THOMAS MORE

Martin Zhelyazkov
Student Bachelor Applied Computer Science

2025 - 2026

Table of Contents

Introduction

This document describes the internship project carried out at Tomas Bata University in Zlin under the supervision of prof. Roman Senkerik. The internship had a single clear research question: when modern Large Language Models are used as the creative engine of an algorithm-design loop, how do the leading frameworks for that idea actually compare on the same benchmark, the same model, and the same machine? The deliverable is a reproducible benchmark harness that puts three such frameworks - LLaMEA, LLM4AD, and frontEASE - side by side on a fixed set of optimization tasks, plus the experimental evidence gathered with it.

The realization document is structured to mirror how the project was actually delivered. Chapter 1 explains the business case and walks through every up-front decision (frameworks, tasks, LLM backend, implementation language) using the Weighted Ranking Method, so each decision is auditable, including a short description of each framework under test. Chapter 2 describes the system that was built, at the architectural level, with one short code excerpt where it makes the design clearer. Chapter 3 lists the functionalities that the system delivers - what it can actually do today. Chapter 4 presents the experimental findings through twenty-four figures in Chapter 4, ordered as a story: how long the experiment took, how often frameworks failed, when those failures appeared over time, and finally the objective results. Chapter 5 summarises the end point of the internship and how the project can be used after the internship ends, and Chapter 6 lays out the concrete future steps. The remainder of the document is the evaluation, a short conclusion, the reference list, the generative-AI acknowledgment required by the course policy, and an attachments overview.

1. Analysis

1.1 Business case: why benchmark LLM-driven algorithm design

Automated Algorithm Discovery (AAD) is a fast-moving research direction that uses Large Language Models to write, mutate, and refine algorithm source code under the guidance of an evaluation signal. The appeal for industry and research labs is direct: instead of paying a specialist weeks to hand-tune a heuristic for a particular optimization problem, the lab points an LLM-driven framework at a scoring function and lets the loop discover a competitive algorithm autonomously. Multiple research groups have published frameworks that claim this capability, and the AI Lab at Tomas Bata University wanted an independent, in-house answer to a simple question that the published papers do not answer in a comparable way: under identical conditions, which framework gives the most reliable, lowest-cost results, and is the result stable enough to put behind a research workflow?

Answering that question for the lab is the business case of this internship. The output of the project is not a one-off paper; it is a reusable benchmarking harness that any future colleague can rerun on a new LLM, on a new framework, or on a new task with one configuration file. That makes the harness an asset that keeps producing value after the internship: every time the team upgrades the LLM backend, or a new AAD framework is published, the harness can place it in context against the existing evidence within hours rather than weeks.

1.2 The three frameworks under test

1.2.1 LLaMEA (Large Language Model Evolutionary Algorithm)

LLaMEA [1] is an LLM-native evolutionary framework in which a population of Python algorithm candidates is created, mutated, and selected in a loop driven by a Large Language Model. The LLM both proposes new code and critiques existing candidates against the task description. The design goal is to discover competitive heuristics without hand-written operators: evolution happens in program space rather than in a fixed parameter vector.

In this benchmark, LLaMEA was the fastest framework per trial in median wall-clock time and among the most reliable on the simple Sphere landscape, but its useful-output rate dropped on the multimodal Rastrigin and Rosenbrock tasks compared with LLM4AD. That profile fits an LLM loop that converges quickly when the landscape is forgiving but sometimes exhausts its search budget before producing a harness-valid candidate on harder tasks.

1.2.2 LLM4AD — Evolution of Heuristics (EoH)

LLM4AD [2] is a broader toolkit for LLM-driven algorithm design; this project uses its Evolution of Heuristics (EoH) method [3], which is the entry point most comparable to LLaMEA in spirit. EoH maintains a population of heuristic programs, uses the LLM to generate and refine them, and scores each candidate by running it on the task. The loop emphasises structured exploration of heuristic ideas with a relatively large per-trial LLM budget.

In the self-evolving experiment, LLM4AD achieved the highest useful-output rates on all three continuous tasks and the strongest headline objective medians on valid runs, at the cost of roughly twice the median wall-clock time of LLaMEA per trial. For an overnight unattended benchmark where reliability matters more than saving a few minutes per lap, that trade-off was favourable.

1.2.3 frontEASE

frontEASE [4] is an evolutionary-algorithm platform developed at Tomas Bata University. Unlike the two LLM-native frameworks, frontEASE grew from a classical evolutionary-computation pipeline (web UI, .NET REST API, FoP-IMT Python Core) and later gained an LLM-as-operator branch. The benchmark harness drives it headlessly through the REST API and polls the Python Core until each trial finishes.

In the final continuous triplet run (Chapter 4), frontEASE was the slowest per trial in median wall-clock time but produced harness-valid objectives on roughly 86 percent of attempts overall — comparable to LLaMEA on Sphere and slightly ahead on the multimodal tasks in that metric, though still behind LLM4AD. A minority of trials still report success with objectives the harness rejects (out-of-bound or missing values); Chapter 4 separates that reliability gap from raw runtime cost.

1.3 Decision: which three frameworks to compare

The supervisor predefined the three frameworks above because together they cover two LLM-native code-evolution traditions and one established evolutionary platform with an LLM operator. An analytical comparison was still performed to anticipate integration effort. The criteria below are the ones that matter for an unattended, reproducible benchmark: API maturity, documentation quality, LLM-native design, ease of running inside an isolated process, and fidelity of best-objective reporting.

How to read this table: *the first column lists each evaluation criterion together with its weight in parentheses (1 = nice-to-have, 5 = non-negotiable). The remaining columns show each candidate's per-criterion score on a 1-10 scale, expressed as 'raw score -> weighted contribution' (raw x weight). The last row sums the weighted contributions into the final WRM total; the highest total wins.*

Criterion (weight 1-5)	LLaMEA	LLM4AD (EoH)	frontEASE
API maturity (4)	8 -> 32	7 -> 28	6 -> 24
Documentation (3)	7 -> 21	7 -> 21	6 -> 18
LLM-native design (5)	9 -> 45	9 -> 45	6 -> 30
Process-isolation friendliness (5)	8 -> 40	8 -> 40	5 -> 25
Reporting fidelity (5)	8 -> 40	7 -> 35	6 -> 30
Weighted total	**178**	**169**	**127**

Table 1 - Weighted Ranking Method for the three target frameworks.

All three frameworks remain in scope because the assignment is explicitly comparative. The WRM does not eliminate any candidate; it correctly predicts the operational profile observed later - the two LLM-native frameworks integrate cleanly, while frontEASE needs an extra adapter layer to bridge its .NET REST API to its Python Core executor.

1.4 Decision: which tasks to benchmark on

The harness covers two task families. The primary family is the canonical continuous black-box optimization (BBO) trio in five dimensions [5]: Sphere [11], Rastrigin [12], and Rosenbrock [13], each on the cube [-5, +5]^5. These three landscapes are the textbook stress test for a generic optimizer because they exercise three different failure modes (unimodal convex, highly multimodal, elongated curved valley) and all share a known global minimum of zero. That hard lower bound is important: it lets the harness automatically reject any framework that reports a negative objective on a sum-of-squares landscape, which - as Chapter 4 shows - turned out to be the single biggest reliability issue in the experiment.

A second, smaller family of programmatic tasks (a 2048-board heuristic, a coding task scored by hidden unit tests, and a writing task scored by a keyword vocabulary) was added so that the harness is not limited to vector-valued objectives. The programmatic tasks share the same harness contract as the continuous tasks (one Python function in the generated source) but use task-specific evaluators. They are demonstrated as an attachment to the harness rather than as headline experimental evidence, because their sample sizes inside the internship window were too small for statistical comparison.

1.5 Decision: which LLM and how to host it

Every framework in scope accepts an OpenAI-compatible HTTP endpoint, so the question was not which API specification to use but where to host the model. Three candidate backends were considered: a managed cloud provider (OpenAI), a self-hosted vLLM server, and a self-hosted Ollama server [6]. The criteria that matter for this benchmark are cost during long unattended runs, model-version reproducibility (a vendor silently upgrading its model would invalidate the comparison), throughput stability, integration ease across all three frameworks, and independence from the shared university network.

How to read this table: *same layout as Table 1 - criteria with weights in the first column, candidates across the top, cells show raw score (1-10) -> weighted contribution, and the bottom row is the WRM total.*

Criterion (weight 1-5)	OpenAI cloud	vLLM (self-hosted)	Ollama (self-hosted)
Cost for long runs (5)	3 -> 15	8 -> 40	8 -> 40
Model-version reproducibility (5)	4 -> 20	9 -> 45	9 -> 45
Throughput and stability (4)	9 -> 36	8 -> 32	7 -> 28
Ease of integration (4)	9 -> 36	7 -> 28	8 -> 32
Network independence (3)	3 -> 9	8 -> 24	9 -> 27
Weighted total	**116**	**169**	**172**

Table 2 - Weighted Ranking Method for the LLM hosting backend.

Ollama won on the combination of zero per-token cost during overnight runs, locked model versions (the model file is pulled once and reused), and one-line operational ergonomics. The model chosen was

llama3.1:8b [7], served on the lab subnet. The same endpoint is consumed by all three frameworks, so any quality gap observed between them is attributable to the framework rather than the model.

1.6 Decision: which language implements the benchmark harness?

Sections 1.3-1.5 chose which frameworks, tasks, and LLM backend to compare. Section 1.6 answers a separate question: in which programming language should the comparison infrastructure itself be written? That infrastructure is the benchmark harness — the runner that schedules trials, the adapters that talk to LLaMEA, LLM4AD, and frontEASE, the sandbox that executes generated code, and the scripts that export figures. It is not the language in which the frameworks generate algorithms (that is always Python in this project); it is the language of the measurement apparatus around them.

Python was the natural choice because both LLaMEA [1] and LLM4AD [2] ship as Python packages, candidate evaluation is Python executed in a subprocess, and the analysis stack uses pandas [20], SciPy [21], and matplotlib [22]. C# / .NET 8 was scored only because frontEASE [4] exposes a REST API written in .NET; the harness still calls it over HTTP and contains no C# source. Rust was included as a modern systems-language baseline. Table 3 records the Weighted Ranking Method scores.

How to read this table: *Table 3 compares languages for implementing the harness only. First column: criterion and weight (1-5). Other columns: Python 3.11, C# / .NET 8, and Rust. Each cell is raw score (1-10) -> weighted points. The bottom row is the total; Python wins because it matches the framework APIs and the scientific-Python toolchain.*

Criterion (weight 1-5)	Python 3.11	C# / .NET 8	Rust
Match with framework APIs (5)	9 -> 45	4 -> 20	4 -> 20
Scientific-Python ecosystem (5)	9 -> 45	5 -> 25	5 -> 25
Type-safety for harness code (3)	6 -> 18	9 -> 27	9 -> 27
Compile / iteration speed (3)	9 -> 27	7 -> 21	5 -> 15
Ease of unattended scripting (4)	9 -> 36	7 -> 28	6 -> 24
Weighted total	**171**	**121**	**111**

Table 3 - WRM: language for the benchmark harness (not the frameworks under test).

2. How the system was built

This chapter describes the system at the architectural level. It is deliberately not a step-by-step setup guide - the repository's README files cover that role. The intent here is to show the shape of the solution and the design choices behind it.

2.1 Architecture at a glance

The harness is a config-driven Python pipeline organised in three horizontal layers. The top layer is a thin driver that reads a JSON configuration, expands it into a list of (framework, task, seed) trials, and dispatches each trial to a worker. The middle layer is a common adapter contract that every framework integration implements; this layer is the abstraction that makes the three frameworks comparable. The bottom layer is the result store, an append-only collection of per-run JSON files plus a heartbeat file that the in-browser status dashboard reads to show progress live.

Each individual trial runs in an isolated subprocess that is killed if it exceeds a per-trial timeout. The subprocess returns its result through a small JSON file rather than through standard output, so a crash in the candidate code that the LLM produced cannot poison the harness. Resume semantics are inherent in the design: if the harness is interrupted, the next launch reads the existing per-run files and schedules only the missing (framework, task, seed) combinations.

2.2 The adapter layer

The adapter contract is intentionally minimal. Each adapter knows how to take a task definition plus a seed, drive its framework's discovery loop with the shared LLM endpoint, and return a single structured result describing whether the run succeeded, the best objective achieved, the elapsed time, and any error code that explains a failure. Putting just these five pieces of information in the contract is what makes the three frameworks comparable: the rest of the harness never has to know how each framework is implemented internally. The shape of the contract is shown below as a short Python excerpt.

```
class BaseAdapter(ABC):
    @abstractmethod
    def run(self, task: Task, seed: int,
            llm_config: LLMConfig) -> TrialResult: ...

@dataclass
class TrialResult:
    success: bool
    best_value: float | None
    elapsed_sec: float
    error_code: str | None
    framework_artefacts: dict
```

Three concrete adapters live behind this contract: one for LLaMEA [1], one for LLM4AD's Evolution of Heuristics method [3], and one for frontEASE [4]. The frontEASE adapter is the most involved because it has to start the platform's .NET REST host, register the task, and poll the FoP-IMT Python Core for the lifecycle of the run before returning a result through the same TrialResult shape.

2.3 Sandboxing the code that the LLM writes

The LLM-discovered algorithm is, by definition, code that has not been reviewed by a human. The harness therefore treats every candidate as untrusted input and applies two layers of protection [8]. The first layer is static: a small AST-based validator scans the generated source for the operations that have no place in a numerical optimizer (network access, subprocess spawning, file deletion, shell-out helpers). If the validator finds a forbidden import or call, the candidate is rejected before it ever runs. The second layer is dynamic: the candidate is executed inside a dedicated subprocess with a strict wall-clock timeout, so an infinite loop or a runaway memory allocation cannot freeze the benchmark.

This pairing is what allows the harness to be left running overnight on a workstation. Without the static layer a malicious or accidental shell-out could disturb the host; without the dynamic layer a single non-terminating candidate could halt the entire night's experiment.

2.4 The analysis pipeline

Producing the figures in Chapter 4 is a deliberately separate pipeline. The runner only writes raw per-trial JSON files; the analysis layer reads those files into a pandas dataframe, normalises the columns across frameworks, applies the bound-check that distinguishes a useful result from a self-reported success that violates the task's known lower bound, and exports the figures used in the report. Keeping the two stages separate is the reason the experiment is fully reproducible: nothing inside the runner ever transforms or aggregates the raw data, so a re-analysis of the same JSON files always yields the same figures.

3. What the project can do

This chapter is a feature-level description of the delivered system - what an operator can ask the harness to do today, without writing new code. It is the catalogue that a future user (or a future internship student picking up the project) needs to know in order to use the harness productively.

3.1 Core capabilities

How to read this table: *the first column is the name of a capability the harness exposes today; the second column states what that capability does and what data or process it touches.*

Item	Description
Multi-framework benchmarking	Runs LLaMEA, LLM4AD (Evolution of Heuristics), and frontEASE on exactly the same task definitions, the same seeds, the same LLM endpoint, and the same per-trial timeout.
Two task families	Continuous black-box optimization (Sphere, Rastrigin, Rosenbrock in 5D) and programmatic tasks (a 2048 heuristic, a coding task, a writing task). Tasks are registered through a small registry so adding a fourth task is a single edit.
Config-driven experiments	Each experiment is defined by one JSON file describing the frameworks, tasks, seeds, LLM settings, timeouts, and output directory. Environment variables override any field at launch time so the same JSON can drive different machines.
Resume after interruption	Result storage is append-only. If the harness is killed mid-run the next launch reads the existing per-run JSON files and schedules only the trials that are still missing.
Live status dashboard	A small auto-refreshing HTTP server reads the heartbeat file and displays the current trial, the elapsed time, the cumulative success rate per framework, and the per-trial log location.
Sandboxed candidate execution	Every LLM-generated algorithm is first scanned by an AST validator for forbidden operations and then executed in a subprocess with a wall-clock timeout.
Statistical analysis notebook	A Jupyter notebook ingests the per-run JSON files, applies the comparison protocol, and exports the figures used in Chapter 4.
Reproducible outputs	Raw data, configuration, code, and analysis pipeline are all committed to the repository, so any figure in Chapter 4 can be regenerated by re-running the notebook on the same per-run directory.

Table 4 - Functionalities currently provided by the harness.

3.2 Typical operator workflow

An operator picks or writes a JSON configuration that selects the frameworks, tasks, and seed pool they want, points the LLM endpoint at their preferred Ollama (or any OpenAI-compatible) backend, and launches the runner. The status dashboard updates live as trials complete; if the operator needs to free the machine

for something else, they kill the runner and resume later. Once the run is complete (or once an in-progress run has gathered enough data), the operator runs the analysis notebook against the per-run directory and obtains the figures of Chapter 4 within seconds.

4. Findings

This chapter presents the self-evolving experiment in the order a reader would naturally ask the questions: how long did the runs take, how often did each framework fail, when did those problems show up over wall-clock time, and only then - on the trials that produced usable output - how good were the objectives? The evidence base contains 3,242 JSON trial records; the figures use a balanced 3,240-row subset, with 360 attempts per framework and task.

The experiment window spans about eight calendar days of unattended continuous execution (18 May through 26 May 2026 in the recorded timestamps). The chapter is deliberately not a catalogue of plots. Act I (Sections 4.1-4.2) establishes operational cost and failure rates. Act II (Section 4.3) places failures and quality drift on a timeline. Act III (Section 4.4) reports objective quality only after the reader understands which runs were trustworthy enough to compare.

4.1 How long did the experiment take?

Before interpreting objectives, the operator needs to know wall-clock cost. Figure 1 shows the distribution of elapsed seconds per trial for each framework (all runs in the merged dataset). The y-axis is logarithmic so that a six-hundred-second lap and a sixty-second lap remain visible on one chart.

The medians tell a clear operations story: LLaMEA is the cheapest per trial, LLM4AD is roughly twice as expensive because Evolution of Heuristics spends more LLM budget per lap, and frontEASE is an order of magnitude slower because every trial pays the polling overhead of the FoP-IMT Python Core lifecycle. Any overnight schedule must budget for these differences before chasing a lower median objective.

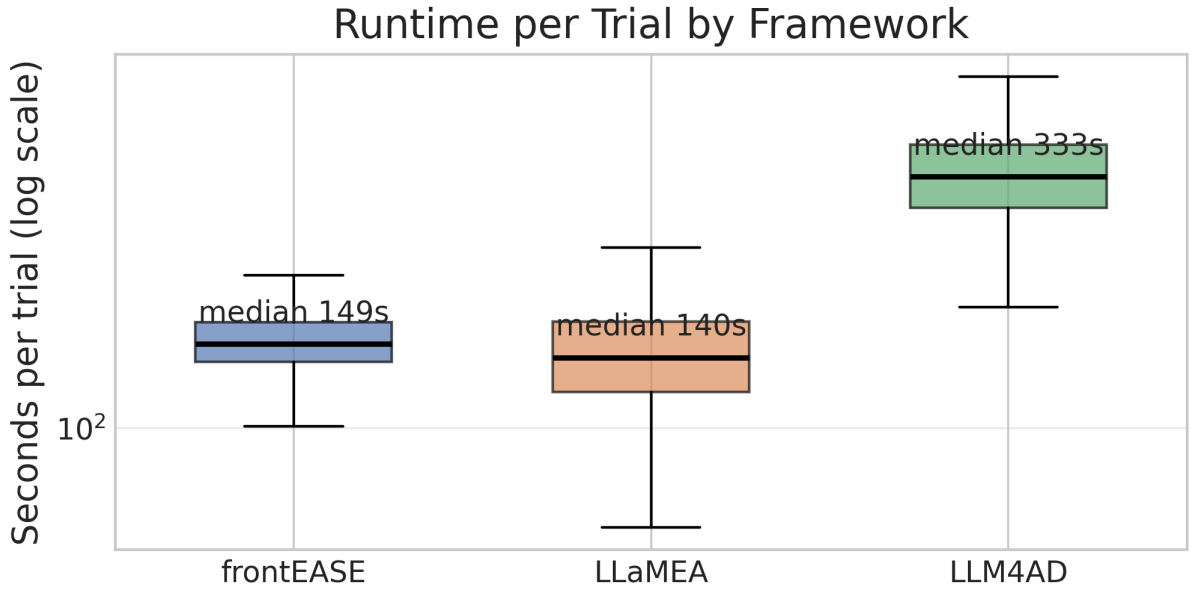

Figure 1 - Elapsed wall-clock time per trial per framework.

4.2 How often did each framework fail?

Figure 2 states the complementary view of failure: the harness-valid rate per (framework, task) with Wilson 95% confidence intervals [14]. Valid means the run finished and the reported best objective passed the bound-check (finite, non-penalty, not below the known minimum of zero). Everything below 100% is, in practice, a failed or unusable trial from the benchmark's point of view.
Figure 3 breaks down why trials were not valid: timeouts, declared errors, invalid objectives, or success with a metric the harness rejects. Together, the two figures answer 'how often' and 'why'. frontEASE [4] still shows a visible stack of harness-rejected objectives on harder tasks, but no longer at zero percent valid; LLM4AD [2] remains the most reliable overall (about 93 percent harness-valid across the balanced table), with LLaMEA near 71 percent.

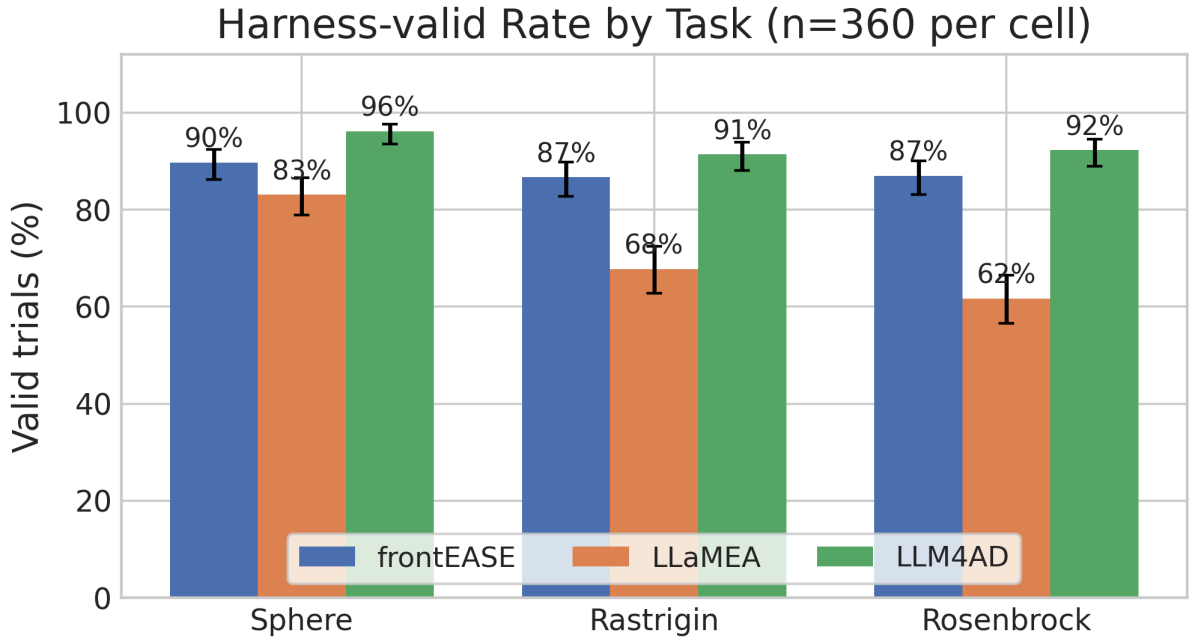

Figure 2 - Harness-valid rate per (framework, task) with Wilson 95% CIs (higher = fewer failures).

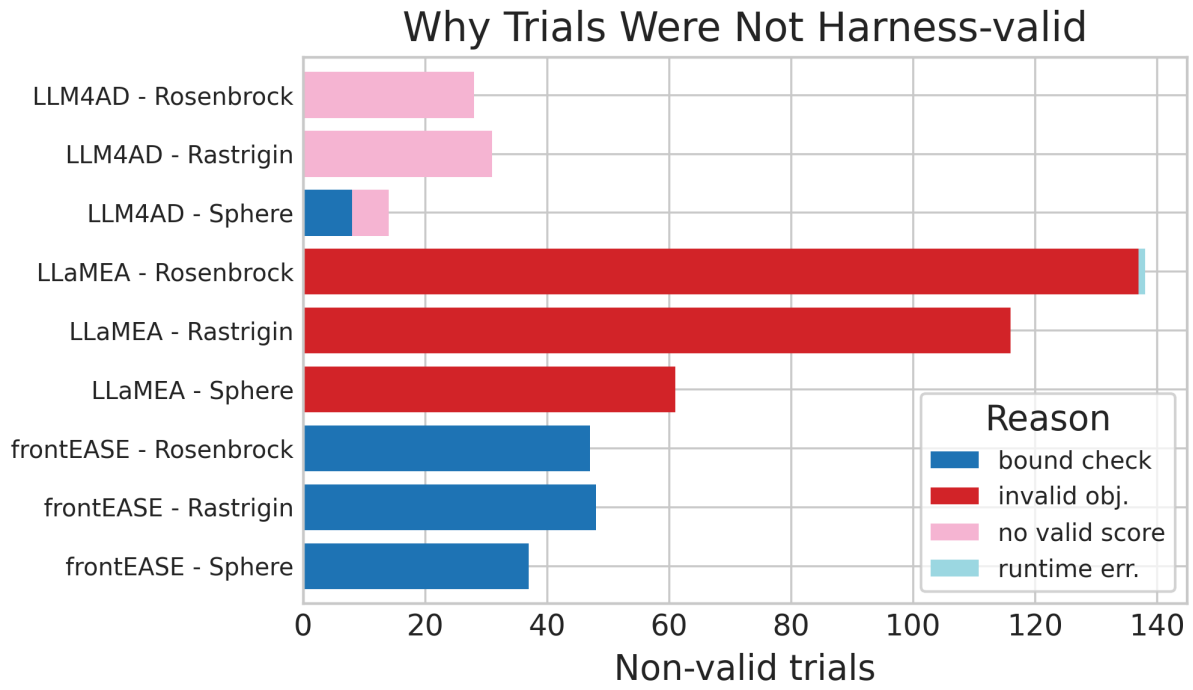

Figure 3 - Failure / invalid-objective taxonomy per (framework, task).

Figures 4a-4c repeat the reliability view for one landscape each so a single-task headline can be quoted without reading the grouped chart.

4.2.1 Sphere 5D

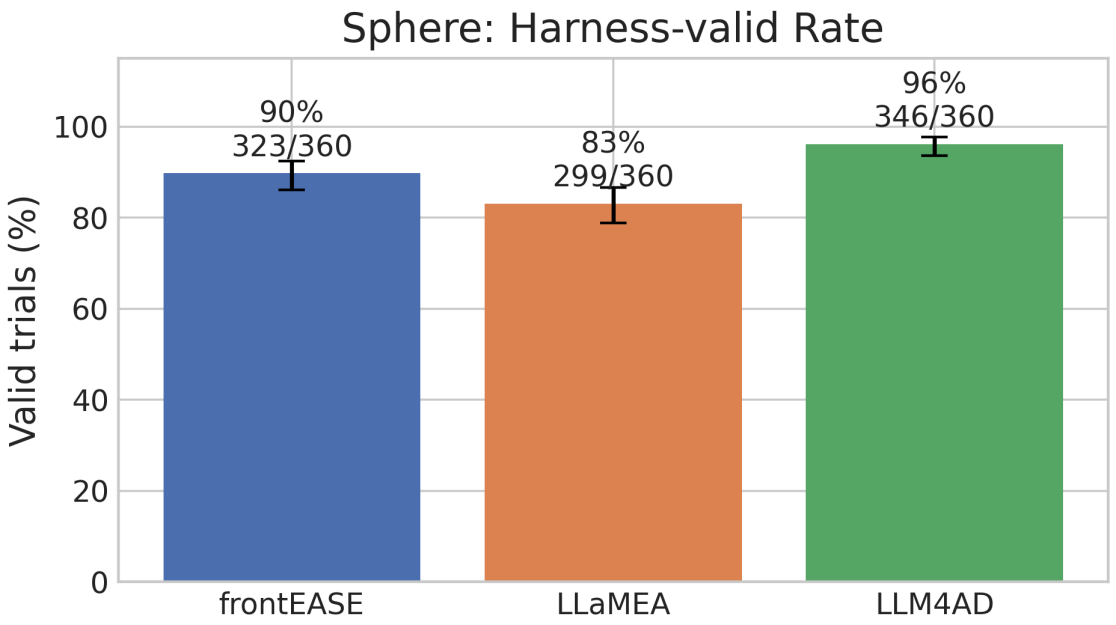

Figure 4a - Harness-valid rate on Sphere 5D.

4.2.2 Rastrigin 5D

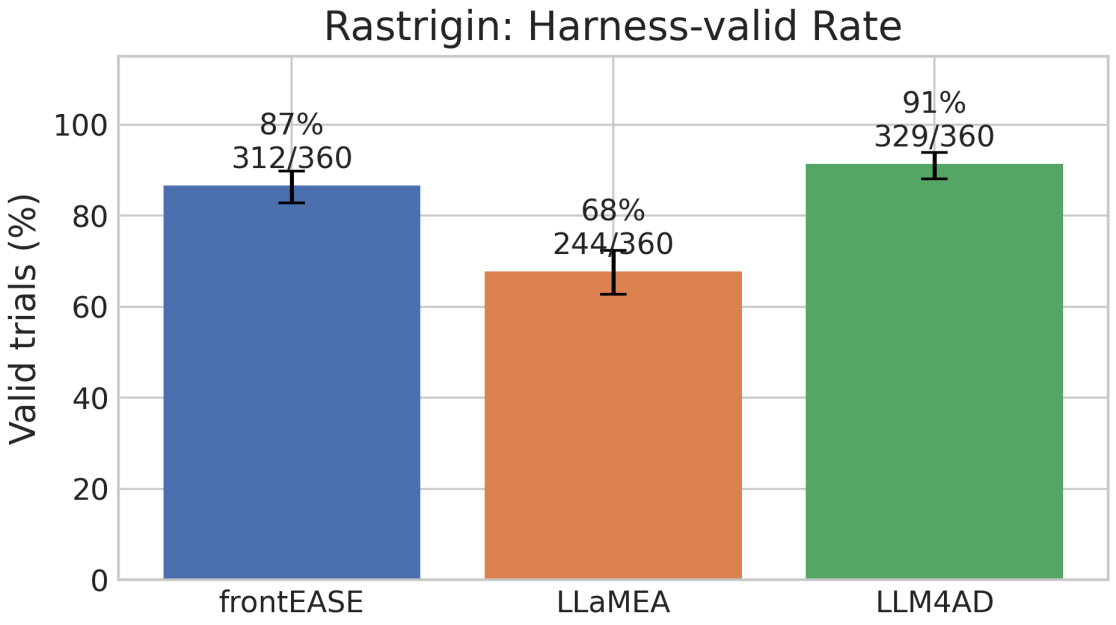

Figure 4b - Harness-valid rate on Rastrigin 5D.

4.2.3 Rosenbrock 5D

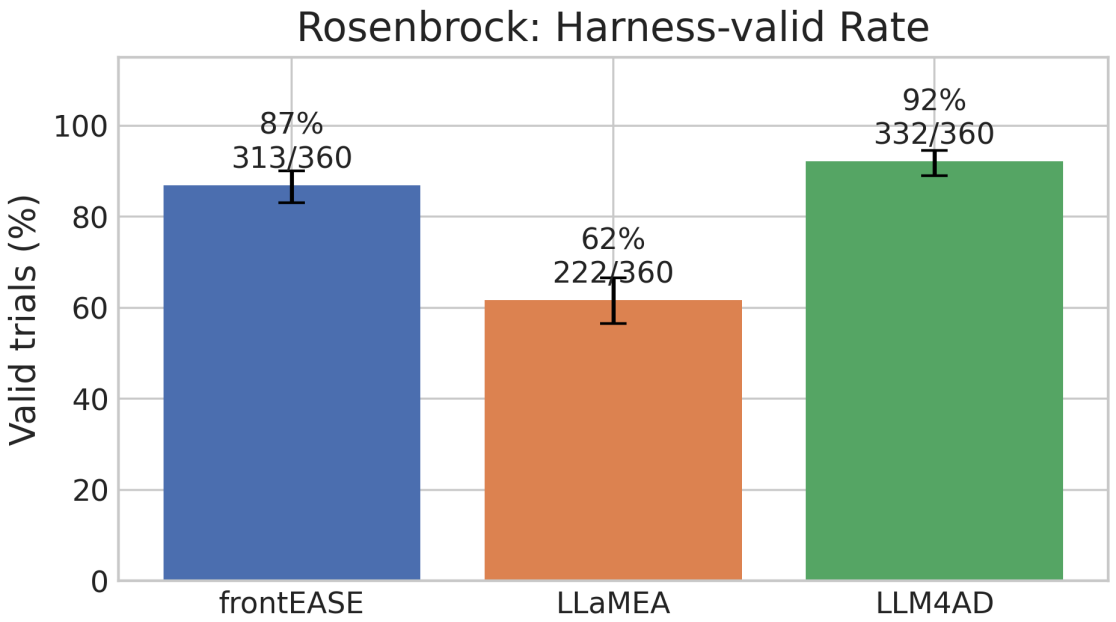

Figure 4c - Harness-valid rate on Rosenbrock 5D.

4.3 When did each framework struggle during the run?

Failure rate alone does not say whether problems were spread across the whole internship window or clustered in one bad night. Figures 5a-5c plot harness-valid trials against wall-clock completion time with a rolling median of best objective (fixed window in the notebook). Read them as a timeline: flat or drifting lines show whether a framework kept improving, plateaued, or produced occasional wild outliers late in the schedule.

These are the right plots for the 'self-evolving' claim in calendar time. A steep early drop followed by a flat line means the framework found a workable algorithm quickly and then mostly explored variations that did not beat it within the budget.

4.3.1 Sphere 5D

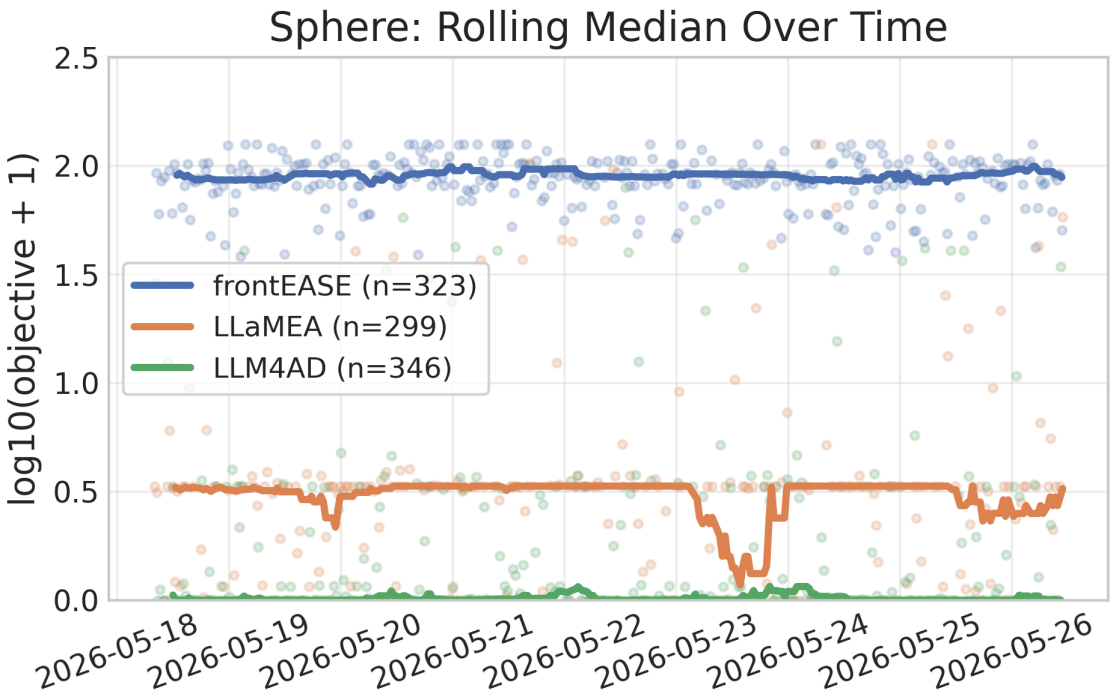

Figure 5a - Rolling median best objective over wall-clock time (Sphere).

4.3.2 Rastrigin 5D

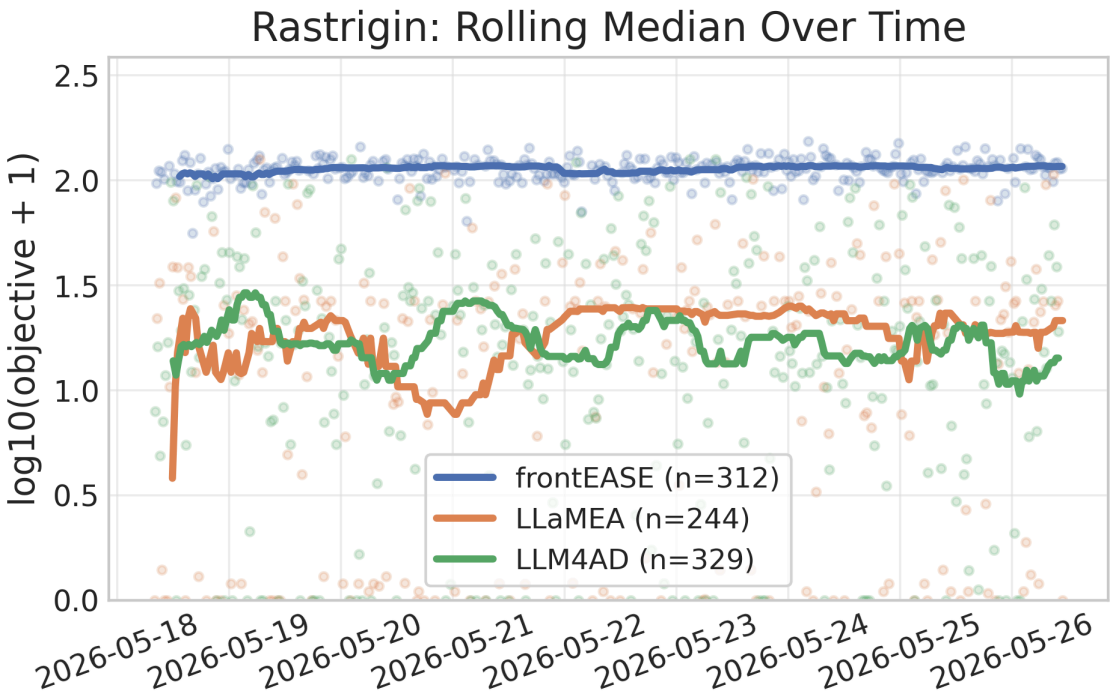

Figure 5b - Rolling median best objective over wall-clock time (Rastrigin).

4.3.3 Rosenbrock 5D

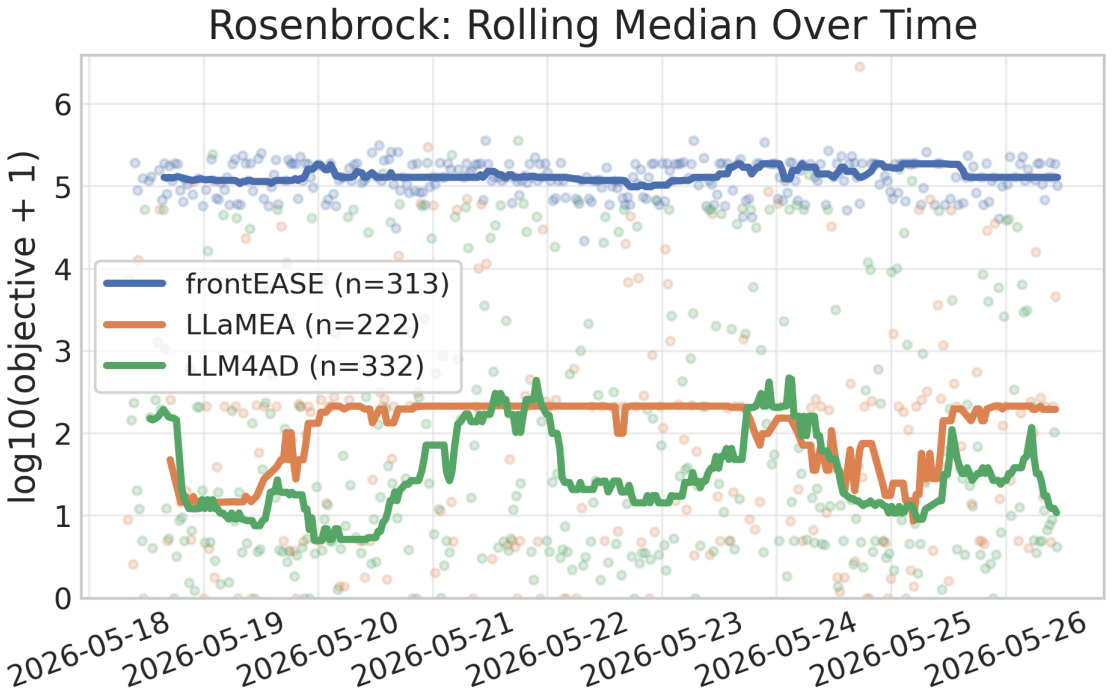

Figure 5c - Rolling median best objective over wall-clock time (Rosenbrock).

4.4 Results on harness-valid runs

Only after Sections 4.1-4.3 does it make sense to compare objective quality. The figures below filter to harness-valid runs so that out-of-bound reporting cannot win the benchmark by accident.

4.4.1 Sphere 5D — results

The Sphere landscape [11] is the simplest of the three: unimodal with global minimum zero. Figures 6a and 7a show the distribution and ECDF of the real best objective on harness-valid runs only. Axis numbers are the actual objective values reported by the framework (lower is better; optimum is 0). On Rastrigin and Rosenbrock the charts may use a log axis when values span many orders of magnitude, but the tick labels still refer to real objectives, not a signed-log transform.

Figures 8a-10a summarise median objectives, the share of runs that reach readable quality targets (fixed cut-offs and multiples of the best result on that task), and Holm-corrected pairwise tests [9] [15]. On Sphere, LLM-native frameworks cluster near numerical zero when they produce valid output.

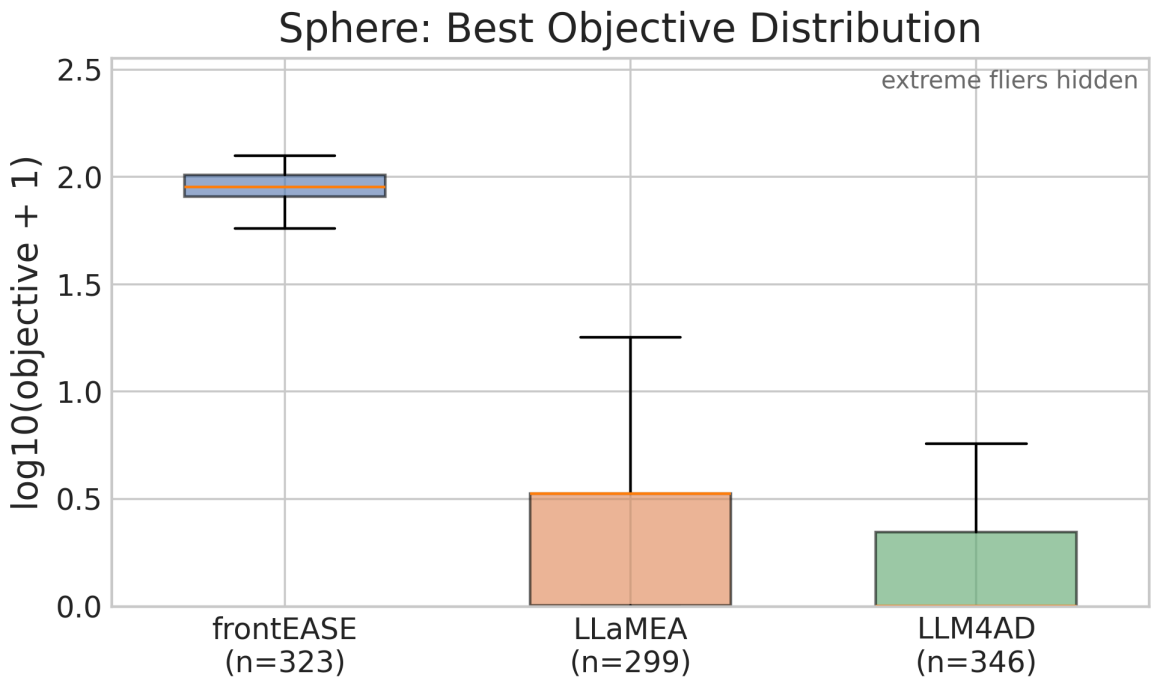

Figure 6a - Distribution of best objective on Sphere 5D (harness-valid runs).

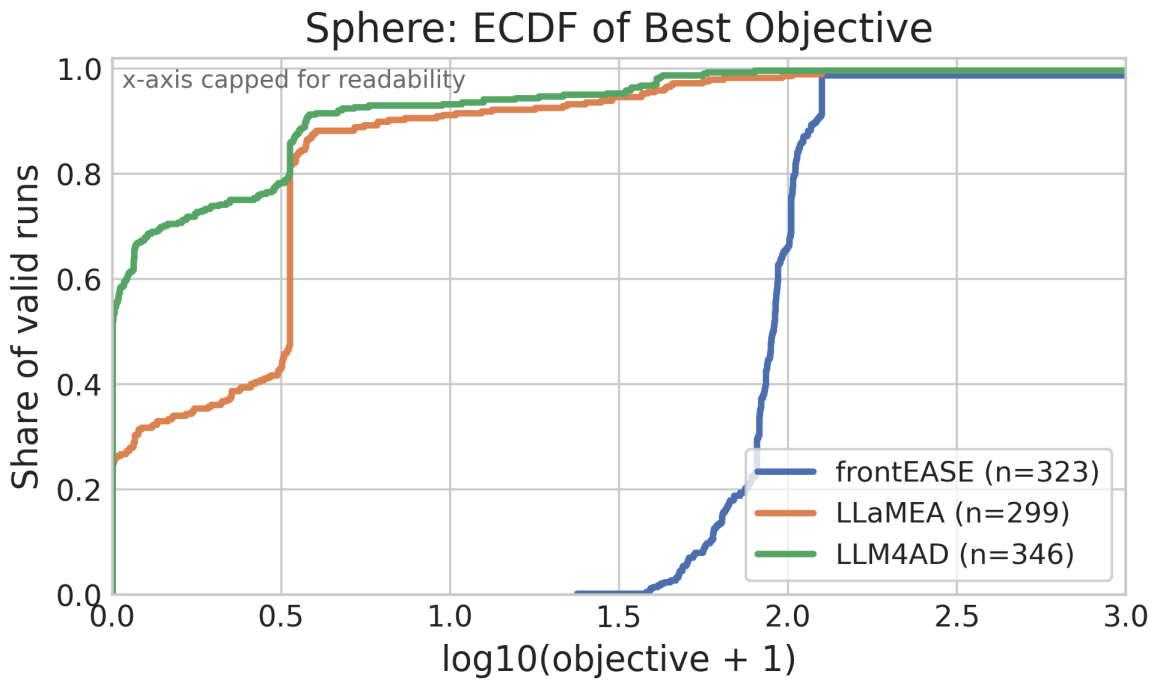

Figure 7a - ECDF of best objective on Sphere 5D.

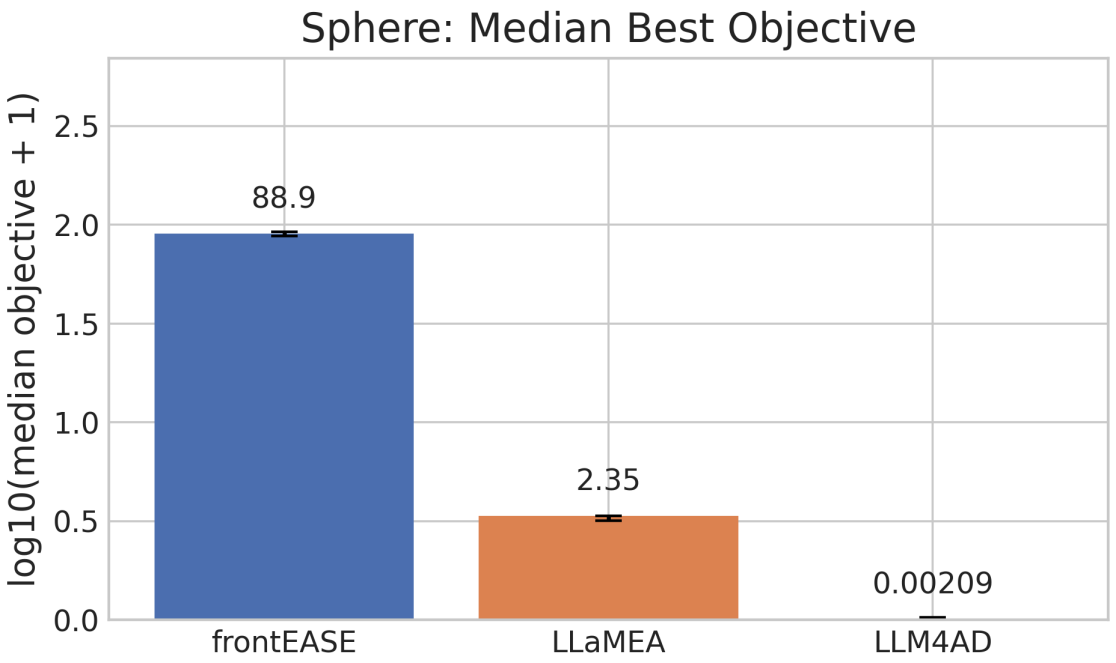

Figure 8a - Median objective with intervals (Sphere).

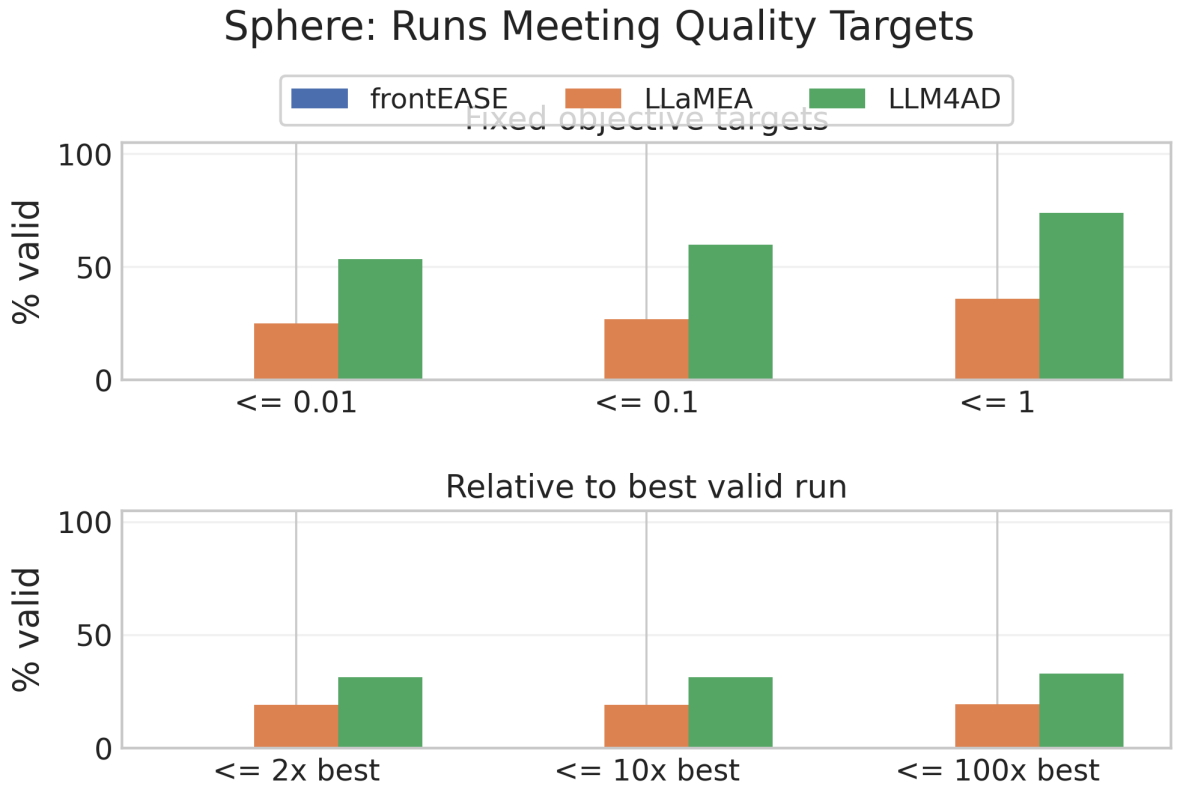

Figure 9a - Share of runs meeting objective quality targets (Sphere).

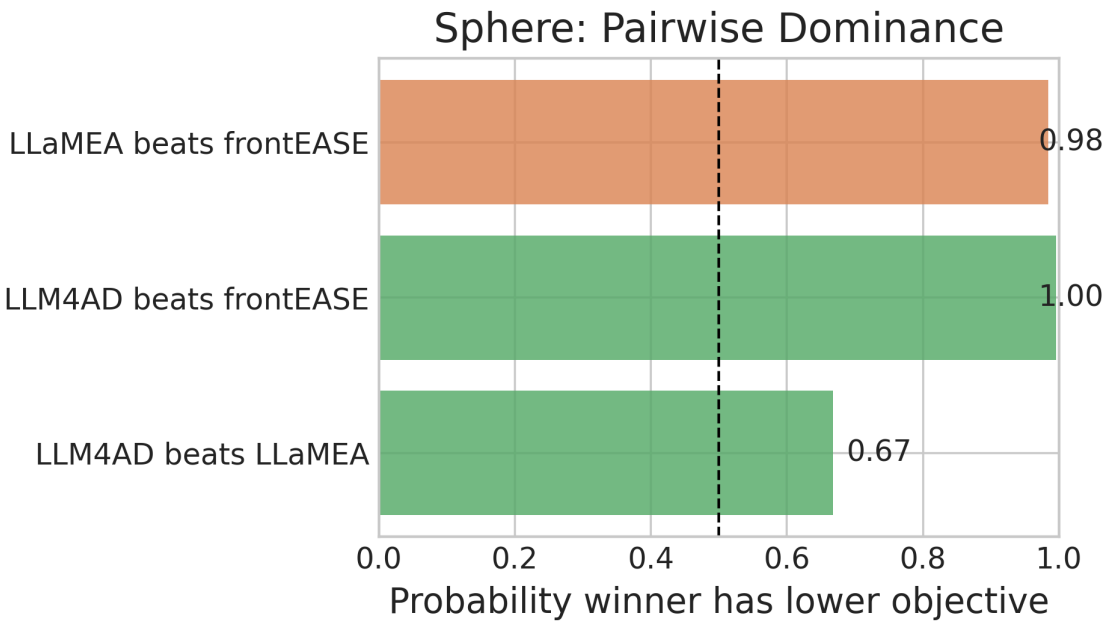

Figure 10a - Pairwise comparison bars (Sphere).

4.4.2 Rastrigin 5D — results

Rastrigin [12] is highly multimodal; valid-run counts differ more across frameworks than on Sphere (see Figure 4b), so read the failure sections before interpreting Figures 6b-10b. Figures 6b and 7b are the per-task boxplot and ECDF at full width for readability.

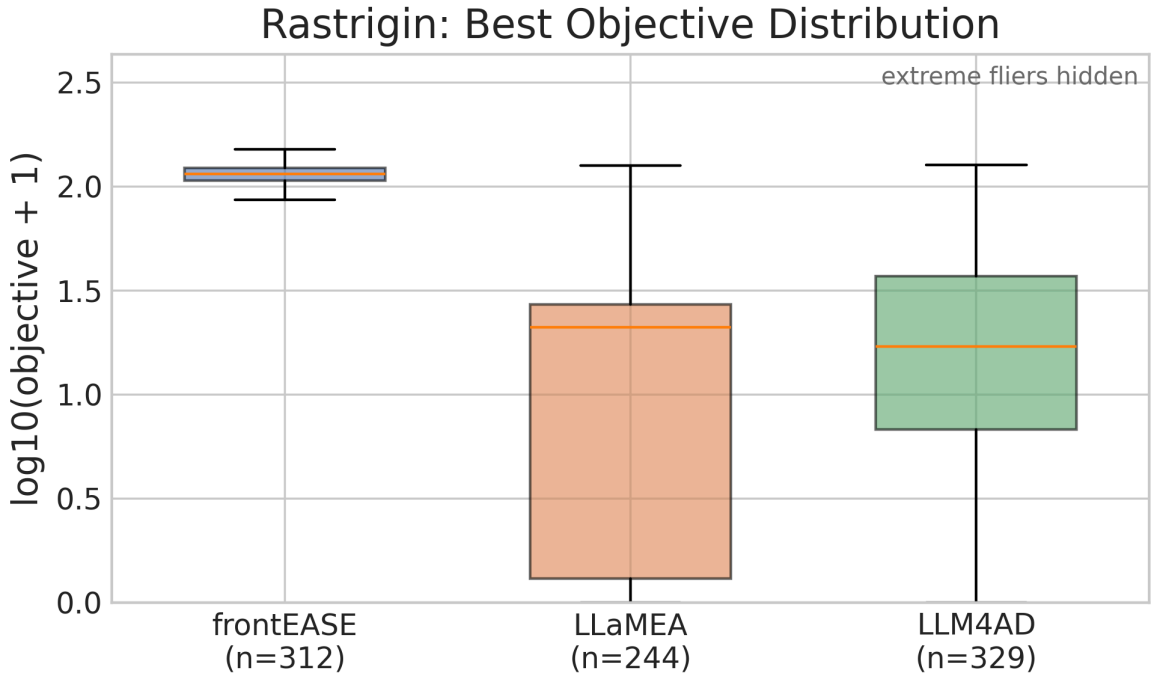

Figure 6b - Distribution of best objective on Rastrigin 5D.

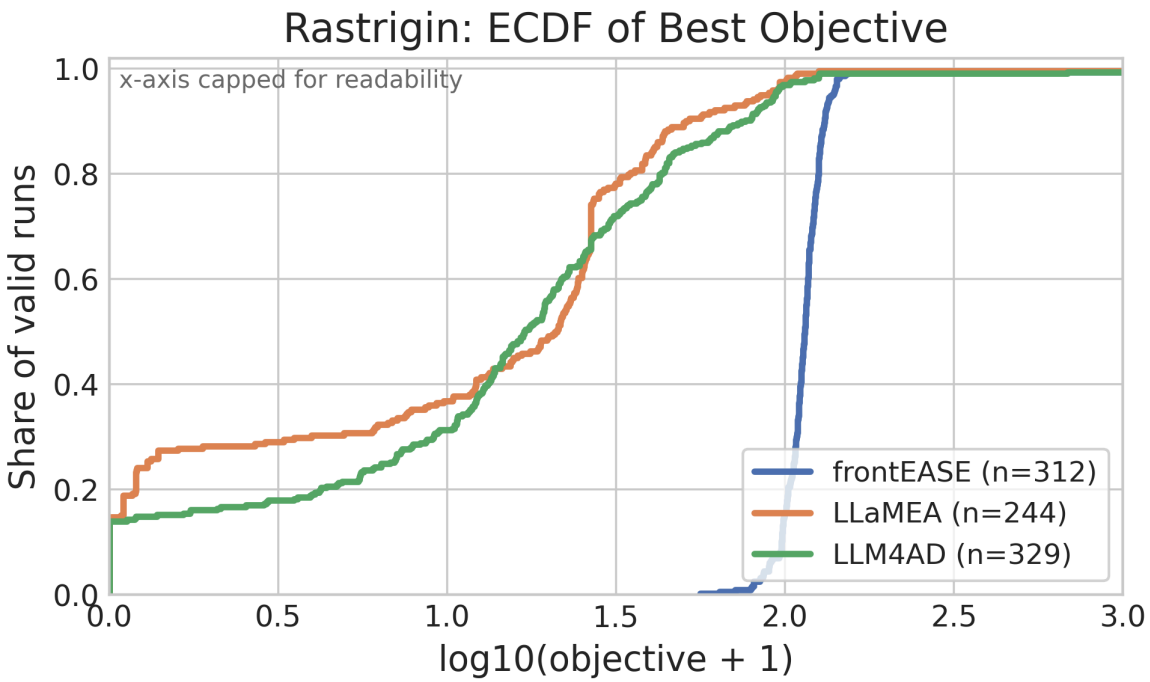

Figure 7b - ECDF on Rastrigin 5D.

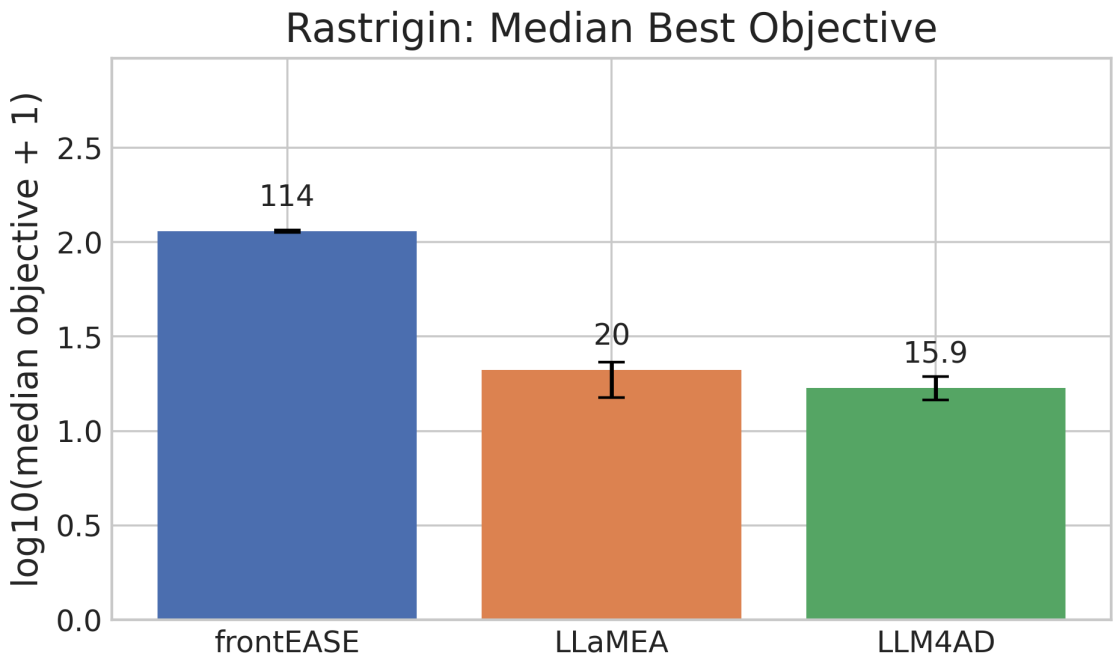

Figure 8b - Median objective with intervals (Rastrigin).

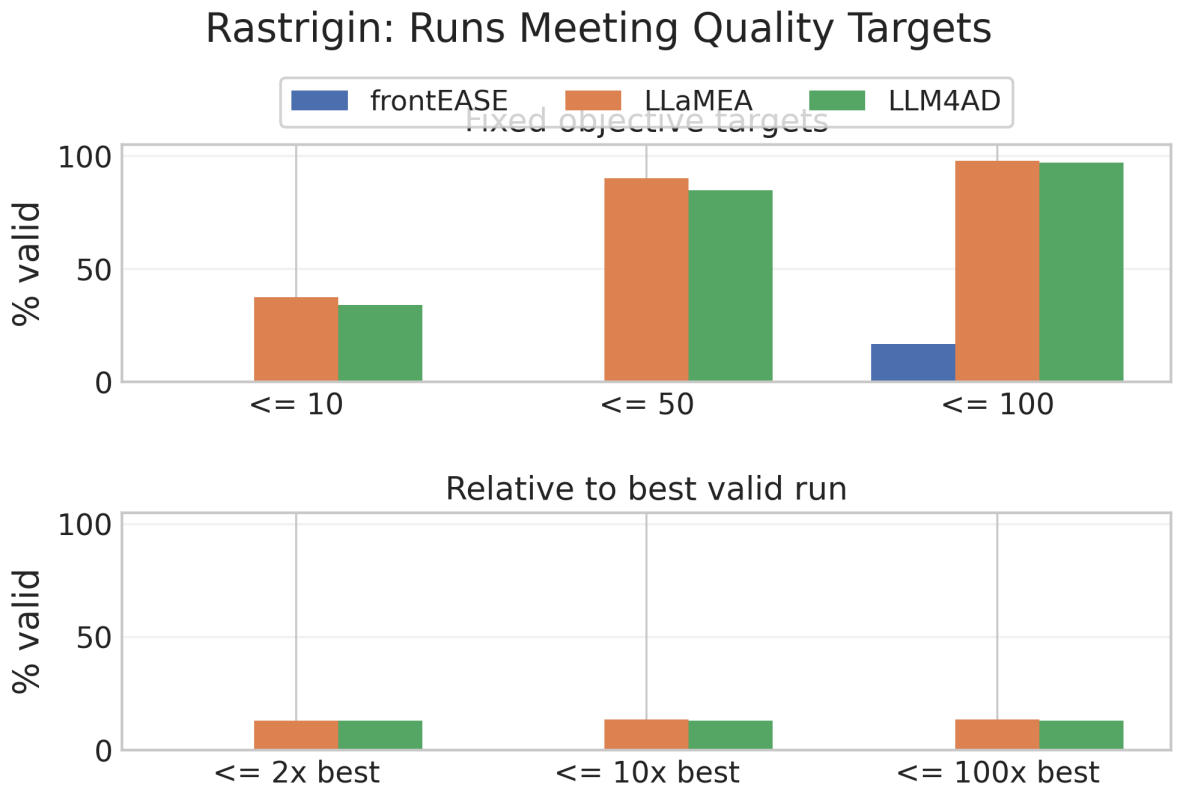

Figure 9b - Share of runs meeting objective quality targets (Rastrigin).

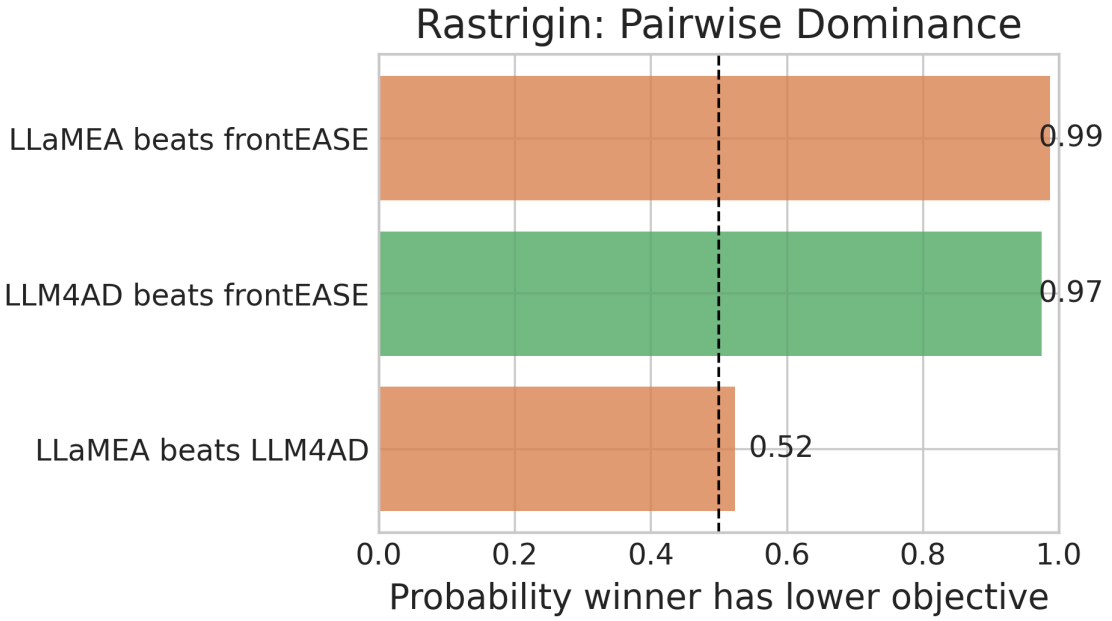

Figure 10b - Pairwise comparison bars (Rastrigin).

4.4.3 Rosenbrock 5D — results

Rosenbrock [13] has a narrow curved valley and was the hardest landscape for headline medians in this run. Figures 6c and 7c are the per-task distribution and ECDF; Figures 8c-10c close the chapter with median bars, quality-target bars, and pairwise comparison.

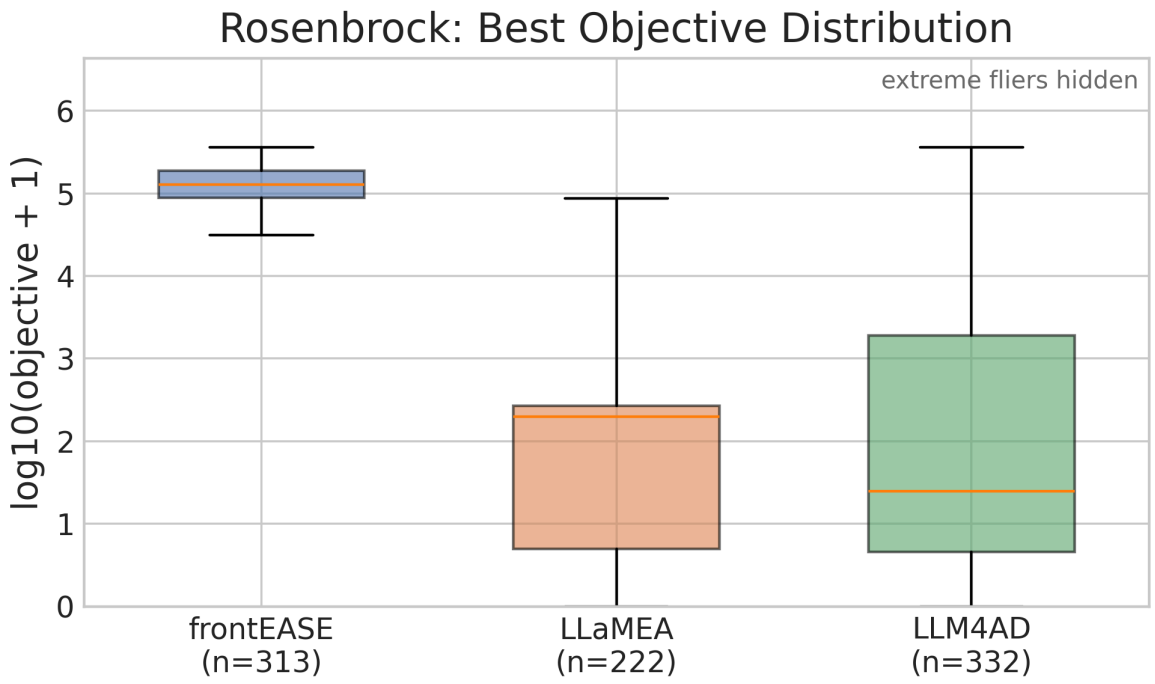

Figure 6c - Distribution of best objective on Rosenbrock 5D.

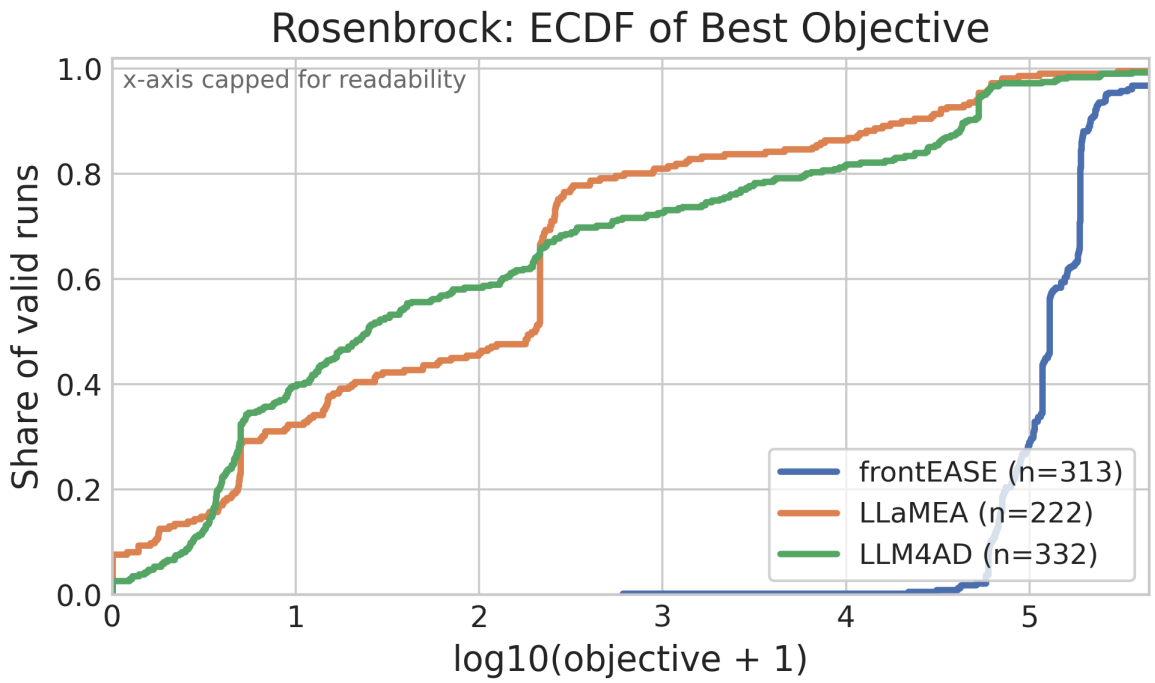

Figure 7c - ECDF on Rosenbrock 5D.

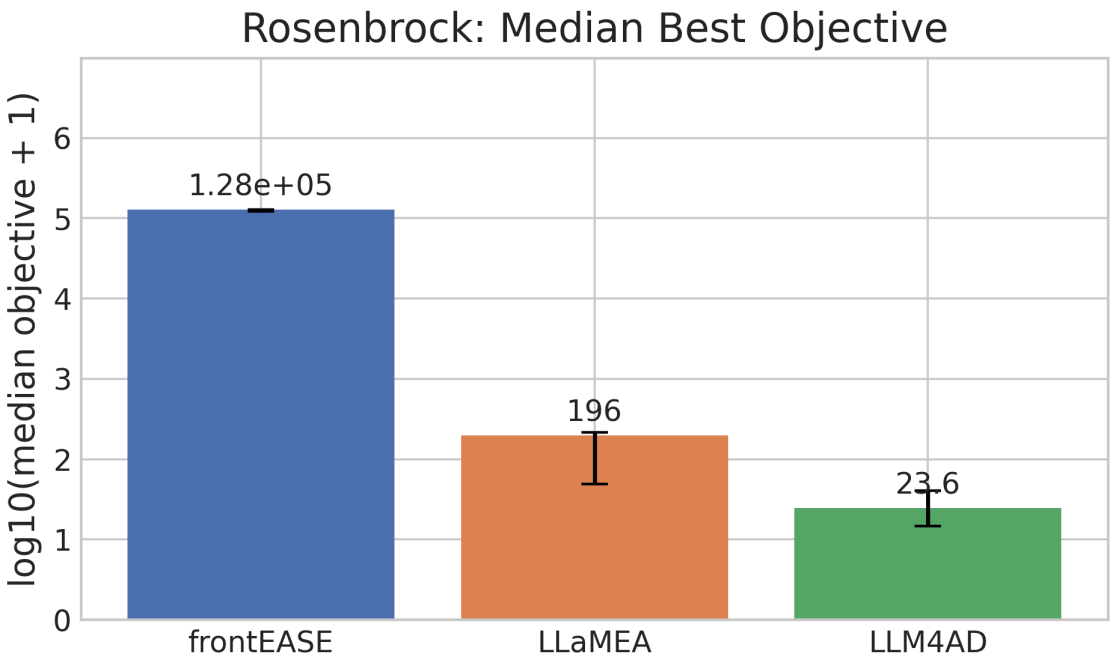

Figure 8c - Median objective with intervals (Rosenbrock).

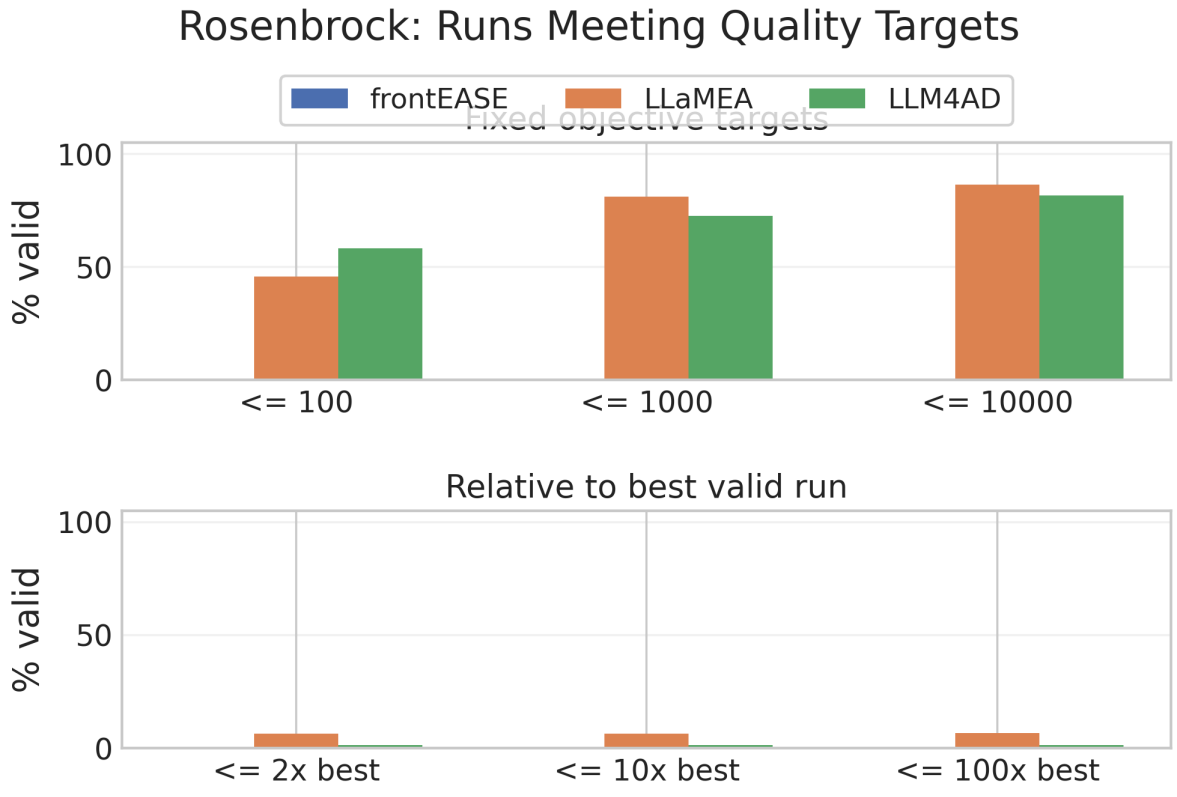

Figure 9c - Share of runs meeting objective quality targets (Rosenbrock).

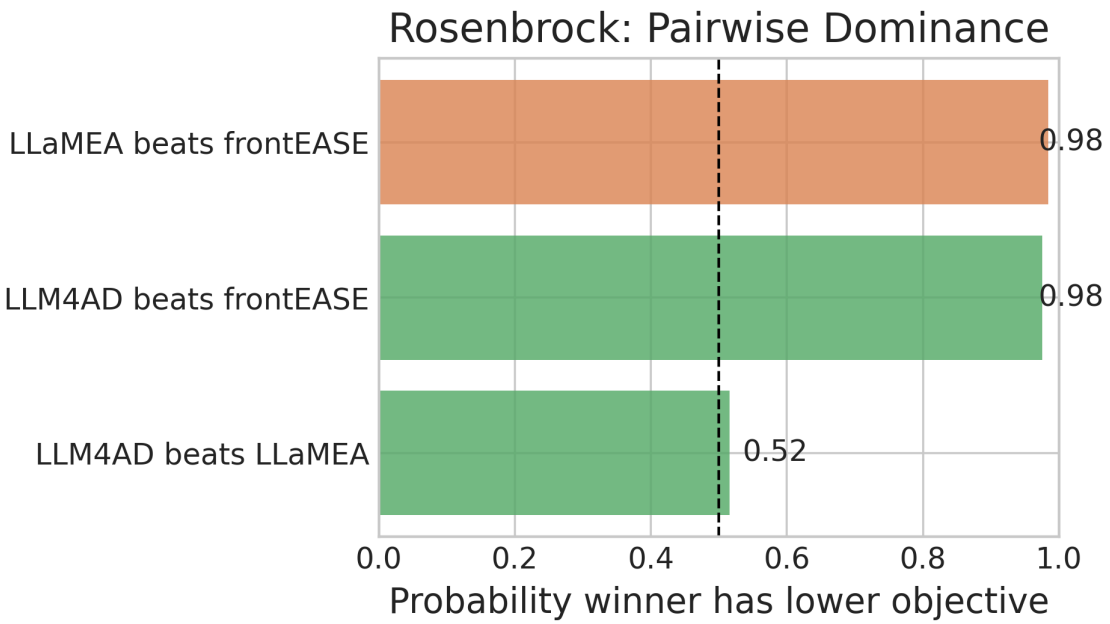

Figure 10c - Pairwise comparison bars (Rosenbrock).

5. End point and how the project can be used after the internship

5.1 What was delivered

The internship ends with a complete, documented, reproducible benchmarking system and the experimental evidence gathered with it. The end point is concrete: another researcher at the AI Lab can clone the repository, install the dependencies listed in the README, point the configuration at their Ollama backend, and rerun any experiment in this document with one command.

How to read this table: the first column names a deliverable; the second column states what is complete today and what, if anything, still belongs in future work.

Item	Description
Benchmark harness	Done: config-driven runner, adapter layer, sandbox, and result storage. Future work: add CI smoke tests and new framework adapters as needed.
Adapters	Done: one adapter per framework (LLaMEA, LLM4AD-EoH, frontEASE), each isolated behind the BaseAdapter contract. Future work: tighten the remaining frontEASE objective-reporting edge cases.
Configurations	Done: paper-oriented, continuous overnight, and self-evolving profiles are included. Future work: add profiles for stronger LLMs and larger task suites.
Raw evidence	Done: 3,242 per-trial JSON records from the final continuous triplet run. Future work: extend the evidence base when new models, frameworks, or tasks are tested.
Analysis	Done: the analysis notebook exports the figures and comparison tables used in Chapter 4. Future work: rerun it whenever a new evidence set is produced.
Operational tooling	Done: PowerShell launchers for unattended overnight runs and a small in-browser status dashboard. Future work: automate smoke checks before long runs.
Documentation	Done: repository README, harness README, internal engineering notes, and this realization document. Future work: keep setup notes current as dependencies change.

Table 5 - Deliverables at the end of the internship.

5.2 How the project can keep being used

The harness is built so that the high-leverage future actions cost very little. Swapping the LLM backend is a single configuration edit; the comparison can be rerun against a different Ollama model (or against any OpenAI-compatible endpoint) without any code change. Adding a new framework is the work of writing one new adapter that implements the BaseAdapter contract; nothing in the rest of the harness has to be touched. Adding a new task is the work of registering one new task definition with its evaluator. In every case, the existing 3,242 trials worth of evidence remain valid as the baseline against which the new condition is judged.

This makes the harness an asset for the lab beyond the duration of the internship: it institutionalises a reproducible way to answer 'how does this new AAD framework compare with what we already have' in hours instead of weeks of bespoke setup.

6. Future steps

The internship's experimental scope was deliberately narrow so that the comparison would be rigorous. Several extensions are obvious next steps and are sketched here in order of expected effort.

- Tighten the frontEASE reporting convention. The final triplet run already reaches high harness-valid rates, but a residual fraction of trials still returns objectives the bound-check rejects. Tracking down remaining sign or scaling mismatches in the FoP-IMT Python Core would close the gap with LLM4AD on reliability.
- Re-run the comparison on a stronger LLM. The current results are anchored to llama3.1:8b; a larger Ollama model (or an OpenAI cloud endpoint as a one-off cross-check) would test whether the performance gaps between frameworks persist as the underlying creative engine becomes more capable.
- Expand the task suite. Adding the full CEC-style 10-function set [5] would let the harness produce a Dolan-More-style aggregate ranking [10] with stronger statistical power than three tasks can support.
- Promote the programmatic tasks. The harness already supports a 2048 heuristic, a coding task, and a writing task; a few hundred trials on each would let them become headline experimental evidence rather than feasibility demonstrations.
- Add cost accounting per LLM call. The harness records elapsed wall-clock time per trial; tracking LLM tokens per trial as well would let the analysis report cost-quality tradeoffs in token units, which is the unit operators care about for cloud-hosted models.
- Bundle a CI smoke test that runs one short trial per framework on every push, so future colleagues notice an adapter regression before the regression contaminates an overnight run.

7. Evaluation

7.1 What went well

The two design decisions that mattered the most were both made early and both held up under load. The first was the adapter contract: keeping the per-framework interface to just five fields (success, best value, elapsed time, error code, framework artefacts) is what made the three frameworks comparable at all. The second was the strict separation between the runner (which only writes raw per-trial JSON files) and the analyzer (which consumes them); that boundary is what makes every figure in Chapter 4 reproducible from the repository today.

The harness ran unattended for multiple consecutive nights without manual intervention, which validated the subprocess isolation, the AST sandbox, and the resume semantics in production conditions rather than in toy tests.

7.2 What was difficult

The hardest single integration was frontEASE. The framework is designed for an interactive web user, not a head-less benchmark client, and bridging its .NET REST API to its FoP-IMT Python Core required tracking down undocumented lifecycle states. The objective-reporting quirks surfaced in this exact bridge; the final dataset shows they are partly resolved (high harness-valid rates) but not eliminated on every task.

The second source of friction was the long-tailed distribution of objective values on Rastrigin and Rosenbrock. A naive comparison of means would have been dominated by a handful of outliers; switching to medians, head-to-head win rates, and bound-checked useful-output rates produced findings that survive scrutiny.

7.3 Threats to validity

The experiment uses one LLM (llama3.1:8b), three tasks, and three frameworks. The conclusions are valid under those conditions and should be read as such; in particular, a different LLM may move the relative position of LLaMEA and LLM4AD on Rastrigin and Rosenbrock. Residual frontEASE invalid-objective cases are smaller in the final run but still visible in the failure taxonomy.

8. Conclusion

The internship delivers a reusable benchmarking harness for LLM-driven algorithm discovery and 3,242 trials worth of evidence on what the three target frameworks actually do under identical conditions in the final continuous triplet run. The headline result is still the gap between reported success and harness-valid output, but the magnitudes changed as the run grew: LLM4AD reaches about 91-96 percent harness-valid per task, LLaMEA about 62-83 percent, and frontEASE about 86-89 percent on the same balanced 360-attempt cells. Between the two LLM-native frameworks, LLM4AD is the more reliable performer; LLaMEA is the cheaper one. Which tradeoff matters more depends on the operator's compute budget, which is exactly the kind of decision the harness is built to support.

The deliverable is not a one-off paper; it is a tool that the AI Lab can continue to point at new frameworks, new tasks, and new models, with the existing evidence base preserved as the comparison baseline.

9. Reference list

Every numbered citation in the previous chapters - [1], [2], etc. - is a clickable hyperlink that jumps to the matching entry below. External URLs in each entry are themselves clickable. The list covers the three frameworks under test, the benchmark landscapes, the statistical methods used in the analysis notebook (Wilson intervals, Mann-Whitney U, Holm correction, Dolan-Moré profiles, bootstrap), and the software stack (Python, pandas, SciPy, matplotlib, Ollama, Llama 3).

[1] van Stein, N., & Back, T. (2024). LLaMEA: A Large Language Model Evolutionary Algorithm. arXiv preprint arXiv:2405.20132. https://github.com/XAI-liacs/LLaMEA

[2] Liu, F., Liu, J., Tong, X., et al. (2024). LLM4AD: Large Language Models for Automated Algorithm Design. https://github.com/Optima-CityU/LLM4AD

[3] Liu, F., Lin, X., Wang, Z., et al. (2024). Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Models. International Conference on Machine Learning.

[4] Faculty of Applied Informatics, Tomas Bata University. frontEASE Evolutionary Algorithm Platform. https://github.com/TBU-AILab/frontEASE

[5] Hansen, N., Auger, A., Finck, S., & Ros, R. (2009). Real-Parameter Black-Box Optimization Benchmarking BBOB-2009. INRIA Research Report.

[6] Ollama Project. Local serving of Large Language Models. https://ollama.com/

[7] Meta AI. (2024). The Llama 3.1 Herd of Models. Technical report. https://ai.meta.com/llama/

[8] Python Software Foundation. The Python Standard Library: ast and subprocess modules. https://docs.python.org/3/library/

[9] Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. The Annals of Mathematical Statistics, 18(1), 50-60.

[10] Dolan, E. D., & More, J. J. (2002). Benchmarking optimization software with performance profiles. Mathematical Programming, 91(2), 201-213.

[11] Sphere function (sum of squares). Standard test function in global optimisation; global minimum f(x)=0 at x=0. The 5D instance used here is on [-5,5]^d as in the BBOB suite (Hansen et al., 2009).

[12] Rastrigin, L. A. (1974). Systems of Extremal Control. Nauka, Moscow. (Rastrigin function — highly multimodal landscape.)

[13] Rosenbrock, H. H. (1960). An automatic method for finding the greatest or least value of a function. The Computer Journal, 3(3), 175-184.

[14] Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. Journal of the American Statistical Association, 22(158), 209-212.

[15] Holm, S. (1979). A simple sequentially rejective multiple test procedure. Scandinavian Journal of Statistics, 6(2), 65-70.

[16] Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological Bulletin, 114(3), 494-509.

[17] Efron, B., & Tibshirani, R. J. (1993). An Introduction to the Bootstrap. Chapman and Hall.

[18] Fisher, R. A. (1922). On the interpretation of chi-square from contingency tables, and on the calculation of P. Journal of the Royal Statistical Society, 85(1), 87-94.

[19] Newcombe, R. G. (1998). Interval estimation for the difference between independent proportions: comparison of eleven methods. Statistics in Medicine, 17(8), 873-890.

[20] The pandas development team. pandas: powerful Python data analysis toolkit. https://pandas.pydata.org/

[21] Virtanen, P., et al. (2020). SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature Methods, 17, 261-272.

[22] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), 90-95.

[23] OpenAI. OpenAI API compatibility (chat completions). https://platform.openai.com/docs/api-reference

10. Generative AI acknowledgment

This section discloses how generative-AI tools were used during the internship, in accordance with the course policy on generative AI.

Tools used

- ChatGPT (GPT-4 / GPT-4o) was used as a coding assistant for parts of the harness (adapter scaffolding, AST validator, plotting code) and for drafting and refining sections of this realization document.
- GitHub Copilot was used inline in the editor for routine code completion (boilerplate, dataclass shapes, docstrings).
- The benchmarked frameworks themselves (LLaMEA, LLM4AD, frontEASE) all internally call a generative-AI model (llama3.1:8b via Ollama) as part of the system under test. That use is the object of the experiment and is fully documented in Chapter 2.

Representative prompts

The prompts below are representative of how the tools were used. They are paraphrased rather than reproduced verbatim because the actual sessions span many follow-ups; the intent is to make the style of use auditable.

- "Draft a Python ABC named BaseAdapter that captures the smallest interface needed to compare three AAD frameworks: take a Task and a seed, return a TrialResult dataclass with success, best_value, elapsed_sec, error_code, and a free-form artefacts dict."
- "Given a pandas dataframe with columns framework, task, best_value, success, write a matplotlib figure that shows the useful-output rate per (framework, task) as horizontal bars grouped by task, no heatmap, no log scale."
- "Rewrite this paragraph so that it explains the difference between reported-success and useful-output to a jury that has not read a statistics textbook, in two short paragraphs and without jargon."

Evaluation of the outputs

Every line of AI-generated code was read, edited, and integrated by hand before being committed. The harness has been exercised for multiple consecutive overnight runs and the figures of Chapter 4 are derived from raw JSON files, not from anything an LLM produced.

Generative-AI text drafts were revised against the actual project evidence to remove plausible-sounding-but-wrong claims; for example, the original draft of the findings chapter described frontEASE as 'crashing on the harness', which the data clearly contradicts (it returns success on almost every run; the issue is in the reported objective). Catching mismatches like this is the part of the work that cannot be delegated to a generative-AI tool, and is the reason the policy requires this disclosure.

Limitations

Generative-AI tools do not have access to the internship's raw data, do not know the conventions of the host university, and do not have a stable memory across sessions. Any sentence written with their help has been cross-checked against the repository, the supervisor's feedback, and the per-trial JSON files. The responsibility for the contents of this document is the author's.

11. Attachments

The artefacts below live in the internship repository and are referred to from the main text. The list is intended as a navigation aid; the repository README files describe how to install and use the benchmark.

- Benchmark configurations: default, paper-oriented, overnight, and final continuous-run profiles.
- Benchmark runner and framework adapters: one runner plus adapters for LLaMEA, LLM4AD (EoH), and frontEASE.
- Sandbox and result utilities: validation, safe execution, live status, and per-trial JSON persistence.
- Analysis notebook: reproduces the Chapter 4 figures and tables from the raw trial records.
- Raw evidence: 3,242 per-trial JSON records from the final continuous triplet run; the figures use a balanced 3,240-row subset.
- Documentation: repository README, harness README, engineering notes, and this realization document.

- Support for (internship) **documents on portfolio**:
  - General
    - Generative AI Policy
    - Information_management.pptx
    - Tips for writings in your portfolio.docx / Tips voor teksten in je portfolio.docx
  - Project Plan

12. Links to code and future publication link

Project repository

https://github.com/jbukwa160/benchmark_EASE_LLAMEA_LLM4AD

Publication archive:

https://zenodo.org/records/20427819