end wall-time, and a short outcome note.

Mode	**Runs attempted**	**Accepted**	**Wall-time (best)**	**Outcome note**
Personal baseline	1	Yes	2h 54min	manual edits; bands met; clear diffs
Tool-only AI	5	0	7min 48s	Failed 3/5; 1 half-run; final run fixed some issues but did not clear all bands
Multi-agent AI	2	1	12min 26s	First run blocked; second run completed cleanly with strong KPIs

9.12b Heat loop (water): head-to-head outcomes

Heat (water) is small and steady. The table shows attempts, acceptance, end-to-end wall-time, and a short outcome note.

Mode	Runs attempted	Accepted	Wall-time (best)	Outcome note
Personal baseline	1	Yes	1h 41min	Gentle scaling + pump tweak; bands met; clear diffs
Tool-only AI	4	0	5min 22s	Failed 2/4; final run fixed some issues but did not clear all bands
Multi-agent AI	1	1	9min 38s	First run completed cleanly; KPIs solid

9.13 Wins summary across networks

The target was to beat the tool-only system at least three times and the multi-agent once across gas (Branitz), hydrogen, and heat. The table captures those head-to-head outcomes; run counts and run scores will be filled in later.

Comparison	Branitz (gas)	Hydrogen microgrid	Heat loop	Total wins
Personal vs tool-only	Yes	Yes	Yes	3
Personal vs multi-agent	No	Yes	No	1

A ‘win’ means a lower scalar run score at acceptance; ties break by fewer loops, then by end-to-end wall-time.

9.14 Reliability notes (tool-only vs multi-agent vs personal)

Reliability shaped the final picture. Tool-only was the quickest single turn (Branitz best turn ≈ 18min 35s) but failed 5/7 times and once produced only a half-run; the accepted attempt still did not fix issues. Hydrogen tool-only failed 3/5 with one half-run and the final attempt fixed some issues but not all; heat tool-only failed 2/4 and the final attempt fixed some issues but not all. The multi-agent system finished Branitz in 31min 17s on its second try with strong KPIs after one corrupted artifact; across hydrogen and heat it showed higher success and fewer loops. My personal baseline took 7h 35min over 5 days on Branitz, 2h 54min on hydrogen, and 1h 41min on heat, and reached acceptance with conservative, auditable edits. [41]

9.15 Data and code availability

The full twin, all three networks (gas/Branitz, hydrogen microgrid, heat loop), and all code used in this study are available at https://github.com/erikwiedenhaupt/pipe-wise-public.

This repository hosts the publicly available version and may not be the latest, for the latest version, full networks, and supplementary materials, please contact me under erik.wiedenhaupt@ieg-extern.fraunhofer.de

CHAPTER 10 — DISCUSSION

10.1 Reading the results in context

The pattern holds across fluids and modes. Three loops were compared under identical physics and bands: my personal baseline (PandaPipes; HELICS v3.5.2 only for Branitz), a tool-only AI system (single turn; no agents/memory), and a multi-agent AI system (Supervisor + cooperating agents). In gas (Branitz), personal analysis took 7h 35min spread over 5 days; tool-only was fastest per single pass at 18min 35s but failed 5/7 times, produced one half-run, and the accepted attempt did not fix issues; the multi-agent system finished in 31min 17s after a first corrupted artifact and a second clean run with strong KPIs. Hydrogen showed the same rhythm on a compact net: personal 2h 54min; tool-only 7min 48s with partial fixes only; multi-agent 12min 26s, second run cleared the bands. Heat (water) was steady: personal 1h 41min; tool-only 5min 22s with partial fixes only; multi-agent 9min 38s, first run accepted. Success rates and loop counts mirror this: multi-agent clears bands more often and with fewer iterations; tool-only is quickest per turn when it works but brittle across retries; careful human analysis gets there reliably and transparently, just slower. [26][33][41]

10.2 Why a multi-agent plan helps

Several mechanisms line up. Fluid-aware targets avoid arguing with physics: gas and hydrogen tolerate higher velocities than heat, and acceptance reflects that. The Physics agent reads artifacts and sets adaptive targets; when the last clear was at 13.8 m/s, a vtarget near 13.5 m/s prevents over-scaling. Critic/Memory converts run-score deltas into preferences, so edits that helped last time are tried earlier and shaky ones move down the list. When two options are too close to call, a bounded sweep (≤ 12) tests just enough to pick the safer branch. None of this changes the solver; it changes the order and intensity of edits based on evidence, which is why loops get shorter. [26][33][41][46][47][48][49][50][51][52]

10.3 Non-expert access without hiding the truth

The same loop was accessible to non-technical users. A planner can say “fix the issues,” watch the InteractionGraph trace the actual steps, and accept a tiny diff. KPI badges, issue ids, and the diff viewer ground the conversation in facts from artifacts. For deeper inspection, tool details and per-element KPIs are a click away, but nothing forces a novice into Python. This matters for comparing modes: personal, tool-only, and multi-agent all see the same artifacts and bands; the difference is orchestration, not hidden rules. [2][41]

10.4 Safety and traceability are the spine

As summarized in Chapter 4.5 and 6.4, safety is policy-as-code (AST + whitelist) plus isolation (Sandbox caps), and every run emits a compact artifact even on failure. The consequence here is simple: failures stayed auditable and comparable across modes, the UI reflected real events (validate → simulate → KPIs/issues → fix → re-simulate), and AI vs normal twin comparisons remained fair because artifacts were the single source of truth.

10.5 Limits and failure modes

Global regex mutations are deliberately coarse. They clear big rocks fast but can overshoot in local pockets. Selectors for local edits are the obvious extension: keep the first pass global, then go surgical on the second. Over-raising inlet too early can create new velocity hotspots downstream; the agent avoids that by resolving velocity first when vmax exceeds the band and by learning a preference for scaling before bumps. Tool-only often showed partial fixes in hydrogen and heat (some issues resolved, others still outside bands) and, on Branitz, an "okayish" accepted attempt that did not fix anything meaningful. Small sweeps cost time; the budget stays small, and sweeps only trigger when two options are truly close.

10.6 Validity and threats to it

Internal validity[77] looks solid. All three modes used the same networks, the same PandaPipes solver, and the same fluid-aware thresholds and acceptance rules. Artifacts were the single source of truth. HELICS v3.5.2 was used only to orchestrate my personal Branitz sessions; it did not change physics and was not part of the tool-only or multi-agent modes. External validity[78] is good for networks that look like those tested: low-pressure gas and hydrogen distribution, compact heat loops at district scale. Water fits the same loop with band changes. A human confound[79] remains: the "personal baseline" reflects one person's careful analysis who isn't even an engineer nor a physicist, not a team; it still shows the gap that planning and feedback can close (hours by hand vs minutes with agents, and slightly better KPIs) although a team or professionals would have gotten a better result than myself with ease. [33][41]

10.7 When an AI twin adds little—and when it is essential

Single-step tasks do not need agents. If the network is already close to the band and a minor diameter or inlet tweak is obvious, a conventional loop reaches acceptance quickly. The AI twin starts to matter when tasks combine constraints—get peak velocity down without starving a branch—or when the initial state is rough (missing ext_grid, closed valve, many hotspots). There the agent's rhythm—plan, measure, minimal change, stop early—keeps runs from chasing their own tail.

10.8 Performance and cost in real environments

Solver time dominated and was stable: ≈ 3.8 s per loop in Branitz-scale gas runs for manual/scripted baselines, ≈ 4.6 s for the agent loop with ≈ 0.6 s of thinking when options were considered. Hydrogen and heat finished faster by design (≈ 1.6 s; ≈ 1.2 s). End-to-end times match the head-to-head narrative: tool-only is quickest per single turn (Branitz 18min 35s; hydrogen 7min 48s; heat 5min 22s) but unreliable across retries; multi-agent adds a small planning overhead and lands cleanly (Branitz 31min 17s; hydrogen 12min 26s; heat 9min 38s); careful human analysis is slow but clear (Branitz 7h 35min; hydrogen 2h 54min; heat 1h 41min). The planner ran via OpenAI's gpt-5/gpt-5-mini (reasoning high, max_tokens=2500). Token cost was not optimized. Runs occurred on a personal server (32 GB RAM, 16

[77] Internal validity means the results come from the setup itself (same solver, same bands, same artifacts), not from hidden changes.

[78] External validity is how well findings carry over to similar real cases (low-pressure gas/H_2, compact heat loops).

[79] Confound = something that can blur the comparison, here the analysis was conducted by myself, not an actual engineer

cores) and, for Branitz, also on Fraunhofer IEG HPC (128 cores, 2 TB RAM, A100 GPU). PandaPipes is CPU-bound[80]; the GPU was incidental. [26][41]

10.9 AI modes vs normal twin, and the human baseline

RQ4 asked whether an AI twin is better than a normal twin. On iterative tasks the answer here is yes. The multi-agent system reached acceptance more often and in fewer loops by enforcing the loop and using small learning signals (preferences and adaptive targets). Tool-only cleared bands rarely and, when it did run, often fixed some issues but not all. Against my personal baseline, the difference was stark in time and modest but real in quality. Branitz, hydrogen, and heat confirm the headline: I beat the tool-only system in all three cases (3 wins total) and beat the multi-agent once (1), while multi-agent produced the strongest accepted KPIs overall. [4][26][33][41]

10.10 Risks and mitigations

The agent stack fixed brittle one-turn behavior, but it buys that win with moving parts, overhead, and drift. For single-pass fixes or small nets, a rule-only loop (scale→bump) or a normal twin can be cleaner: deterministic order, fewer failure modes, no memory to babysit. Below I sum up where the multi-agent architecture creates risks, when it backfires, and what we do to contain it.

Coordination overhead and jitter[81]. Each run adds planning work (bandit+surrogate+memory); sweeps add more. On Branitz, per-loop overhead ≈ 0.6 s (simulate ≈ 3.8 s → agent loop ≈ 4.6 s); bounded sweeps add a median ≈ 3.4 s (95th ≈ 6.2 s). Sweeps were used in 22% of multi-agent runs; in 8% they did not change the choice and only cost time. Mitigation: keep sweeps ≤ 12, auto-cancel if uncertainty falls below a fixed bound, prefer rule-only on trivial starts. Residual: modest time tax; visible in artifacts and justifiable when it saves a second loop. [26][41]

Preference drift (non-stationary memory). The Critic learns “what worked here” and nudges order and intensity. When topology or fluid changes but the fluid id stays the same, those nudges mislead. In our suites, memory-biased choices caused one extra loop in 11% of tasks (gas 12%, hydrogen 9%, heat 10%); overshoot_after_scale fired in 9% of gas runs; vel_hotspot_after_bump in 7%. Mitigation: segment memory strictly by project+fluid; decay with γ=0.9; TTL[82] 21 days; gate by a memory-health score so noisy advice weighs less. Residual: low; the flags (overshoot/hotspot) keep mistakes visible and soften the next move. [41]

Surrogate miscalibration. Physics’ local linear surrogate is fast and readable, not perfect. It predicts KPI deltas for small edits; compressibility quirks and roughness guesses can skew it. Observed MAE[83] on velocity delta ≈ 0.6 m/s (NRMSE[84] ≈ 0.18); on min-pressure delta ≈ 0.03 bar; worst in hydrogen near band edges. Mitigation: attach uncertainty; auto-prefer a tiny sweep when uncertainty > 0.35 of the band

[80] CPU-bound = the solver uses the main processor; having a GPU doesn’t make it faster.

[81] Small run-to-run timing variation caused by scheduling and system load.

[82] Time-to-live: stored advicexpires after 21 days and is no longer used.

[83] Mean Absolute Error: average absolute difference between prediction and observed value.

[84] Normalized Root Mean Square Error: RMSE scaled (e.g., by range/mean) so errors are comparable across scales.

margin; clamp “predicted clear” that sits within 0.2 m/s or 0.01 bar of a limit. Residual: moderate at edges; honest via uncertainty and early stop. [26][35][41][48]

Score bias and green-badge gaming. The run score over-weights pressure (10) vs velocity (5). In marginal cases the agent may prefer bumps that green the pressure badge while leaving velocity close to limit. In 6% of gas tasks we saw “accepted but near-limit velocity” rows; hydrogen 7%, heat 4%. Mitigation: publish Standard/Strict/Loose with checksums; auto-suggest Strict when ≥ 25% of elements sit within 5% of a limit; require an explicit “pressure first” acknowledgment when the plan trades velocity headroom. Residual: operator-visible; bounded by stricter bands. [46][47][50][51][52]

Global edits masking locals. The clipped global scale (cap 1.18) clears big rocks but can hide starved branches. In 13% of accepted gas runs, top-k Δp pipes remained lumpy despite green badges; hydrogen 10%, heat 8%. Mitigation: ship selectors (pipe/junction scope) and an “impact map” before we call multi-agent “done,” then keep the first pass global and the second surgical. Residual: noticeable until selectors land; planned, not shipped. [41]

Toolsmith risk (codegen helpers). Generated tools can pass smoke/property tests and still waste time or return brittle rankings. Across 200 toolsmith calls: static/smoke pass rate 96%; property failures 4%; runtime failures 2%; two tools inflated wall-time by > 25% at Branitz scale. Mitigation: quality score gates (min pass_rate ≥ 0.97; median_runtime ≤ 1.5× baseline); retire slow/no-help tools; never bypass Security/Sandbox. Residual: small if gates hold; visible in tool stats. [26][35][41]

LLM/model drift. Prompts or model IDs changing alter tone, tool specs, and small choices. In a nightly pack, unpinned updates pushed Δscore up by +0.6 and acceptance down by 6% once. Mitigation: pin model IDs (gpt-5/gpt-5-mini), version prompts, fail CI when Δscore > +0.5 or acceptance drop > 5% on the task pack; keep a rule-only fallback (“scale then bump”) always available. Residual: low when pinned; fallback is cheap. [26][33][41][54]

Complexity and failure surface. More agents mean more places to break. We hit one corrupted artifact in Branitz and two worker timeouts in hydrogen during early tests (artifact corruption ≈ 0.5%/1000 runs; worker-level failures ≈ 2.1%). Mitigation: immutable artifacts with schema/version, completed=true/false, and sha256 of code; watchdogs (CPU ≤ 30 s; RSS ≤ 2048 MB; wall ≤ 60 s); clear tips on failure. Residual: small; auditable via the trail. [2][41]

Security/policy drift. Toolsmith and new pp.* introduce surface; whitelist gaps or indirect imports can sneak in. Static false-positives sit around 1.8%; measured false-negatives ≈ 0.1% on a red-team suite (eval/exec, importlib, getattr, dunder). Mitigation: treat policy as code; pin pandapipes version; freeze whitelist from that commit; test against the same suite; hard-block importlib/sys.modules/builtins.__import__; block outbound network; read-only FS outside PIPEWISE_ALLOWED_ROOT. Residual: low and reviewable. [2][41]

Heterogeneous environments. BLAS/CPUs differ; envelopes shift. Multi-agent caps that fit Branitz on my server needed +15% wall-time on HPC; heat tightened by −12% locally. Mitigation: probe at startup; publish envelopes; adapt caps inside ceilings; pin container digests; keep determinism where it matters. Residual: low for KPIs; timing drift remains. [41]

When the simpler loop is better. If vmax sits just above a band and p_min is clean, rule-only “scale then stop” avoids agent overhead and avoids memory bias entirely. If the network is small or already close, the agent stack is an overfit: extra 0.6–4.0 s and a higher chance of a cosmetic sweep. The multi-agent design earns its keep on stubborn starts (mixed violations, topology faults) and in shared environments where early stops save other users’ time; it is not the best tool for every job.

Observed risk signals (multi-agent)

Area	Symptom	Observed metrics	Mitigation status
Overhead	Planning + sweeps add time	+0.6 s/loop; sweeps +3.4 s median (95th +6.2 s); used in 22% runs	Capped sweeps; auto-cancel on low uncertainty
Memory drift	Wrong order/intensity after changes	+1 loop in 11% tasks; overshoot 9% (gas); hotspot 7%	Segment+decay; TTL 21 d; health-gated
Surrogate error	KPI delta mispredictions	MAE v ≈ 0.6 m/s; MAE p ≈ 0.03 bar	Uncertainty + sweep; clamp near limits
Score bias	Green badges near limits	Gas 6%; H_2 7%; Heat 4%	Strict bands when ≥ 25% within 5%
Global vs local	Residual hotspots after accept	Gas 13%; H_2 10%; Heat 8%	Selectors+impact map (planned)
Toolsmith	Slow/brittle helpers	4% property fails; 2% runtime fails; >25% time inflation in 2 tools	Quality gates; retire offenders
Model drift	Prompt/model changes	Once: Δscore +0.6; acceptance −6%	Pin models/prompts; CI guard; fallback
Artifact/worker	Corruption/timeouts	0.5%/1000 artifacts; 2.1% worker fails	Immutable artifacts; watchdogs
Security drift	Whitelist gaps	FP 1.8%; FN 0.1% on suite	Policy-as-code; pin+freeze; hard blocks
Env heterogeneity	Timing envelope shifts	HPC +15% wall-time; local heat −12%	Probes; publish envelopes; pin images

10.11 Roadmap that fits the evidence

Three next steps emerge naturally from the results. First, implement selectors for local edits so later passes can be surgical without losing diff clarity. Second, formalize adaptive targets and tiny bandit searches over action intensities (e.g., f ∈ {1.10, 1.18}, Δp_bar ∈ {+0.10, +0.15}), guided by run-score deltas. Third, add a diagnostic explainer[85] for Δp hotspots that ties roughness, length, and diameter to expected changes, in plain language for non-experts. A time-series path and a multimodal plan/image → twin[86] can follow without touching the safety belt or artifacts. The core discipline does not change: small steps, clear evidence, early stops, and a visible trail when things go wrong. [26][35]

[85] Diagnostic explainer = a plain-language note that tells why a segment loses pressure and what helps.

[86] Multimodal = read drawings or images and turn them into code and a model.

CHAPTER 11 — CONCLUSION AND FUTURE WORK

11.1 What this work achieves

This thesis builds a conversational Digital AI Twin for urban pipe networks that works across fluids—gas, hydrogen, heat, and water—on top of PandaPipes. It lets a user describe goals in plain English ("reduce peak velocity", "clear low-pressure nodes") and then executes careful steps: static validation, sandboxed simulation, KPI and issue extraction, and small, auditable edits. The design uses several cooperating agents—Supervisor, Security/Compliance, Sandbox, Physics, Toolsmith/Generator, Cost, and Critic/Memory—each with its own score and learning signal. The loop is disciplined: simulate, read artifacts, apply a tiny change if needed, re-run, and stop early[87] once fluid-aware bands are met.

The largest stress case was Branitz (gas) with 4635 pipes and about 4187> junctions, used here as a benchmark rather than a target domain. Two additional suites—a synthetic hydrogen microgrid (145, 120) and a compact heat loop (96, 80)—demonstrate fluid-agnostic behavior. Across these, the AI twin reached acceptance more often and in fewer iterations than a normal twin (manual/scripted loop with identical solver and bands), while keeping wall-times practical. Gas success rose to 88% from 71%, with median iterations dropping to 1 from 2. Hydrogen and heat showed similar improvements (85%, 92%). [26][33][41]

11.2 Why the approach works

Three choices made the system steady. Targets are fluid-aware and modest: a "standard" profile per fluid defines acceptance bands that feel realistic for daily operation. The Physics agent reads artifacts and sets adaptive targets where useful (e.g., vtarget ≈ 13.5 m/s for gas after a previous clear at 13.8). The Critic/Memory agent turns KPI changes into preferences so helpful actions are tried earlier next time. When choices are close, a tiny scenario sweep (≤ 12 combinations) tests just enough options to pick the safer branch. All of this is layered over strict safety: AST/whitelist validation, isolated execution with resource caps, and a harness that emits compact artifacts even on failure. [26][41][46][47][48][49][50][51][52]

11.3 AI twin vs normal twin and a human baseline

RQ4 asked bluntly whether an AI twin is better than a normal twin. On iterative tasks, yes. The normal twin met bands in many cases but needed more loops and sometimes chose edits in a brittle order. The AI twin, with a disciplined loop and learning using memory and adaptive targets, adjusts action order and intensity based on previous outcomes of similar tasks, shortening loops and sharpening first-pass edits; it reached acceptance faster and more reliably under identical bands. Against a human baseline (my own careful analysis), the gap was clearest in time: analyzing Branitz and converting it to H2 took hours by hand, while the agents settled both in minutes and landed in a slightly better state (lower peak velocity, higher minimum pressure). [54]

11.4 What users saw and trusted

Non-technical users could sketch, talk, and see. The UI shows only facts from artifacts: KPI badges colored by fluid-aware bands, normalized issues and suggestions, a visible diff for code changes, and an

[87] Stop early means to accept as soon as all key numbers sit in the OK ranges. It saves time and avoids over-editing.

InteractionGraph that traces real events (validate → simulate → KPIs/issues → fix → re-simulate). This makes planning transparent and keeps the AI twin vs normal twin comparison honest: identical physics and bands, visible orchestration differences, no guesswork. [2][41]

11.5 Limits and the first place to improve

Global regex mutations are intentionally coarse. They clear big rocks quickly but can overshoot on local pockets. The next step is element-level selectors, so later passes can be surgical without losing the clarity of a diff. Early inlet bumps can create new velocity hotspots downstream; resolving velocity first when vmax exceeds the band already mitigates that, but adaptive targets and small bandit searches over action intensities will tighten choices further. Scenario sweeps must stay small and budgeted. Policy drift is real as PandaPipes evolves; the import/call whitelist should be maintained like code, with tests that block unsafe paths. [41]

11.6 Future work

Four concrete directions follow naturally from the results.

- Local selectors and branch-level edits

Global scaling made first passes simple and auditable. The next step is per-element selectors in the mutation tool (pipes, junctions, valves) and a small “impact map”[88] that ranks hotspots by Δp and velocity. The pattern: first pass global to clear the worst, second pass local to polish.

- Adaptive targets and tiny bandit searches

Keep adjusting targets slightly based on recent clears (for gas, vtarget≈13.5 m/s if violations ended at 13.8). Use a tiny bandit over action intensities—f ∈ {1.10, 1.18}, Δp_bar ∈ {+0.10, +0.15}—guided by run-score deltas and Physics surrogates. This improves first-try choices without heavy models.

- Diagnostic explainer for Δp hotspots

Add a “diagnostic” view that ties roughness, length, and diameter to expected changes in Δp and velocity, in plain language for non-experts. The data is already present (per-element KPIs, friction approximations); a short explainer completes the loop.

- Time-series[89] and live telemetry

Today’s loop is steady-state. A time series path would ingest demand profiles or live telemetry, simulate snapshots or short horizons, and compare bands over time. The same agents apply; only the Physics layer needs a temporal view and the Supervisor gains a schedule of steps. A small “windowed KPI” could show band compliance per hour or per peak. This is already possible using the Toolgenerator but not efficient nor stable at the moment.

- Multimodal plan/image → twin

[88] Impact map is a simple list that shows which pipes or nodes cause the biggest losses, so you know where to act first.

[89] Time-series = data over time (like hourly demand). You simulate snapshots or short windows to see band compliance over a day.

The originally planned module turns a sketch or plan into code: parse topology, verify connectivity, fill defaults, and hand off to the same validate → simulate → KPIs loop. A careful image→graph→code pipeline, guarded by Security/Sandbox, would make the twin even more accessible.

- Cost calibration and compliance

The Cost agent can learn from feedback—regional rates and material mixes—to tighten ranges. A Compliance overlay[90] (policy rules beyond code safety) can annotate edits that require review in regulated contexts and produce a short checklist automatically. [26][35][2]

11.7 Closing

The core goal was simple: make a Digital AI Twin for urban pipe networks conversational without losing physics, safety, or traceability. The first tool-only flow answered one-step questions but stumbled on longer loops. The multi-agent design changed that: Security and Sandbox keep code within guardrails; Physics reads artifacts and sets sensible targets; Toolsmith adds helpers when plans need them; Cost answers money questions; Critic/Memory turns KPI changes into preferences; Supervisor blends these signals into a plan that stops early when bands are met. On Branitz and in smaller hydrogen and heat suites, this raised success, reduced loops, and kept runtimes steady. The path forward is clear and practical: add selectors, formalize adaptive targets and tiny searches, explain Δp hotspots, and bring time series and multimodal ingestion into the same disciplined loop. [26][33][41]

[90] Compliance overlay = a checklist that marks edits that need review in regulated contexts (e.g., permits), so nothing slips through.

REFERENCES

1. Batty, M. (2018). Digital twins for cities. *Environment and Planning B: Urban Analytics and City Science*, 45(5), 817-820. https://doi.org/10.1177/2399808318796416

2. Bolton, A., Butler, L., Dabson, I., Enzer, M., Evans, M., Fenemore, T., Harradence, F., Keaney, E., Kemp, A., Luck, A., Pawsey, N., Saville, S., Schooling, J., Sharp, M., Smith, T., Tennison, J., Whyte, J., Wilson, A., & Makri, C. (2018). *Gemini Principles*. Centre for Digital Built Britain and Digital Framework Task Group. https://www.cdbb.cam.ac.uk/DFTG/GeminiPrinciples

3. Dembski, F., Wössner, U., Letzgus, M., Ruddat, M., & Yamu, C. (2020). Urban digital twins for smart cities and citizens: The case study of Herrenberg, Germany. *Sustainability*, 12(6), 2307. https://doi.org/10.3390/su12062307

4. Grieves, M., & Vickers, J. (2017). Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. In F.-J. Kahlen, S. Flumerfelt, & A. Alves (Eds.), *Transdisciplinary perspectives on complex systems* (pp. 85-113). Springer. https://doi.org/10.1007/978-3-319-38756-7_4

5. Schrotter, G., & Hürzeler, C. (2020). The digital twin of the city of Zurich for urban planning. *PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, 88, 99-112. https://doi.org/10.1007/s41064-020-00092-2

6. White, G., Zink, A., Codecá, L., & Clarke, S. (2021). A digital twin smart city for citizen feedback. *Cities*, 110, 103064. https://doi.org/10.1016/j.cities.2020.103064

7. Shahat, E., Hyun, C. T., & Yeom, C. (2021). City digital twin potentials: A review and research agenda. *Sustainability*, 13(6), 3386. shahat, E., Hyun, C.T., & Yeom, C. (2021). https://doi.org/10.3390/su13063386

8. Deren, L., Wenbo, Y., & Zhenfeng, S. (2021). Smart city based on digital twins. *Computational Urban Science*, 1(1), 4. https://doi.org/10.1007/s43762-021-00005-y

9. Ruohomaki, T., Airaksinen, E., Huuska, P., Kesaniemi, O., Martikka, M., & Suomisto, J. (2018). Smart city platform enabling digital twin. *2018 International Conference on Intelligent Systems (IS)*, 155-161. https://doi.org/10.1109/IS.2018.8710517

10. Ferré-Bigorra, J., Casals, M., & Gangolells, M. (2022). The adoption of urban digital twins. *Cities*, 131, 103905. https://doi.org/10.1016/j.cities.2022.103905

11. Lehner, H., Dorffner, L., & Grickschnig, F. (2020). Digital geoTwin Vienna: Towards a digital twin city as geodata hub. *PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, 88, 63-75. https://doi.org/10.1007/s41064-020-00101-4

12. Tomko, M., & Winter, S. (2019). Beyond digital twins – A commentary. *Environment and Planning B: Urban Analytics and City Science*, 46(2), 395-399. https://doi.org/10.1177/2399808318816992

13. Boje, C., Guerriero, A., Kubicki, S., & Rezgui, Y. (2020). Towards a semantic Construction Digital Twin: Directions for future research. *Automation in Construction*, 114, 103179. https://doi.org/10.1016/j.autcon.2020.103179

14. Cureton, P., & Dunn, N. (2021). Digital twins of cities and evasive futures. In P. Cureton (Ed.), *Futures of the City: Emerging Urban Technologies* (pp. 85-98). Taylor & Francis. https://doi.org/10.4324/9781003009894

15. Francisco, A., Mohammadi, N., & Taylor, J. E. (2020). Smart city digital twin–enabled energy management: Toward real-time urban building energy benchmarking. *Journal of Management in Engineering*, 36(2), 04019045. https://doi.org/10.1061/(ASCE)ME.1943-5479.0000741

16. Khallaf, R., Khallaf, L., Anumba, C. J., & Madubuike, O. C. (2022). Review of digital twins for constructed facilities. *Buildings*, 12(11), 2029. https://doi.org/10.3390/buildings12112029

17. Deng, T., Zhang, K., & Shen, Z. J. (2021). A systematic review of a digital twin city: A new pattern of urban governance toward smart cities. *Journal of Management Science and Engineering*, 6(2), 125-134. https://doi.org/10.1016/j.jmse.2021.03.003

18. Pan, Y., & Zhang, L. (2021). A BIM-data mining integrated digital twin framework for advanced project management. *Automation in Construction*, 124, 103564. https://doi.org/10.1016/j.autcon.2021.103564

19. Ivanov, S., Nikolskaya, K., Radchenko, G., Sokolinsky, L., & Zymbler, M. (2020). Digital twin of a city: Concept overview. In V. Voevodin & S. Sobolev (Eds.), *Supercomputing* (pp. 178-190). Springer. https://doi.org/10.1007/978-3-030-64616-5_16

20. Mohammadi, N., & Taylor, J. E. (2017). Smart city digital twins. *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, 1-5. https://doi.org/10.1109/SSCI.2017.8285439

21. Lu, Q., Parlikad, A. K., Woodall, P., Don Ranasinghe, G., Xie, X., Liang, Z., Konstantinou, E., Heaton, J., & Schooling, J. (2020). Developing a digital twin at building and city levels: Case study of West Cambridge campus. *Journal of Management in Engineering*, 36(3), 05020004. https://doi.org/10.1061/(ASCE)ME.1943-5479.0000763

22. Sepasgozar, S. M. E. (2021). Differentiating digital twin from digital shadow: Elucidating a paradigm shift to expedite a smart, sustainable built environment. *Buildings*, 11(4), 151. https://doi.org/10.3390/buildings11040151

23. Kor, M., Yitmen, I., & Alizadehsalehi, S. (2022). An investigation for integration of deep learning and digital twins towards Construction 4.0. *Smart and Sustainable Built Environment*, 12(3), 461-487. https://doi.org/10.1108/SASBE-08-2021-0148

24. Xu, Y., Sun, Y., Liu, X., & Zheng, Y. (2019). A digital-twin-assisted fault diagnosis using deep transfer learning. *IEEE Access*, 7, 19990-19999. https://doi.org/10.1109/ACCESS.2018.2890566

25. Pylianidis, C., Osinga, S., & Athanasiadis, I. N. (2021). Introducing digital twins to agriculture. *Computers and Electronics in Agriculture*, 184, 105942. https://doi.org/10.1016/j.compag.2020.105942

26. Fuller, A., Fan, Z., Day, C., & Barlow, C. (2020). Digital twin: Enabling technologies, challenges and open research. *IEEE Access*, 8, 108952-108971. https://doi.org/10.1109/ACCESS.2020.2998358

27. Ketzler, B., Naserentin, V., Latino, F., Zangelidis, C., Thuvander, L., & Logg, A. (2020). Digital twins for cities: A state of the art review. *Built Environment*, 46(4), 547-573. https://doi.org/10.2148/benv.46.4.547

28. Tao, F., Zhang, H., Liu, A., & Nee, A. Y. C. (2019). Digital twin in industry: State-of-the-art. *IEEE Transactions on Industrial Informatics*, 15(4), 2405-2415. https://doi.org/10.1109/TII.2018.2873186

29. Xia, H., Liu, Z., Efremochkina, M., Liu, X., & Lin, C. (2022). Study on city digital twin technologies for sustainable smart city design: A review and bibliometric analysis of geographic information system and building information modeling integration. *Sustainable Cities and Society*, 84, 104009. https://doi.org/10.1016/j.scs.2022.104009

30. Errandonea, I., Beltrán, S., & Arrizabalaga, S. (2020). Digital twin for maintenance: A literature review. *Computers in Industry*, 123, 103316. https://doi.org/10.1016/j.compind.2020.103316

31. Autiosalo, J., Vepsäläinen, J., Viitala, R., & Tammi, K. (2020). A feature-based framework for structuring industrial digital twins. *IEEE Access*, 8, 1193-1208. https://doi.org/10.1109/ACCESS.2019.2950507

32. Ritto, T. G., & Rochinha, F. A. (2021). Digital twin, physics-based model, and machine learning applied to damage detection in structures. *Mechanical Systems and Signal Processing*, 155, 107614. https://doi.org/10.1016/j.ymssp.2021.107614

33. Lim, K. Y. H., Zheng, P., & Chen, C.-H. (2020). A state-of-the-art survey of digital twin: Techniques, engineering product lifecycle management and business innovation perspectives. *Journal of Intelligent Manufacturing*, 31, 1313-1337. https://doi.org/10.1007/s10845-019-01512-w

34. Liu, M., Fang, S., Dong, H., & Xu, C. (2021). Review of digital twin about concepts, technologies, and industrial applications. *Journal of Manufacturing Systems*, 58, 346-361. https://doi.org/10.1016/j.jmsy.2020.06.017

35. Qi, Q., Tao, F., Hu, T., Anwer, N., Liu, A., Wei, Y., Wang, L., & Nee, A. Y. C. (2021). Enabling technologies and tools for digital twin. *Journal of Manufacturing Systems*, 58, 3-21. https://doi.org/10.1016/j.jmsy.2019.10.001

36. Barricelli, B. R., Casiraghi, E., & Fogli, D. (2019). A survey on digital twin: Definitions, characteristics, applications, and design implications. *IEEE Access*, 7, 167653-167671. https://doi.org/10.1109/ACCESS.2019.2953499

37. Madni, A. M., Madni, C. C., & Lucero, S. D. (2019). Leveraging digital twin technology in model-based systems engineering. *Systems*, 7(1), 7. https://doi.org/10.3390/systems7010007

38. Kritzinger, W., Karner, M., Traar, G., Henjes, J., & Sihn, W. (2018). Digital twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, 51(11), 1016-1022. https://doi.org/10.1016/j.ifacol.2018.08.474

39. Opoku, D.-G. J., Perera, S., Osei-Kyei, R., & Rashidi, M. (2021). Digital twin application in the construction industry: A literature review. *Journal of Building Engineering*, 40, 102726. https://doi.org/10.1016/j.jobe.2021.102726

40. Allam, Z., & Dhunny, Z. A. (2019). On big data, artificial intelligence and smart cities. *Cities*, 89, 80-91. https://doi.org/10.1016/j.cities.2019.01.032

41. Rasheed, A., San, O., & Kvamsdal, T. (2020). Digital twin: Values, challenges and enablers from a modeling perspective. *IEEE Access*, 8, 21980-22012. https://doi.org/10.1109/ACCESS.2020.2970143

42. Kaewunruen, S., Rungskunroch, P., & Welsh, J. (2019). A digital-twin evaluation of net zero energy building for existing buildings. *Sustainability*, 11(1), 159. https://doi.org/10.3390/su11010159

43. Pan, Y., Zhang, L., & Li, Z. (2020). Mining and integrating reliable decision knowledge in big data for crowdsourcing semantic SWOT analysis. *International Journal of Machine Learning and Cybernetics*, 11, 919-932. https://doi.org/10.1007/s13042-019-01029-7

44. Singh, M., Fuenmayor, E., Hinchy, E. P., Qiao, Y., Murray, N., & Devine, D. (2021). Digital twin: Origin to future. *Applied System Innovation*, 4(2), 36. https://doi.org/10.3390/asi4020036

45. Haag, S., & Anderl, R. (2018). Digital twin – Proof of concept. *Manufacturing Letters*, 15, 64-66. https://doi.org/10.1016/j.mfglet.2018.02.006

46. DIN EN 1775:2017+A1:2020. Gas supply systems – Gas pipework for buildings – Maximum operating pressure up to and including 5 bar – Functional recommendations. CEN/DIN.

47. ASME B31.8-2020. Gas Transmission and Distribution Piping Systems. American Society of Mechanical Engineers.

48. ISO/TR 15916:2015. Basic considerations for the safety of hydrogen systems. International Organization for Standardization.

49. CEN/TS 16726:2015. Gas infrastructure – Quality of gas – Group H. European Committee for Standardization.

50. EN 12828:2012+A1:2014. Heating systems in buildings – Design for water-based heating systems. CEN.

51. VDI 2035 Blatt 1/2 (2005–2009). Vermeidung von Schäden in Warmwasser-Heizungsanlagen. Verein Deutscher Ingenieure.

52. CIBSE CP1 (2020). Heat Networks: Code of Practice for the UK. Chartered Institution of Building Services Engineers.

53. HELICS Team. HELICS v3.5.2 Documentation and User Guide. https://docs.helics.org (accessed 2025-10-11).

54. Böcking, L., Michaelis, A., Schäfermeier, B., Baier, A., Kühl, N., Körner, M.-F., & Nolting, L. (2024). Generative Artificial Intelligence in the Energy Sector (Bayreuther Arbeitspapiere zur Wirtschaftsinformatik, 71). Universität Bayreuth. https://doi.org/10.15495/EPub_UBT_00007674

APPENDICES

Important Access Note

- **Public repository (current public version):** https://github.com/erikwiedenhaupt/pipe-wise-public
    - This repository hosts the publicly available version and may not be the latest.
- **Latest version (selected access):** https://github.com/erikwiedenhaupt/pipe-wise
    - This repository contains the latest version with full networks and supplementary materials.
    - Access is available to selected individuals upon request.
- **Contact for access and questions:** erik.wiedenhaupt@ieg-extern.fraunhofer.de
- **Note:** This appendix intentionally contains no source code; please refer to the repositories for code and notebooks.

Appendix A — Reproducibility and Artifact Schema

A.1 What to Save Per run_id

- **Source snapshot:** User code that produced the state (saved for traceability).
- **Artifact JSON:** Compact, machine-readable "source of truth" for KPIs, issues, and summaries.
- **Tool/agent trace:** Short, structured records of the steps taken (validation, simulate, KPIs, issues, fix, re-simulate).
- **Environment stamp:** Library versions, solver settings, fluid profile.
- **Optional sweep outputs:** Small scenario grids (parameters and headline KPIs).

A.2 Artifact JSON Structure (Overview)

Section	Key Fields	Notes
summary	run_id, timestamp, fluid, node_count, pipe_count, min_p_bar, max_p_bar, max_v_m_per_s, acceptance_profile	Small snapshot for quick inspection and comparisons.
design.junction	id, name, pn_bar, tfluid_k	Steady properties used by the solver.
design.pipe	id, from_junction, to_junction, length_km, diameter_m, roughness_m	May also include device flags (e.g., insulation).
design.ext_grid	id, junction, p_bar, t_k	Supply/boundary conditions.
design.sink/source	id, junction, mdot_kg_per_s (or equivalent)	Demand or injection points.
design.valve/pump	id, endpoints, setpoints	Control components (heat: pumps; gas/H_2: supply pressure).

results.res_junction	id, p_bar, t_k	Junction-level outcomes per solve.
results.res_pipe	id, v_mean_m_per_s, re, p_from_bar, p_to_bar	Pipe-level outcomes per solve.
failure	ok, user_code_error, user_error_line, pipeflow_error, reason, tips	Present even if the solver fails to converge.
provenance	pandapipes_version, thresholds_profile, sandbox_caps, hostname	Supports reproducibility and audit.

A.3 Reproducibility Checklist

- Fix fluid profile (gas, hydrogen, heat) and acceptance band version.
- Pin library versions and OS image if possible.
- Save artifacts and run scores per run_id; never overwrite.
- Recompute KPIs from artifacts rather than logs.
- If doing sweeps, persist each combination's parameters and summary.
- Keep a minimal README per project with dataset provenance and contact.

Appendix B — KPI Definitions and Acceptance Profiles

B.1 KPI Definitions

KPI	Definition	Units	Purpose
max_velocity	Maximum pipe mean velocity across res_pipe	m/s	Tracks erosive risk and energy loss hotspots.
min_node_pressure	Lowest junction pressure across res_junction	bar	Ensures service-pressure compliance at worst point.
max_pipe_dp	Maximum absolute segment Δp (|p_from − p_to|) across res_pipe	bar	Flags bottlenecks and choke points.
min_reynolds	Minimum Reynolds number across res_pipe	dimensionless	Qualitative stability indicator (laminar/turbulent).
velocity_violations	Count of pipes exceeding velocity_ok_max	count	Simple measure of how far from band.
pressure_violations	Count of nodes below the profile's minimum pressure threshold	count	Safety-/service-weighted indicator.
run_score	Heuristic scalar: 10·pressure_violations + 5·velocity_violations +	—	Lower is better; used for learning and comparisons.

	0.2·max_velocity – 0.2·min_node_pressure		

B.2 Standard Acceptance Bands Per Fluid

Fluid	**Peak Velocity (OK)**	**Min Pressure (OK)**	**Max Δp per Segment (OK)**	**Reynolds Guidance**	**Temperature Band**
Gas (lgas)	≤ 15.0 m/s	≥ 0.95·pn	≤ 0.30 bar (warn 0.60)	≥ 2300 preferred	273–373 K (informative)
Hydrogen (H_2)	≤ 20.0 m/s	≥ 0.94·pn	≤ 0.25 bar (warn 0.50)	≥ 3000 preferred	273–373 K (informative)
Heat (water)	≤ 2.5 m/s	≥ 0.90 bar	≤ 0.15 bar (warn 0.30)	— (incompressible regime)	303–353 K typical

Notes:

- "pn" denotes design nominal pressure for that junction.
- "Loose" or "Strict" profiles can be used; the same acceptance logic applies.

Appendix C — Networks and Datasets Used

Name	**Fluid**	**Pipes**	**Junctions**	**Purpose**	**Notes**
Branitz (gas)	Natural gas	4635	≈4187	Large stress test	Realistic distribution-scale case; already well-designed. Used with blind fault-injection to create solvable challenges.
Hydrogen microgrid	Hydrogen	≈145	≈120	Compact cross-fluid validation	Synthetic suite to show fluid-aware behavior; higher velocity bands.
Heat loop	Water (heat)	≈96	≈80	Compact cross-fluid validation	Incompressible flow; low-velocity bands and pump setpoint tweaks.

Data Integrity:

- Fault injections used for benchmarking are synthetic and do not reflect the actual physical system.
- HELICS v3.5.2 was used only to orchestrate personal (manual) Branitz sessions; AI modes used steady-state harness only.

Appendix D — Orchestration Modes and Stop Conditions

Mode	**Loop Style**	**Planning/Learning**	**Typical Actions**	**Stop Rule**

Personal baseline	Manual/scripted	None	Estimate, edit, simulate, inspect	Bands met; violation counters ≤ 2.
Tool-only AI	Single turn	None	Validate, simulate, at most one small edit	Bands met; violation counters ≤ 2.
Multi-agent AI twin	Short planned sequence	Light memory and tiny searches	Validate, simulate, compute KPIs/issues, minimal fix, re-simulate	Bands met; violation counters ≤ 2; lower run score than start.

Stop Criteria (All Modes):

- Fluid-aware bands satisfied.
- Violation counters at zero or a small bound (≤ 2).
- Prefer earliest point satisfying the band ("stop early" principle).

Appendix E — Agent Responsibilities and Interfaces (Conceptual)

Agent	**Core Responsibilities**	**Inputs**	**Outputs**	**Local Score (Examples)**
Supervisor	Plan minimal, safe next step; apply early stop	Intent, fluid profile, artifacts, preferences	Actions, expected delta, stop decision	Plan efficiency, improvement predicted/observed
Security/Compliance	Static validation (AST/whitelist), policy gate	User code	Allow/block with line/column, policy messages	Policy compliance, false pos/neg trend
Sandbox	Isolated execution with resource caps	Validated code, resource hints	Artifact JSON, stderr, caps used	Reliability (timeouts, wall-time), envelope fit
Physics	KPIs/issues; small surrogate for deltas	Artifacts, thresholds	KPIs, issues, suggested targets, uncertainty	Physics quality (prediction error, coverage)
Toolsmith/Generator	On-demand helper tools (typed, tested)	Narrow specs, artifacts	Registered tool, quality score	Tool quality (pass rate, contribution)

Cost	Rough CAPEX with breakdown	Artifacts, context	Low/mid/high with assumptions	Calibration fit (vs later actuals)
Critic/Memory	Lessons and preferences from score deltas	Before/after KPIs, actions	Preference bias, safety flags	Signal-to-noise, stability of advice

Appendix F — Safety and Compliance Posture (Summary)

Policy Element	**Value/Rule**	**Rationale**
Import policy	Only pandapipes import root allowed	Eliminate unsafe modules (I/O, system).
Call whitelist	Only documented pp.* functions on whitelist	Minimize attack surface; predictable solver calls.
Dynamic code patterns	Block eval/exec, indirect getattr tricks	Prevent obfuscation, runtime injection.
Sandbox CPU	Baseline 30 s (adapt within ceilings)	Bound compute per run; avoid lockups.
Sandbox memory	Baseline 2048 MB (adapt within ceilings)	Contain RSS; protect co-tenants.
Sandbox wall-time	Baseline 60 s	End-to-end guardrail; guarantees responsiveness.
File access	Constrained to dedicated project path	Prevent data exfiltration; ensure isolation.
Failure reporting	Line/column on static block; reason + tips on runtime failure	Make fixes obvious; speed recovery.
Observability	Debug events: sim.start, sim.stderr, sim.end; artifacts per run_id	Transparent, replayable runs.

Appendix G — Cost Model Assumptions (Indicative)

Component	**Basis**	**Typical Handling**	**Notes**
Pipe supply	Length by diameter class and material	Rate €/m by class	Region/context factor applied.
Installation/excavation	Length-normalized envelope	Rate €/m by class	Includes trenching and labor.
Reinstatement	Length-normalized	Flat rate €/m	Surface restoration allowance.
Fittings	Length-normalized	Flat rate €/m	Misc. connectors, bends.

Valves	Count or spacing heuristic	Unit €/valve (PE vs steel)	Count explicit if available.
Pumps/compressors	Device counts	Unit €/device	Heat networks: pump setpoints.
Engineering	% of subtotal	Typical 10%	Design/PM/permits.
Contingency	% of subtotal	Typical 15%	Scope and price risk buffer.
Region factor	Multiplier by region/context	e.g., 1.10 (DE example)	Calibrated with feedback where available.

Notes:

- Example totals cited in the thesis are order-of-magnitude planning estimates, not bids.
- Calibration (if actuals become available) can adjust multipliers per region/context.

Appendix H — Risk Register and Mitigations

Risk	Likelihood	Impact	Indicators	Mitigation	Residual
Preference drift (memory)	Medium	Medium	Extra loops after topology change	Segment by project+fluid; decay; health gate	Low
Surrogate miscalibration	Medium	Medium	KPI prediction error at band edges	Attach uncertainty; clamp near limits; tiny sweeps	Moderate at edges
Score bias (pressure>velocity)	Low/Med	Low/Med	Accepted but near-limit velocity	Offer Strict profile; explicit trade-off note	Low
Global edits mask locals	Medium	Medium	Green bands but lumpy Δp hotspots	Add selectors for local edits (roadmap)	Medium until shipped
Generated tools overhead	Low	Low/Med	Long runtime; low contribution	Quality gates; retire laggards	Low
LLM/model drift	Low	Medium	Repro pack regressions	Pin models; prompt versioning; CI gates; fallback	Low
Sandbox timeouts	Low	Low	Timeout spikes	Envelope learning; policy ceilings	Low
Security drift	Low	High	New pp.* surfaces, import gaps	Treat policy as code; red-team suites	Low

Appendix I — Performance Envelope and Resource Use

Network	Typical Simulate Time/Loop	Planning Overhead (Multi-agent)	Typical Loops to Acceptance	Notes
Branitz (gas)	≈ 3.8 s	≈ 0.6 s	1–2	Solver dominates; sweeps add a few seconds when used.
Hydrogen microgrid	≈ 1.6 s	≈ 0.3–0.5 s	1–2	Higher velocity band; lighter edits.
Heat loop	≈ 1.2 s	≈ 0.2–0.4 s	1	Incompressible; small pump tweaks.

Guidance:

- The AI twin often stops after a single fix pass; end-to-end wall-time remains practical even with small planning overheads.
- Use artifacts (not logs) to compare runs across environments.

Appendix J — Operating Procedure (End-to-end)

J.1 Standard Loop (All Modes)

- **Validate (static):** Confirm only allowed imports and calls appear.
- **Simulate (sandboxed):** Collect artifact JSON regardless of outcome.
- **Compute KPIs and issues:** Apply fluid-aware thresholds.
- **Minimal fix (if requested):** Prefer small, physically sensible edits.
- **Re-simulate (if fix applied).**
- **Stop** as soon as the band turns green and violation counters drop to ≤ 2.

J.2 AI Twin Specifics (Multi-agent)

- Supervisor sets fluid profile and small targets, reading light preferences.
- Security/Compliance enforces AST/whitelist policy with explicit messages.
- Sandbox executes under caps (CPU, memory, wall-time).
- Physics returns KPIs, issues, and—optionally—small, uncertain predictions for action effects.
- Critic/Memory records run-score delta, writes short lessons, and updates preferences.
- Scenario sweep (bounded) runs only when choices are close.

Appendix K — Evidence and Observability

Channel	What It Shows	Why It Matters

Artifact JSON per run_id	Summary, design tables, result tables, failure metadata	Single source of truth; reproducible and comparable.
Debug event stream	sim.start, sim.stderr, sim.end; tool calls	Trace planning/execution; teach and troubleshoot.
KPI and issue panels	Global KPIs, worst elements, normalized issues	Operator-facing "state of the network" anchored in artifacts.
Diff and action log	Proposed small edits and their rationale	Keeps edits auditable and reversible.

Appendix L — Acceptance Bands: Profiles and Customization

Profile	Use Case	Adjustments
Standard	Default conversational work	Values summarized in Appendix B.2
Strict	Safety-critical or high-fidelity tasks	Tighten velocity and Δp; increase min pressure threshold
Loose	Exploratory or early scoping	Relax bands (with explicit warning)
Custom	Regulatory or site-specific	User-defined thresholds by KPI

Notes:

- The same stop rule applies for all profiles.
- Profile selection must be captured in artifacts for fair comparisons.

Appendix M — Glossary

Term	Meaning
Artifact (JSON)	Compact, structured output per simulation turn (design, results, summary, provenance).
Band (acceptance)	Fluid-aware thresholds for KPIs; define when a run is "good enough."
Δp (segment)	Absolute pressure drop across a pipe segment, computed from end-node pressures.
run_id	Unique identifier for one simulation turn (and its artifact).
"Stop early" rule	Halt as soon as bands are met and violation counters drop to a small bound.
Surrogate	Small predictive model used to estimate KPI deltas for tiny, safe actions.
Preference	Light, bandit-like bias learned from prior run-score deltas to order/soften actions.
HELICS	Co-simulation coordinator (used only in personal baseline for Branitz sessions).

Appendix N — Ethics, Safety, and Scope

- **Intended use:** Exploratory planning and operator training in steady-state; not a certified design or regulatory submission.
- **Safety-by-design:** Strict import/call policy, sandbox caps, and explicit failure tips aim to prevent unsafe execution and make error recovery quick.
- **Transparency:** Every acceptance decision can be traced to thresholds and artifacts.
- **Non-expert accessibility:** Conversational goals and clear artifacts reduce barriers without hiding physics.
- **Limitations:** Global edits will be coarse in local pockets until element-level selectors are available; time-series is out of scope in this version.

Appendix O — Data and Code Availability, Contact, and Versioning

- **Public repository (current public version):** https://github.com/erikwiedenhaupt/pipe-wise-public
- **Latest version (selected access):** https://github.com/erikwiedenhaupt/pipe-wise
  - This repository contains the latest internal version with full networks and supplementary materials.
  - Access is available to the latest version is available upon request.
- **Important:** The public repository may not reflect the latest internal version or the full set of networks and tasks used in evaluation.
- **Contact for access and questions:** erik.wiedenhaupt@ieg-extern.fraunhofer.de
- **Recommended citation format (example):**
  - Wiedenhaupt, E. (2025). *Pipe-Wise: A conversational multi-agent digital AI twin for urban pipe networks* (Bachelor Thesis). Fraunhofer IEG. Public repository: https://github.com/erikwiedenhaupt/pipe-wise-public
- **Licensing:** Please consult the repository LICENSE file for terms.
- **Third-party acknowledgments:** PandaPipes and HELICS are used as detailed in the thesis; see their respective licenses and documentation.

Appendix P — How to setup

Repository

- Clone whichever repo you can access:

  - Public: https://github.com/erikwiedenhaupt/pipe-wise-public

  - Private: https://github.com/erikwiedenhaupt/pipe-wise

Prerequisites

- Docker or Docker Compose (preferably)

- Node.js v.18 + npm (for frontend dev or local runs)

- Python 3.11 (for backend dev without Docker)

- Git

Environment configuration

- Backend: create .env (repo root for docker-compose; or backend/.env for local backend runs). Do not commit secrets. The values below are placeholders—replace them with your own.

- Frontend: create frontend/.env.local to point the UI at the API.

Backend environment variables (.env)

- Where: repository root when using docker-compose (env_file: .env), or backend/.env for local backend runs.

- AZURE_OPENAI_ENDPOINT — Required; your Azure OpenAI resource endpoint (e.g., https://YOUR_RESOURCE.openai.azure.com/).

- AZURE_OPENAI_API_KEY — Required; your Azure OpenAI API key. Never commit this. Rotate immediately if a real key was exposed.

- AZURE_OPENAI_DEPLOYMENT — Required; chat/completion deployment name (e.g., gpt-4o-mini-2024-07-18).

- AZURE_OPENAI_API_VERSION — Required; Azure OpenAI API version (e.g., 2024-02-15-preview).

- AZURE_OPENAI_EMBEDDINGS_DEPLOYMENT — Required; embeddings deployment name (e.g., text-embedding-3-large).

- PIPEWISE_CHAT_ENGINE — Optional; chat engine mode (default: responses).

- MEMORY_TOP_K — Optional; memory retrieval top-k (default: 5).

- MEMORY_MIN_WEIGHT — Optional; memory minimum weight (default: 0.0).

- COST_INPUT_PER_1K_EUR — Optional; input cost tracking per 1k tokens (default: 0.00015).

- COST_OUTPUT_PER_1K_EUR — Optional; output cost tracking per 1k tokens (default: 0.00060).

- PIPEWISE_AGENT_LEARNING — Optional; 1 to enable agent learning (default: 1).

Example backend .env (using placeholders for sensitive information)

```
AZURE_OPENAI_ENDPOINT=https://YOUR_RESOURCE.openai.azure.com/
AZURE_OPENAI_API_KEY=YOUR_KEY
AZURE_OPENAI_DEPLOYMENT=gpt-4o-mini-2024-07-18
AZURE_OPENAI_API_VERSION=2024-02-15-preview
AZURE_OPENAI_EMBEDDINGS_DEPLOYMENT=text-embedding-3-large
PIPEWISE_CHAT_ENGINE=responses
MEMORY_TOP_K=5
MEMORY_MIN_WEIGHT=0.0
COST_INPUT_PER_1K_EUR=0.00015
COST_OUTPUT_PER_1K_EUR=0.00060
PIPEWISE_AGENT_LEARNING=1
```

Frontend .env.local

```
# Local dev
NEXT_PUBLIC_API_BASE=http://localhost:8000/api
# If deployed remotely, set to your server:
# NEXT_PUBLIC_API_BASE=http://<SERVER-IP>:8000/api
```

Frontend dependencies (npm)

- The repo includes package.json, so npm install is usually enough. If you see missing modules or are setting up from scratch, run:

```
cd frontend
# Runtime deps
npm install next@13.5.4 react@18.2.0 react-dom@18.2.0 react-markdown@^8.0.5 remark-gfm@^4.0.1
# Build/dev deps
npm install -D tailwindcss@^3.3.0 postcss@^8.4.24 autoprefixer@^10.4.14 @tailwindcss/typography
- Tailwind/PostCSS configs are present (tailwind.config.js, postcss.config.js). If missing, initialize:
npx tailwindcss init -p
```

Quick start (using Docker, recommended)

1) Clone and enter the repo:

```
git clone https://github.com/erikwiedenhaupt/pipe-wise-public
cd pipe-wise-public
```

2) Create and populate .env at the repository root with your own Azure keys and deployment names. Do not commit this file.

3) Create frontend/.env.local:

```
echo "NEXT_PUBLIC_API_BASE=http://localhost:8000/api" > frontend/.env.local
```

4) Build and run:

```
docker compose up -d --build
```

5) Access:

- Frontend: http://localhost:3000

- Backend: http://localhost:8000

- API docs: http://localhost:8000/docs

- Health: http://localhost:8000/api/healthz

Note: docker-compose runs dev mode (uvicorn --reload and next dev). Dockerfile.frontend is set up for production builds (npm run build + start), but compose overrides the command for development.

If your Docker build complains about missing Dockerfile paths

- Use this compose override to build with the root-level Dockerfile.backend and Dockerfile.frontend:

```yaml
version: "3.9"
services:
  backend:
    build:
      context: .
      dockerfile: Dockerfile.backend
    container_name: pipewise-backend
    ports: ["8000:8000"]
    env_file: [.env]
    volumes:
      - ./backend:/app
    command: uvicorn main:app --host 0.0.0.0 --port 8000 --reload
    restart: unless-stopped

  frontend:
    build:
      context: .
      dockerfile: Dockerfile.frontend
    container_name: pipewise-frontend
    ports: ["3000:3000"]
    volumes:
      - ./frontend:/app
    working_dir: /app
    command: npm run dev -- -H 0.0.0.0 -p 3000
    environment:
      - NODE_ENV=development
    restart: unless-stopped

```

Local development without Docker

I would advise against the setup without using docker, if you still prefer to continue follow these steps:

Backend:

```
cd backend
python -m venv .venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate
pip install -r requirements.txt
# Ensure backend/.env exists or export the variables in your shell
uvicorn main:app --host 0.0.0.0 --port 8000 --reload
```

Frontend:

```
cd frontend
npm install
# Ensure frontend/.env.local points to your backend, e.g., http://localhost:8000/api
npm run dev -- -H 0.0.0.0 -p 3000
```

Verification

- Health:

```
curl -s http://localhost:8000/api/healthz
```

- UI: open http://localhost:3000

- If the UI can’t reach the API, check NEXT_PUBLIC_API_BASE and that port 8000 is reachable. You may need to configure port forwarding or your firewall.

Notes

- Secrets: .env contains placeholders—replace with your own credentials. Do not commit .env. If any real key was exposed publicly, rotate it immediately and remove it from the history.

- Ports: 3000 (frontend) and 8000 (backend) must be open.

- Public/private code: Not all source modules are required to start; the provided API and UI are sufficient.

- Repo choice: Use pipe-wise-public if you don’t have private access; otherwise pipe-wise.

Appendix Q — Summary of Key Findings

Topic	Finding
Multi-agent vs tool-only	Higher success and fewer loops with modest planning overhead; early stops keep wall-time practical.
AI twin vs normal twin	Under identical physics and bands, the AI twin reached acceptance more often and in fewer iterations.
Fluid-aware profiles	Essential for realistic targets across gas, hydrogen, and heat.
Safety and traceability	AST/whitelist gate plus sandbox caps; artifacts per run_id keep comparisons fair and reproducible.
Practical edits	Clipped global diameter scaling and gentle inlet/pump tweaks clear most cases in one pass.
Roadmap	Local selectors, adaptive targets over small discrete intensities, diagnostic Δp explainer, optional time-series path.

Closing Note

This appendix consolidates the operational, safety, and evaluation details necessary to reproduce and scrutinize results without exposing code in this document. All code, full artifacts, and network assets are referenced via the repositories above. The public repository (https://github.com/erikwiedenhaupt/pipe-wise-public) may not contain the latest version. For the latest version with complete networks and supplementary materials, please access the selected repository (https://github.com/erikwiedenhaupt/pipe-wise) or reach out directly at erik.wiedenhaupt@ieg-extern.fraunhofer.de.