INDEPENDENT TECHNICAL PORTFOLIO CASE STUDY

ElectroBench

A Reproducible Code-Graded Benchmark Prototype for LLM-Generated Electrical Engineering Simulations

Pedro Vitor Brunello Pagliarin

Electrical Engineer | Python | AI Model Evaluation

Porto Alegre, Brazil | pedrovitor.pagliarin@hotmail.com
pvpprojects.netlify.app

Project version: v0.2.0 portfolio release

Porto Alegre, Brazil

Document date: July 2026

Contents

List of Figures

List of Tables

Abstract

ElectroBench is a reproducible, code-graded benchmark prototype designed to evaluate Python solutions generated by large language models for electrical-engineering simulation tasks. The project combines power-system analysis with `pandapower`, converter-control analysis with `python-control`, and nonlinear circuit models implemented with SciPy. It adds numerical grading, engineering tolerances, physical validity checks, diagnostic repair experiments, immutable reference outputs, and separated Docker runner/grader environments. A twelve-task blind set was frozen before the first paid evaluation and tested with five attempts per task and condition. On this descriptive pilot, the baseline condition achieved **18.3%**, neutral official-documentation excerpts achieved **50.0%**, and the union of documentation plus conditional diagnostic repair achieved **60.0%**. The circuit track is explicitly presented as **SciPy-graded**; ngspice is validated through a command-line integration smoke test rather than claimed as the grading backend. The case study documents the engineering rationale, experimental architecture, task examples, blind-evaluation results, failure patterns, limitations, and reproduction path.

Keywords: electrical engineering, Python, large language models, code grading, pandapower, python-control, SciPy, ngspice, Docker, benchmark evaluation

1 Project overview

ElectroBench evaluates whether a language model can produce Python code that *actually executes* an electrical-engineering analysis and returns a numerically correct, contract-compliant result. The project was built as a portfolio-grade prototype for work at the intersection of electrical engineering, scientific Python, and frontier-model evaluation. Its scope covers task design, simulator selection, golden-reference generation, tolerance definition, containerized execution, repeated model trials, failure analysis, and experiment freezing (Pagliarin, 2026b,a).

The benchmark contains twenty-four versioned tasks: twelve development/post-hoc tasks and twelve never-tuned blind tasks. The frozen blind set contains three control-system tasks, five power-system tasks, and four circuit-simulation tasks. Each task is expressed as a structured YAML specification containing an engineering statement, inputs, required outputs, numerical tolerances, and execution constraints.

Portfolio claim in one sentence

ElectroBench is a reproducible code-graded prototype for evaluating LLM-generated electrical-engineering Python, with simulator-backed tasks, numerical tolerances, Docker runner/grader separation, a hash-frozen blind set, and descriptive multi-attempt results.

Table 1: Project scope and evidence at a glance.

Element	Implemented evidence
Task catalog	24 YAML tasks; 12 development/post-hoc and 12 frozen blind tasks
Engineering domains	Power systems, converter control, and nonlinear circuits
Scientific stack	`pandapower`, `python-control`, SciPy; ngspice CLI smoke
Evaluation	Baseline, neutral official documentation, conditional diagnostic repair
Execution	Python 3.11 lockfile; isolated Docker runner; separate trusted grader
Correctness	Immutable goldens, absolute/relative tolerances, output schema, physical invariants
Reproducibility	SHA-256 freeze of task split, task files, references, docs, protocol, and dependency lock
Blind pilot	12 tasks, 5 attempts per task and condition, first paid run after freeze

Source: Project repository and curated evidence files (Pagliarin, 2026a,c).

2 Problem

Language models can write convincing engineering explanations while still producing code that fails to import a library, misuses a simulator API, applies the wrong topology edit, extracts the wrong result table, computes the wrong metric, or returns an answer outside an engineering tolerance. For simulation-heavy problems, plausibility is therefore an insufficient success criterion.

The practical evaluation problem is to distinguish among at least four outcomes:

1. a response that is rhetorically plausible but not executable;
2. executable code that uses the wrong mathematical or simulation procedure;
3. numerically close code that violates an output contract or physical invariant; and

4. a complete solution that executes inside the constrained environment and matches the reference result within justified tolerances.

ElectroBench addresses this problem by turning each challenge into an executable contract. The model must implement a function of the form `solve(inputs) -> dict`; the returned dictionary must contain every required key; and every metric must satisfy its tolerance and validity checks.

3 Why existing AI answers are insufficient

A conventional text-only evaluation often rewards verbal fluency rather than engineering correctness. This is especially risky in electrical engineering because a small implementation error can alter a contingency ranking, a stability margin, a settling-time estimate, or a ripple measurement. Tool-dependent tasks also expose whether the model can navigate real software objects and numerical conventions rather than merely restate textbook formulas.

The blind evaluation provides direct evidence of this gap. Across the active attempts, the benchmark recorded 64 numerical mismatches, 32 execution errors, three security-policy violations, two physically invalid outputs, and two timeouts. These categories show that "wrong" is not a single failure mode: it may arise before simulation, during execution, in metric extraction, or after a numerically plausible but physically inconsistent result.

Interpretation boundary

The project measures code execution and numerical agreement under a defined task contract. It does not claim to measure every form of engineering competence, nor does a passing solution prove general professional readiness. The reported experiment is a descriptive pilot with five attempts per task.

4 Benchmark architecture

Figure 1 summarizes the end-to-end system. Task files, neutral documentation excerpts, and freeze metadata are assembled into prompts. Generated Python is executed inside a runner image that contains the scientific stack and an allowed 10 kV ring-network asset, but does not contain golden references, reference solutions, grader code, or the full ElectroBench package. Results are returned to the trusted grading side, where schema checks, numerical tolerances, and physical invariants are applied. Prompts, generated code, execution traces, latency, cost, and fingerprints are stored as experiment artifacts.

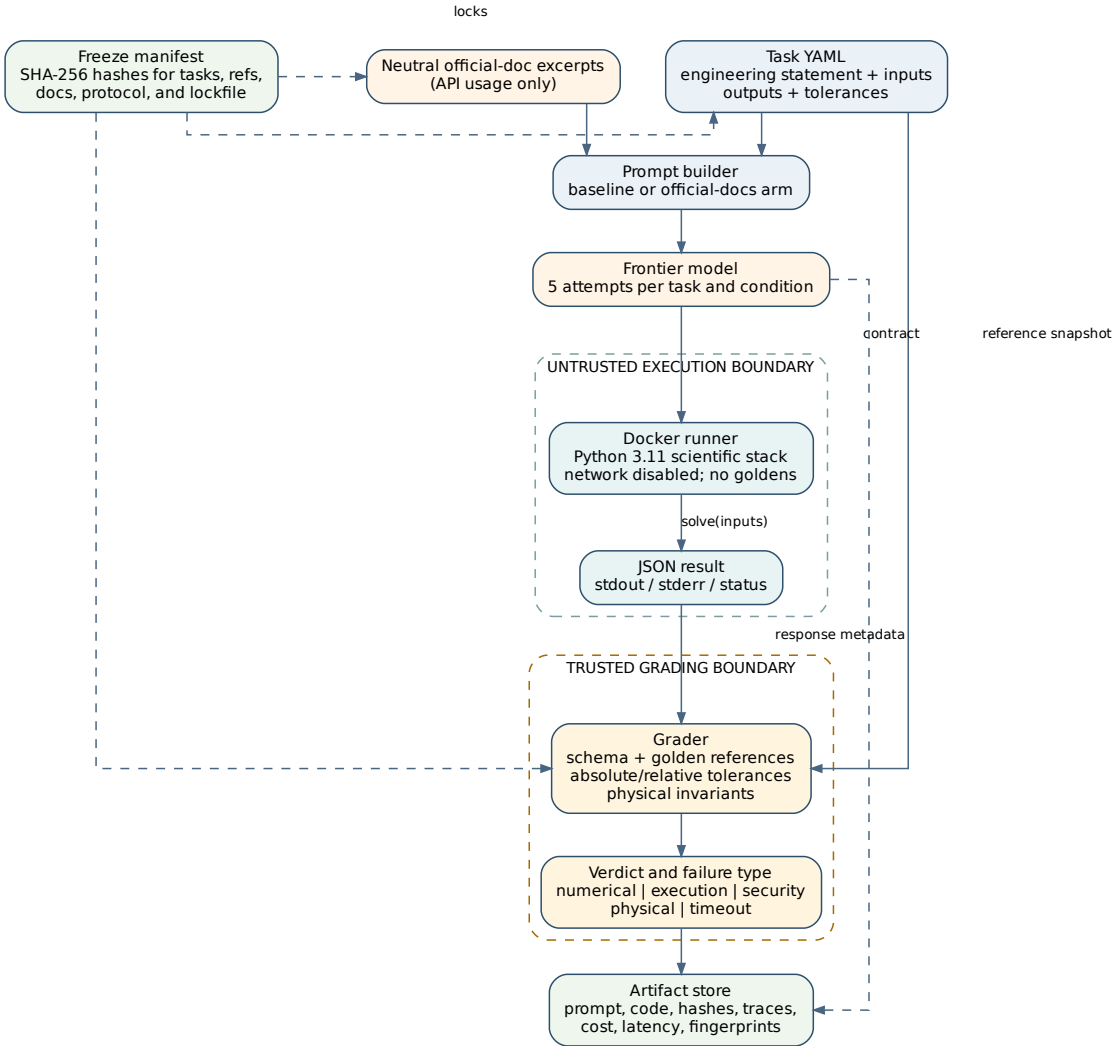

Figure 1: ElectroBench architecture and runner/grader trust boundaries.

Source: Author-generated diagram based on the shipped Dockerfiles, methodology, and experiment harness.

4.1 Runner/grader separation

The runner image is intentionally minimal: Python 3.11, pinned scientific dependencies, ngspice, and the submission-safe `electrobench_assets` package. The grader image contains the full repository, including tasks, tests, reference data, and solutions. The Docker Compose configuration disables networking for the runner. Static import checks additionally block access to internal simulator and golden-reference modules (Pagliarin, 2026b).

This separation is stronger than running model code directly inside the grading process, but it is not described as a production-grade hostile-code sandbox. Further hardening—for example a read-only filesystem, non-root user, dropped Linux capabilities, memory/CPU/PID limits, and stricter seccomp profiles—remains

part of the validity roadmap.

5 Electrical engineering domains

Table 2: Blind task families and the simulator capability each family exercises.

Domain	Blind tasks	Representative problems	Primary stack
Control systems	3	Buck-converter average model with PI controller; nominal, heavy-load, and parameter-sweep cases; open-loop margins and closed-loop step metrics	`python-control`
Power systems	5	IEEE 14/30 single-line N–1 analysis; critical outage ranking; 10 kV ring power flow under time-dependent load profiles	`pandapower`
Circuit simulation	4	Half-wave rectifier with RC load, heavy rectifier case, series RLC transient, and diode clipper metrics	SciPy numerical models; ngspice CLI smoke

Source: Frozen blind split and task YAML files.

5.1 Power systems

The power-system tasks depend on network construction, load scaling, AC power-flow convergence, topology edits, and extraction of results from the simulator's result tables. In the N–1 family, every initially in-service line is removed in turn. The critical contingency is defined as the converging outage with the highest maximum line-loading percentage, with ties broken by the lowest line index. This design forces the solution to combine simulator execution with deterministic post-processing rather than return a single textbook quantity. The project uses `pandapower`, an open-source Python framework for power-system modeling and analysis (Thurner et al., 2018).

5.2 Control systems

The control track models a continuous-conduction-mode buck converter using an average transfer function and an ideal PI voltage controller. The model must construct open- and closed-loop systems, compute gain and phase margins, convert the gain margin to decibels, calculate a time-domain step response, derive overshoot and a custom two-percent settling time, and determine stability from closed-loop pole locations. Parameter-sweep tasks require reporting the worst case over combinations of inductance, capacitance, and load.

5.3 Nonlinear circuits

The circuit track measures waveform-derived quantities such as peak, minimum, mean, ripple, conduction duty, current peak, dissipated energy, and zero crossings. The reference implementation uses numerical Python/SciPy models (Virtanen et al., 2020). Netlist files and ngspice integration are present, and the Docker environment demonstrates that the ngspice CLI can run a batch smoke test (The ngspice Project, 2026); however, the graded circuit tasks currently specify a Python/SciPy backend.

Precise ngspice claim

The defensible wording is: *"circuit simulation track with SciPy models and an ngspice CLI integration smoke test."* The current portfolio does **not** claim that the circuit-task grading path executes ngspice for every candidate solution.

6 Example tasks

6.1 Example 1: IEEE 14-bus N–1 contingency

The following abridged task statement illustrates the level of engineering specificity provided to the model:

Listing 1: Abridged N–1 task statement and output contract.

```
Perform a single-line N-1 contingency analysis on the ieee14 network
while scaling all load P and Q by load_scale. Consider only lines that
are initially in service. Critical contingency = outage with the largest
maximum line loading percent among converging cases; ties break to the
lowest line index.

Return a dict with ALL keys:
critical_contingency, minimum_voltage_pu,
maximum_line_loading_percent, active_power_losses_mw

Tolerances:
critical_contingency: absolute 0
minimum_voltage_pu: absolute 0.001
maximum_line_loading_percent: relative 0.005
active_power_losses_mw: relative 0.002
```

The task cannot be satisfied by naming a likely weak line. A valid solution must build the network, scale the loads, iterate line outages, handle non-converging cases, compute the ranking metric, and report the critical-case metrics in the required schema.

6.2 Example 2: buck converter with PI control

A nominal control task defines

$$G_{vd}(s) = \frac{V_{in}}{LCs^2 + (L/R)s + 1}, \qquad C(s) = K_p + \frac{K_i}{s}, \tag{1}$$

with

$$R = \frac{V_{out}^2}{P\,\lambda}, \tag{2}$$

where λ is the load fraction. The model must compute phase margin, gain margin in decibels, percent overshoot, two-percent settling time, and a binary stability flag. The exact time vector and settling definition are included to remove ambiguity among library defaults.

6.3 Example 3: rectifier with RC load

The rectifier task uses a half-wave source, series resistance, and a parallel RC load. After startup, metrics are measured over the final cycles:

$$V_{ripple} = V_{peak} - V_{min}, \qquad r_{\%} = 100\frac{V_{ripple}}{V_{avg}}. \tag{3}$$

The candidate receives circuit parameters and must implement the numerical model without opening internal files. This example demonstrates the current SciPy-graded circuit methodology while retaining a netlist label for future CLI-backed execution.

7 Code-grading methodology

7.1 Golden references and tolerances

Reference solutions generate immutable numerical outputs for each task. For a reported scalar $\hat{y}_j$ and golden value y_j, the generic acceptance rule is

$$|\hat{y}_j - y_j| \leq a_j + r_j\,|y_j|\,, \tag{4}$$

where a_j is the absolute tolerance and r_j is the relative tolerance. A task passes only when execution succeeds, every required output is present and valid, every metric satisfies its rule, and no physical invariant fails.

Tolerance selection reflects the engineering quantity. Discrete contingency indices use zero absolute tolerance; per-unit voltage may use a small absolute band; loading and loss quantities use relative tolerances; and binary stability flags must match exactly. The task files make these decisions explicit rather than burying them in grader code.

7.2 Failure taxonomy

The grader and harness distinguish among:

- **execution error**: import, syntax, runtime, or non-zero process exit;
- **numerical mismatch**: executable output outside one or more tolerances;
- **physical invalidity**: result violates a defined engineering invariant;
- **security violation**: blocked import or policy breach; and
- **timeout**: execution exceeds the task limit.

This taxonomy supports diagnostic repair. Instead of revealing a reference solution, the system can return structured information about what class of failure occurred and allow the model to attempt a correction.

7.3 Difficulty calibration

Development tasks were iteratively adjusted toward an aggregate no-documentation baseline pass rate of 10–30%. The calibrated development run reached 21.7%. The target is explicitly aggregate: individual task rates still vary, and the project does not claim that every task was independently tuned to the same interval (Pagliarin, 2026a).

8 Blind evaluation

The blind set was designed as a never-tuned evaluation set, separated from the development and post-hoc validation tasks. Before the first paid run, the project froze the twelve-task split and computed SHA-256 hashes for the split file, individual task specifications, official-documentation excerpts, golden-reference bundle, experiment protocol, and dependency lock. The freeze recorded protocol version 0.3.0, grader version 1.0.0, the Python 3.11 environment identifier, and runner/grader image metadata (Pagliarin, 2026c).

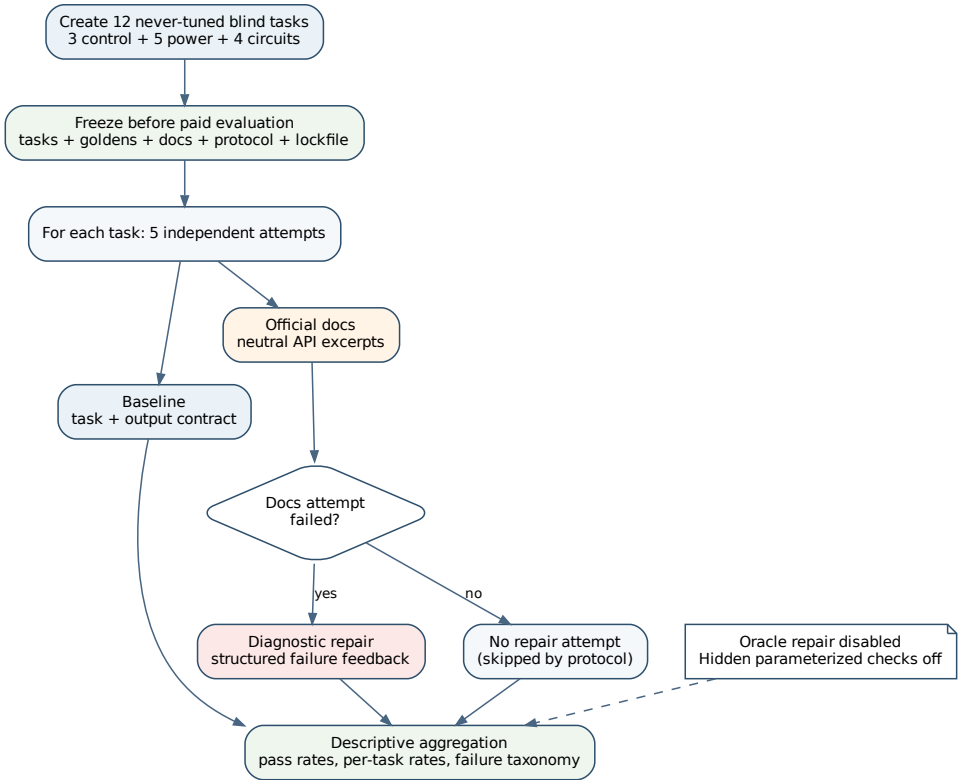

Figure 2: Frozen blind-evaluation protocol.

Source: Author-generated diagram based on the blind protocol and freeze artifacts.

For each of the twelve tasks, five attempts were generated under two primary conditions:

Baseline Task statement, inputs, output contract, and tolerance awareness only.

Official docs The same task plus neutral excerpts describing relevant library APIs without providing the task solution.

A diagnostic repair attempt was created only when the corresponding official-documentation attempt failed. Oracle repair was disabled for primary reporting. Hidden parameterized anti-hardcoding checks were implemented in the repository but disabled in this blind batch. The evaluated model was `deepseek/deepseek-v3.2` through OpenRouter with the Novita provider pin, and model-generated code was executed using the Docker runner (Pagliarin, 2026d).

9 Results

Table 3: Frozen blind-evaluation aggregate results.

Condition	Passes	Attempts	Pass rate
Baseline	11	60	**18.3%**
Official documentation	30	60	**50.0%**
Diagnostic repair | docs failure	6	30	**20.0%**
Documentation ∪ diagnostic repair	36	60	**60.0%**

Source: Recomputed from `supporting_evidence/blind_eval_summary.csv`; see also (Pagliarin, 2026d).

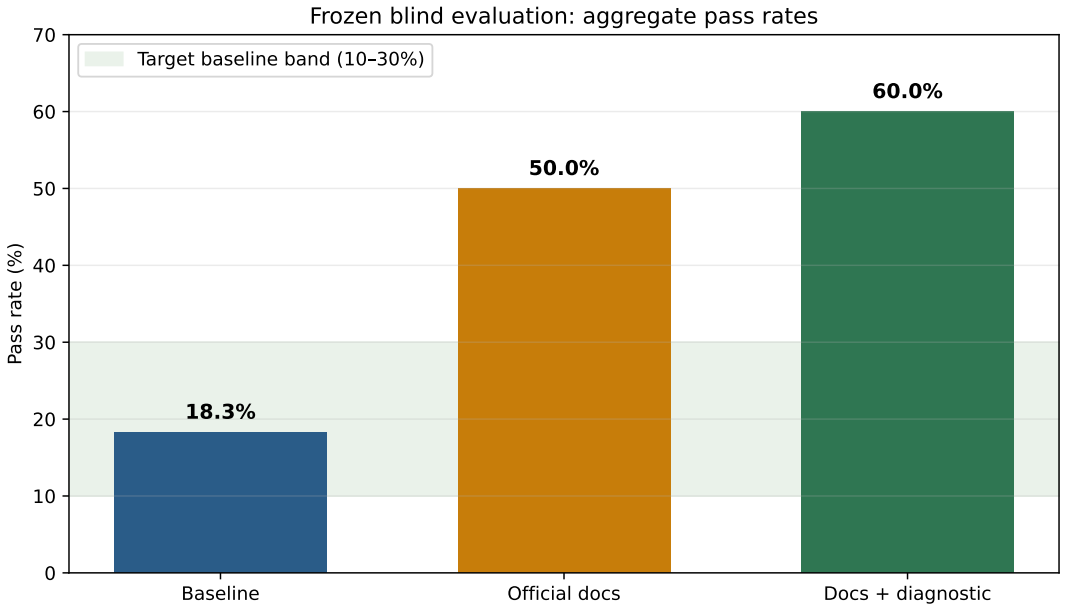

Figure 3: Aggregate pass rates on the frozen blind set.

Source: Author-generated from the shipped blind evaluation CSV. The shaded band is the 10–30% target for the baseline condition.

The baseline aggregate of 18.3% falls inside the target band without post-freeze retuning. Neutral documentation increased the pass rate to 50.0%. When failed documentation attempts received structured diagnostic feedback, six additional task-attempt pairs were recovered, producing a combined documentation-or-repair success rate of 60.0%. Total active API cost recorded for the blind run was approximately US$0.0952.

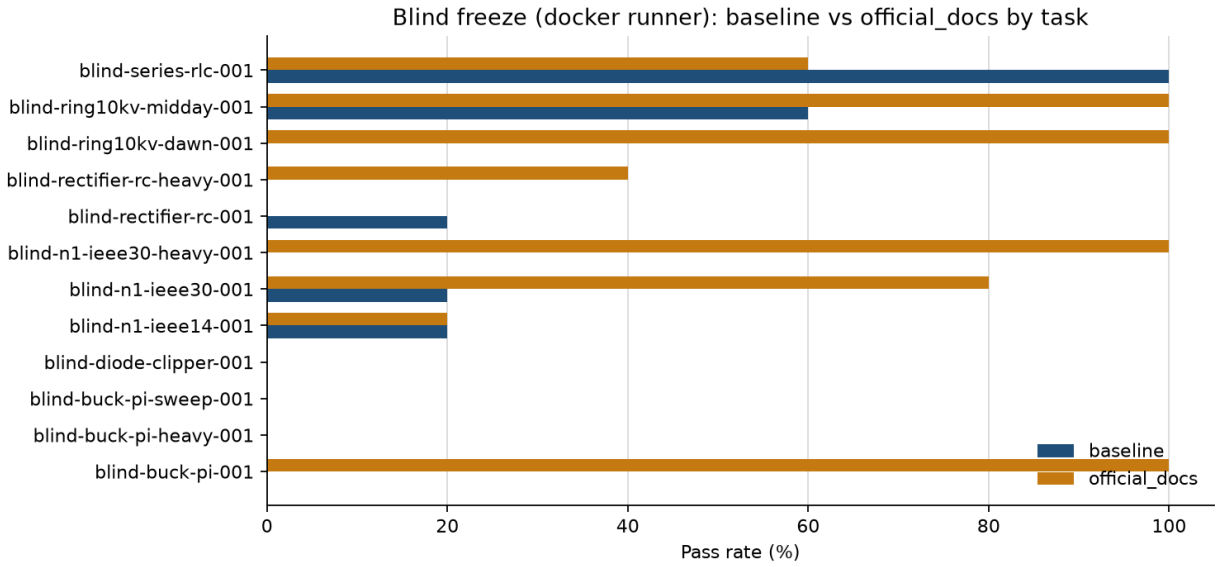

Figure 4: Baseline and official-documentation pass rates by blind task.

Source: ElectroBench curated evidence, generated from the blind-evaluation attempt records.

The task-level view shows substantial heterogeneity. The series-RLC task was solved by all baseline attempts, whereas several control and circuit tasks remained at zero. Documentation transformed the nominal buck task and multiple power-system tasks, but did not uniformly help the circuit family. This variation is why the project reports both aggregate and per-task results.

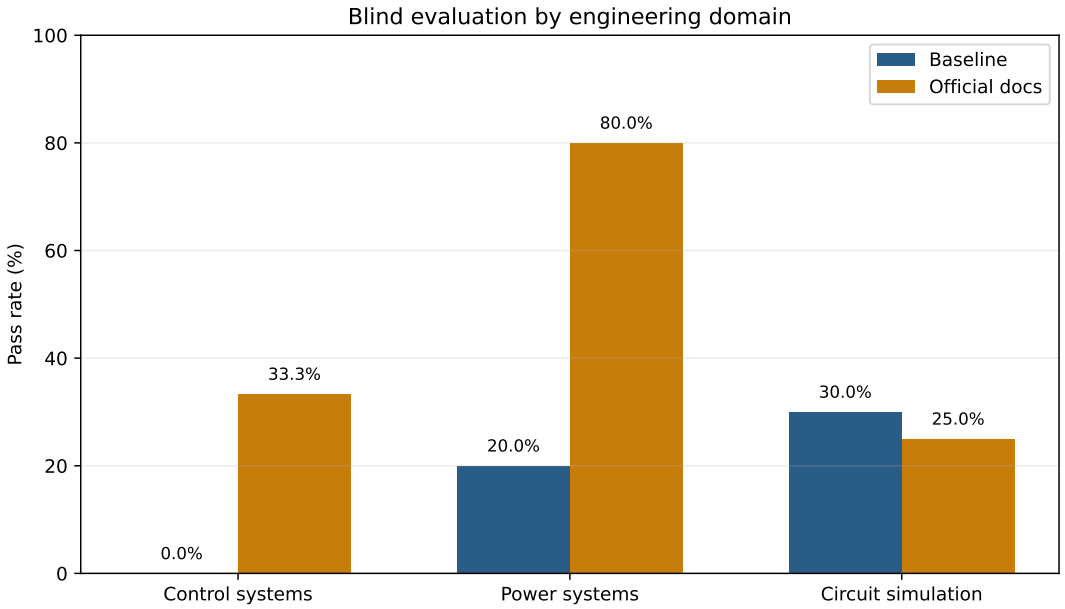

Figure 5: Blind pass rates by engineering domain.

Source: Author-generated from the shipped blind evaluation CSV.

At the family level, documentation raised control performance from 0% to 33.3% and power-system performance from 20% to 80%. Circuit performance decreased from 30% to 25%, indicating that API excerpts

alone did not address the dominant errors in those waveform-modeling tasks.

10 Failure analysis

Figure 6: Failure types among active, non-passing blind attempts.

Source: ElectroBench curated evidence, generated from the blind-evaluation attempt records.

Numerical mismatch was the dominant failure class, followed by execution error. This pattern supports three practical conclusions:

1. **Tool access is necessary but insufficient.** Even code that executes can extract the wrong quantity, use an inconsistent definition, or miss the tolerance.
2. **Documentation has domain-dependent value.** Power-system tasks benefited strongly because neutral documentation clarified network constructors, topology flags, result tables, and safe object copying.
3. **Repair needs targeted diagnostics.** A generic "try again" is less informative than distinguishing an API/runtime failure from a numerically wrong but executable solution.

The nominal buck task illustrates a documentation-sensitive API barrier: all five documentation attempts passed while no baseline attempt did. In contrast, the diode-clipper task remained at zero in both conditions, suggesting that the primary challenge was the model and metric logic rather than library discovery.

11 Limitations

1. **Small descriptive sample.** Five attempts per task are adequate for a portfolio pilot but produce wide uncertainty and should not be treated as definitive model-performance estimates.
2. **Single evaluated model/provider configuration.** The blind run used one model and one pinned provider. Cross-model generalization has not yet been established.
3. **Aggregate calibration.** The 10–30% target was applied to the aggregate baseline on development data. Per-task difficulty remains heterogeneous.
4. **Circuit backend.** Circuit tasks are graded through Python/SciPy models. Ngspice is present and

smoke-tested through the CLI, but it is not the required candidate grading path.

5. **Hidden checks disabled.** Parameterized anti-hardcoding checks exist but were disabled in this blind batch.
6. **Container hardening.** Runner/grader separation and network disabling improve isolation, but the runner is not claimed as a production-grade sandbox for arbitrary hostile code.
7. **No documented external senior review.** The package does not yet include a formal sign-off from a specialist reviewer in power, control, or circuit simulation.
8. **Blind-set publication.** Once full task/reference artifacts are publicly released, the same split should no longer be treated as secret for future model comparisons; a new freeze would be required.

12 How to reproduce

12.1 Local test environment

Published results target Python 3.11 and the pinned dependency lock:

Listing 2: Canonical local setup and test command.

```
python -m venv .venv311
# Activate the environment for the current operating system.
python -m pip install -r requirements.lock.txt
python -m pip install -e ".[dev]"
pytest -q
python scripts/revalidate_goldens_canonical.py
```

12.2 Docker images and smoke evidence

Listing 3: Build the separated images and run smoke checks.

```
docker build -f Dockerfile.runner -t electrobench-runner:py311 .
docker build -f Dockerfile.grader -t electrobench-grader:py311 .
python scripts/smoke_docker.py
python scripts/smoke_runner_isolation.py
python scripts/smoke_ngspice_cli.py
```

The included supporting evidence contains the blind summary CSV, freeze manifest, methodology note, ngspice smoke output, and runner-isolation smoke output. Re-running the paid model experiment requires a private API key stored outside the repository and should use a new experiment identifier rather than overwrite the frozen evidence.

12.3 Evidence map

Table 4: Files included in this Overleaf package for independent inspection.

Path	Purpose
`supporting_evidence/RESULTS.md`	Official project-facing result summary and framing
`supporting_evidence/methodology.md`	Split definitions, calibration rule, isolation, and circuit-backend caveat
`supporting_evidence/blind_eval_summary.csv`	Attempt-level table used to recompute figures and rates
`supporting_evidence/freeze_blind_v1.json`	Machine-readable freeze metadata and SHA-256 hashes
`supporting_evidence/ngspice_cli_smoke.json`	Evidence that ngspice executes in the environment
`supporting_evidence/runner_isolation_smoke.json`	Evidence of runner/grader separation checks
`examples/*.yaml`	Three representative task definitions
`examples/Dockerfile.*`	Runner and grader image definitions

13 Conclusion

ElectroBench demonstrates an end-to-end workflow for converting electrical-engineering expertise into verifiable model-evaluation tasks. Its strongest evidence is not a single score, but the combination of simulator-dependent problem formulation, executable Python contracts, numerical and physical grading, controlled prompt conditions, container separation, immutable experiment metadata, and honest interpretation of limitations.

The frozen blind pilot reached the intended aggregate baseline range and exposed a meaningful documentation effect, particularly in power-system tasks. The next technically valuable extensions are an ngspice-CLI-graded task, per-task calibration with larger samples, evaluation of a second frontier model, stronger container hardening, private parameterized checks during final scoring, and formal review by a senior engineer in at least one subfield.

A Representative task schema

Listing 4: Simplified YAML schema used by ElectroBench tasks.

```yaml
id: <task-id>
task_version: "2.0.0"
engine: pandapower | python-control | ngspice
category: <engineering-category>
statement: |
  <complete engineering problem>
inputs:
  <named parameters>
required_outputs:
  - <metric_1>
  - <metric_2>
```

```
tolerances:
  <metric_1>: {absolute: ...}
  <metric_2>: {relative: ...}
constraints:
  timeout_seconds: 30
  network_access: false
```

B Minimal submission contract

Listing 5: Candidate-code interface expected by the runner.

```python
def solve(inputs: dict) -> dict:
    """Run the required engineering analysis and return all metrics."""
    # Build or load the allowed simulation model.
    # Apply inputs and operating conditions.
    # Execute the solver or numerical integration.
    # Extract exactly the requested engineering metrics.
    return {
        "metric_1": value_1,
        "metric_2": value_2,
    }
```

C Result provenance

All percentages and counts in this case study were recomputed from the shipped `blind_eval_summary.csv`. The source file contains 240 rows because it stores baseline, official-documentation, diagnostic-repair, and disabled oracle-repair records for every task/sample pair. After excluding protocol-skipped rows, 150 active executions remain: 60 baseline, 60 official-documentation, and 30 conditional diagnostic-repair attempts. The active non-passing rows contain 64 numerical mismatches, 32 execution errors, three security violations, two physical-invalidity failures, and two timeouts.

References

Pagliarin, P. V. B. (2026a). *ElectroBench Results Draft: Portfolio and Paper-Facing Evidence*. Independent technical portfolio. Supporting artifact: `supporting_evidence/RESULTS.md`.

Pagliarin, P. V. B. (2026b). *ElectroBench Methodology Note*. Independent technical portfolio. Supporting artifact: `supporting_evidence/methodology.md`.

Pagliarin, P. V. B. (2026c). *ElectroBench Freeze Blind-v1 Manifest*. Created 23 July 2026. Supporting artifact: `supporting_evidence/freeze_blind_v1.json`.

Pagliarin, P. V. B. (2026d). *Blind-eval-v1: Frozen Blind Set Notes*. Independent technical portfolio. Supporting artifact: `supporting_evidence/blind_eval_notes.md`.

Thurner, L., Scheidler, A., Schaefer, F., Menke, J.-H., Dollichon, J., Meier, F., Meinecke, S., and Braun, M. (2018). pandapower—An open-source Python tool for convenient modeling, analysis, and

optimization of electric power systems. *IEEE Transactions on Power Systems*, 33(6), 6510–6521. doi:10.1109/TPWRS.2018.2829021.

Virtanen, P., Gommers, R., Oliphant, T. E., et al. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17, 261–272. doi:10.1038/s41592-019-0686-2.

The ngspice Project (2026). *Ngspice User's Manual*. https://ngspice.sourceforge.io/docs.html.