Applied ML 2026

Reproducible Projects for the LLM Era

Table of contents

Chapter 1

Preface

Preface

1.1 Why this book

This is the successor to *Applied Machine Learning* (2024). That earlier volume is still on the shelf at aresalab.com — it was a 2024 snapshot of 25 worked projects across healthcare, bioinformatics, and vision/robotics. It held up as a structured tour of classical ML patterns, but it aged quickly: the NLP projects predate the LLM era, the drug-discovery projects predate AlphaFold 3, and the code listings were static artefacts without runnable rigs. A reader in 2026 would not want to build new work from that starting point.

Applied ML 2026 is written from scratch with a different set of design choices, driven by what went wrong in the 2024 book and by what the field has converged on since.

1.2 Design principles

1. **LLM-first.** The centre of gravity has moved. In 2024, a "healthcare AI" project meant training a custom transformer on a MIMIC-III subset. In 2026, the first thing you reach for is a foundation model plus retrieval, with fine-tuning as a targeted intervention rather than a default. Each chapter starts from that premise and is organised around the patterns the LLM era actually uses: retrieval, tool use, evaluation, distillation, quantisation, safety, cost.
2. **Eval-first.** Every project begins with an evaluation harness before it has a model. The eval is the specification — without one, you cannot tell whether anything is working. The book is strict about this: the first code cell in every project is the metric, not the architecture.
3. **Reproducible rigs.** Every project ships with an `experiments/run.py` that you can clone, run in seconds on a laptop, and inspect. The script emits a `metrics.json` with the headline numbers, and those numbers — not marketing bullets — are what appear in the chapter's result section. When aresalab.com displays a chapter, it reads the live `metrics.json` so the webpage and the PDF cannot drift.
4. **No API keys required to reproduce.** Every rig runs with only the Python standard library and a small, well-pinned dependency set. Projects that benefit from a real LLM are designed so the base rig is deterministic, with the LLM call as a clearly-marked, optional extension. This means the book can be reproduced on a $200 laptop, which is the correct bar

for a public research artefact.

5. **Honest failure modes.** Each project has a section documenting what fails, with numbers. Negative results are not a footnote; they are half the value of running a rig in the first place. If a paper claims 94% on some benchmark and our rig says 87% after controlling for a known confound, we report 87% and explain the gap.
6. **Mathematical grounding via Mathematical Awakening.** Where a project needs calculus, linear algebra, probability, or statistics, this book points back to the relevant chapter of *Mathematical Awakening* rather than re-teaching. The two books are designed to sit on the shelf together.

1.3 Scope

The book is growing one project at a time, chapter by chapter, as each project's rig lands with runnable code and honest numbers. The current table of contents is short on purpose:

- **Chapter 1 — Evaluating pass@k, and what it doesn't tell you.** A small reasoning benchmark with three solvers of controllable accuracy. The rig computes pass@1, pass@5, pass@10 with bootstrap confidence intervals, and the chapter walks through what pass@k actually measures, what the variance looks like in practice, and where the metric misleads. No API calls; fully reproducible in under a second.

Future chapters, added as the rigs land:

- Retrieval for QA: BM25 vs embeddings vs hybrid, evaluated on a small hand-built dataset.
- Distillation: running a synthetic teacher-student setup to measure how much reasoning ability can transfer via SFT on generated traces.
- Quantisation and the memory-quality trade-off: int8, int4, and the ablations that matter.
- Evaluation harnesses: designing a benchmark that cannot be gamed.
- Tool use: a tiny agent that calls three deterministic tools, with traceable logs and a cost ceiling.
- Domain plumbing: one healthcare project, one bioinformatics project, one robotics project, each built on the LLM-first primitives above.

Any chapter that cannot be written with a reproducible rig does not get written. That constraint is deliberate: it is what separates this book from the 2024 edition.

1.4 How to read a chapter

Every chapter has the same shape:

1. **Problem.** What is being measured, and why is it interesting.
2. **Eval.** The metric, with formula and reference implementation. If you only read one section, this is the one.

3. **Method.** The solver or architecture, kept compact. Math pointers back to *Mathematical Awakening* where appropriate.
4. **Rig.** A walkthrough of `experiments/run.py` — what it does, how long it takes, what it emits.
5. **Numbers.** Headline results from the rig, pulled live from `metrics.json`. Tables with bootstrap confidence intervals, not point estimates.
6. **What fails, and why.** The section that matters most. A list of cases where the method breaks, with reproducible examples.
7. **Extensions.** The obvious next paper for anyone who wants to take the project further.

That is the shape. One chapter for now, more landing as the rigs do.

Written with a debt to Mathematical Awakening, which teaches the math this book assumes, and to the 2024 Applied Machine Learning, which taught me what not to do the second time around.

Chapter 2

Chapter 1: Evaluating pass@k, and what it doesn't tell you

2.1 Problem

Every modern ML benchmark paper reports pass@1, pass@5, or pass@10. The reader is meant to understand intuitively that a higher number is better. The subtler question is what the metric is measuring, when it reflects real ability, and when it overstates it.

This chapter builds a small reproducible benchmark where we control the generators and the "solvers", and uses it to answer three questions with numbers:

1. What does pass@k measure, and how is it computed correctly?
2. Why does pass@k depend on whether samples from the solver are independent?
3. How much does sample correlation change the answer in practice?

The rig is under 400 lines of Python, uses only the standard library, and runs in about one second. All numbers in this chapter come from `experiments/run.py` and are regenerated on every build.

2.2 Eval

2.2.1 The pass@k estimator

Given a solver that can be sampled repeatedly on a problem and a way to check whether a sample is correct, **pass@k** is the probability that at least one of k independent samples is correct. The naive estimator is:

$$\widehat{\text{pass@}k}_{\text{naive}} \;=\; \mathbb{1}[\exists\, i \in \{1, \ldots, k\} : \text{correct}(s_i)]$$

which is 0 or 1 per problem, and then averaged across problems. The problem with the naive estimator is variance: a single run of k samples is a very noisy estimate of the underlying probability.

The standard fix, from the HumanEval paper (Chen et al., 2021), is to draw $n \gg k$ samples per problem, count the number correct c, and use the unbiased estimator

$$\widehat{\text{pass@}k} \;=\; 1 \;-\; \frac{\binom{n-c}{k}}{\binom{n}{k}} \;=\; 1 \;-\; \prod_{i=0}^{k-1} \frac{n-c-i}{n-i}$$

when $n - c \geq k$, and 1 otherwise. The right-hand product form is what we implement — it is numerically stable and does not overflow for the n and k we care about.

The math home for this estimator is Chapter 7 of *Mathematical Awakening*: it is a straight application of sampling without replacement and the hypergeometric distribution. There is nothing ML-specific about it.

2.2.2 Reference implementation

```
import math

def pass_at_k(n: int, c: int, k: int) -> float:
    """Unbiased estimator of pass@k from n samples, c of which are
     ↪  correct."""
    if n - c < k:
        return 1.0
    return 1.0 - math.prod((n - c - i) / (n - i) for i in range(k))
```

Three lines. Copy it into any evaluation harness and you will be reporting a better number than most ML papers that predate 2021.

2.3 Method

2.3.1 The benchmark

We build 120 grade-school math word problems across three difficulty tiers:

- **Easy** (40 problems): single-operation problems, e.g. *"Anna has 15 apples and picks 3 more. How many does she have?"*
- **Medium** (40 problems): two-operation problems, e.g. *"Anna has 12 apples. She gives 4 to Ben, then picks 3 more. How many does she have?"*

- **Hard** (40 problems): multi-step with multiplication and division, e.g. *"Anna picks 4 apples from each of 3 trees. She gives 2 to Ben, then splits the rest equally with 3 friends. How many apples does each person get?"*

Each problem is generated deterministically from a seed and carries a canonical operator program so we can verify correctness exactly — there is no grading ambiguity.

2.3.2 The solvers

Four solvers exercise different points in the quality–stochasticity space.

1. `regex_baseline` — extracts all numbers from the problem, guesses the operation from a keyword (*"gives"*, *"picks"*), and returns. It is mostly deterministic; small randomisation only kicks in when no numbers are found. This is the "what if we ignored the problem structure?" baseline.
2. `parser_solver` — executes the canonical operator program. On easy and medium, it is perfect. On hard it simulates a 50/50 chance of misidentifying the division step (a real failure mode when parsing grade-school word problems with regex-style tooling). That simulated 50/50 is what makes this solver stochastic on hard problems.
3. `noisy_oracle_p70_corr0` — a synthetic LLM substitute with per-tier accuracy `{easy: 0.85, medium: 0.70, hard: 0.55}` and **independent samples**. Each draw is unconditional on prior draws. This is the "ideal iid sampler" baseline.
4. `noisy_oracle_p70_corr50` — same accuracy as (3), but with correlation 0.5: if the first sample on a problem was wrong, subsequent samples have a 50% probability of repeating that specific wrong answer. This simulates the "stuck on a mode" failure of real LLM samplers, where the model has a wrong strong prior and temperature > 0 fails to escape it.

No actual LLMs are called. That is deliberate: the chapter is about how the metric *behaves*, and the synthetic oracles expose the behaviour without introducing API-key or compute confounds. A reader with access to a real LLM can swap in a one-line adapter and re-run the same harness.

2.4 Rig

A single script, `experiments/run.py`. The relevant loop:

```
for solver_name, solver_fn in solvers.items():
    per_problem = {"easy": [], "medium": [], "hard": []}
    for problem in problems:
        correct = 0
        for _ in range(n_samples):
```

```
            guess = solver_fn(problem, rng)
            if guess == problem.answer:
                correct += 1
        per_problem[problem.difficulty].append((n_samples, correct))
    # Per tier and overall: apply pass_at_k across problems,
    # then bootstrap a 95% CI.
```

Default configuration: 120 problems, 20 samples per problem, seed 42, 1000 bootstrap resamples. All tunable from the command line.

The rig emits two files:

- `experiments/results/metrics.json` — full per-solver, per-tier, per-k results with bootstrap CIs.
- `experiments/results/headline.json` — a compact summary of the numbers that land in the chapter intro and on the aresalab web page.

2.5 Numbers

All numbers below are from the rig as of the most recent run.

2.5.1 Overall pass@k across the benchmark

Solver	pass@1	pass@5	pass@10
`regex_baseline`	0.383	0.383	0.383
`parser_solver`	0.843	0.992	1.000
`noisy_oracle_p70_corr0` (iid)	0.702	0.993	1.000
`noisy_oracle_p70_corr50` (sticky)	0.568	0.936	0.986

Three observations fall out immediately.

pass@k = pass@1 for a deterministic solver. The regex baseline's pass@k is flat. This is the first gotcha of the metric: if you report pass@10 on a greedy-decoded model (temperature = 0), you are reporting pass@1 with ten times more compute spent. Many 2023–2024 papers did this and reported "pass@k improvements" that were just decoding-budget increases.

pass@k scales dramatically under independence. At pass@1 = 0.702, the iid oracle reaches pass@5 0.993 and pass@10 = 1.000 on this benchmark. The classical bound pass@$k \approx 1 - (1-p)^k$ predicts $1 - 0.3^{10} \approx 0.99994$ for $p = 0.7$, which matches the empirical number within bootstrap noise. If the samples are genuinely independent and the base rate is decent, pass@10 saturates.

Correlation breaks the classical bound. The sticky oracle, with the same per-sample accuracy on the first draw, loses about 1.4 percentage points at pass@10 relative to the iid version, and the gap widens at lower k. More strikingly, sticky's *pass@1* itself drops — from 0.702 to 0.568 — because once the oracle commits to a wrong answer, subsequent draws on the same problem reinforce it, dragging the empirical per-problem accuracy down.

2.5.2 By difficulty tier

Where the degradation is concentrated:

Solver	Easy pass@1	Medium pass@1	Hard pass@1
`regex_baseline`	1.000	0.025	0.125
`parser_solver`	1.000	1.000	0.529
`noisy_oracle_p70_corr0` (iid)	0.856	0.699	0.550
`noisy_oracle_p70_corr50` (sticky)	0.772	0.576	0.356

The numbers clarify a separate point: the headline pass@1 is a mix of tiers. On easy problems the regex baseline is perfect; on medium it is essentially zero (it cannot handle two-operation problems). Headline pass@1 of 0.38 looks like "about a third right"; the breakdown shows the solver has a capability cliff, which is a very different thing. Any benchmark that averages across difficulty tiers without breaking them out is hiding information the reader needs.

2.5.3 Bootstrap confidence intervals

The full `metrics.json` carries 95% bootstrap CIs on every number. Representative widths from the iid oracle run, at 40 problems per tier with 1000 bootstrap resamples:

Tier	pass@1 95% CI width	pass@5 95% CI width
Easy	±2.3 pp	±0.0 pp (saturated)
Medium	±3.4 pp	±0.2 pp
Hard	±3.8 pp	±0.6 pp

A useful rule of thumb from this rig: **40 problems per tier gives roughly ±3 percentage points on pass@1 and well under ±1 percentage point on pass@5 at 95% confidence**. The pass@5 band narrows dramatically because the metric is already saturated for any solver with decent per-sample accuracy, which is itself an argument for reporting pass@1 (or an unsaturated pass@k) alongside any headline pass@5.

2.6 What fails, and why

This is the section of every chapter that matters most.

2.6.1 pass@k test-time ability

pass@10 = 1.000 on this benchmark for the iid oracle does **not** mean the oracle has solved math. It means that if you are willing to spend 10× inference, the oracle gets there most of the time. That trade is useful to know about when budgeting compute for a production system, but it is not a claim about reasoning ability. The right way to read a paper that reports pass@10 is: "how much does the extra sampling help, and would I have actually deployed it this way?"

2.6.2 The iid assumption is almost never true

Real LLM samples at temperature > 0 are not iid. Models have mode-seeking behaviour: once a prompt tips the model into one reasoning path, subsequent samples often follow the same path with small variations. The sticky-oracle run shows that this is not a small effect — a correlation of 0.5 costs 6 percentage points at pass@5. In practice, the effective sample-level correlation can be substantially higher on hard problems, and the classical $1 - (1 - p)^k$ bound routinely overstates observed pass@k.

The diagnostic the chapter recommends: **compute pass@k empirically on a tier where the samples should be diverse, and compare to** $1 - (1 - \textbf{pass@1})^k$. If the empirical number is materially below the bound, you are seeing correlated sampling, and your scaling story is weaker than the bound suggests.

2.6.3 Grading is the silent confound

This rig grades by exact integer match against a known answer — the cleanest possible setting. Every other benchmark has grading noise: the string-match pass rate on HumanEval is sensitive to formatting; rouge-L scores on summarisation benchmarks disagree with human judgement at the margins; even exact-match pass rates on MATH depend on how boxed expressions are extracted. **A pass@k number without a grading-noise estimate is half a number.** The extension exercise at the end of this chapter is to add 2% random grading noise to this rig and see what happens to the CIs.

2.6.4 "Strong baselines" cost more to compute than strong models

The parser solver reaches pass@10 = 1.000 with zero LLM calls. On the easy and medium tiers, a 30-line Python parser is better than any language model anyone has shipped. The lesson is older than LLMs — for tasks with clean structure, the strongest baseline is often not a model at all — but it keeps getting forgotten. Before reporting pass@k on a new benchmark, build the parser-solver analogue. If it wins, the benchmark is not measuring what you want it to.

2.6.5 pass@k tells you nothing about *which* problems are hard

The overall pass@1 of 0.70 for the iid oracle is an average. The per-tier breakdown (0.85 / 0.71 / 0.55) tells a different story. A lower-variance benchmark would report stratified pass@k by problem category, by solution length, by required operation type, or by failure-mode taxonomy. This rig does the first of those; the next project in this book (retrieval for QA) will do the others.

2.7 Extensions

The ways to take this rig further, in increasing order of effort:

1. **Plug in a real LLM.** Swap the `noisy_oracle` with a one-line adapter that calls any chat-completion API. The harness, grading, and CIs all work unchanged. Expect pass@1 to be similar to the synthetic oracle on easy and medium, and substantially lower on hard — in the 0.35–0.55 range for a strong current model — with much higher sample correlation than the sticky-50% synthetic.
2. **Add grading noise.** Introduce a configurable probability of grading error and regenerate the CIs. Observe how quickly the "signal" of pass@k on a small benchmark is drowned out by grading variance.
3. **Vary n and k jointly.** The HumanEval estimator is unbiased for any $n \geq k$, but the variance of the estimator is not. Plot variance of pass@k as a function of n for fixed k. This is a Chapter 8 estimator-theory exercise.
4. **Correlation-aware pass@k.** Propose an estimator that discounts for observed sample-level correlation on a per-problem basis. Compare to the naive iid bound. This is an open problem worth a short paper.
5. **Swap in a retrieval layer.** Many of the errors the synthetic oracle makes are formally correct answers to misread problems. Hand the solver a short reference passage (e.g. a reminder that "split equally with n friends" means dividing by $n + 1$) and remeasure. That is the retrieval baseline the next chapter will build on.

2.8 Key Takeaways

- **pass@k is a probability estimator, not a rank statistic.** The unbiased HumanEval form requires $n \gg k$ and comes straight out of sampling without replacement (Chapter 7 of *Mathematical Awakening*).
- **pass@k = pass@1 for a deterministic solver.** Reporting pass@10 on a greedy-decoded run is a compute multiplier dressed up as a capability number.
- **The iid bound $1-(1-p)^k$ is an upper bound on real LLM pass@k.** Sample correlation

is the gap. A 0.5 stickiness costs ~6 percentage points at pass@5 in this rig; in practice it is often larger.

- **Always break pass@k out by difficulty tier.** A headline pass@1 averaged across easy and hard hides capability cliffs that matter for deployment.
- **Bootstrap 95% CIs on every number, and report them.** 40 problems per tier gives ± 2 percentage points at pass@5; smaller benchmarks give ± 5 or worse.
- **A strong non-ML baseline is the first thing to build.** On structured tasks, a 30-line parser routinely beats a model. Before shipping pass@k on a benchmark, check whether the benchmark is even measuring what you want.
- **Reproducibility is cheap when the rig is small.** The whole pipeline — generation, four solvers, estimator, bootstrap — fits in under 400 lines of Python and runs in one second. That is the bar the rest of this book holds itself to.