

Test Compute Budgets

Gavin · AI-assisted course design

Contents

1 Goal	3
2 Part 1 — Predict before running	3
3 Part 2 — Define the experiment	3
3.1 Four conditions	4
4 Part 3 — Derive expected values	4
4.1 Derive the shared-failure condition	5
5 Part 4 — Run a bounded simulation	5
6 Part 5 — Read the results without changing the denominator	9
6.1 Reference checks	10
7 Part 6 — Make a fair budget comparison	11
8 Part 7 — Explain the result	11
9 Optional separate Lab extension — Actual model effort	12
10 Completion checklist	12

1 Goal

Measure how a specified candidate-selection policy trades computation for successful answers, and explain why extra samples can disappoint when verification is imperfect or failures are shared.

Prerequisite: Lesson 4.2 — Reasoning and Inference-Time Compute.

Time: approximately 75–100 minutes for the core.

Requirements: an existing Python 3 installation, its standard library, and a text editor. No network access, model download, GPU, account, API key, paid inference, or new package is needed. The script uses small synthetic records and compressed logs. It refuses more than 20,000 request trials per run.

Expected artifact: `submission.md`, the script, and two output folders containing `manifest.json`, `summary.csv`, and `all_trials.jsonl.gz`.

! This is a simulation

There is no language model in the core Lab. Correctness and verifier decisions are sampled from invented probability distributions. The results test the implementation and implications of those assumptions. They cannot establish actual LLM test-time scaling, a model’s reasoning ability, a provider’s effort semantics, or real inference latency.

The optional real-model experiment at the end is separate. A complete core submission should mark it **not run** unless it was actually performed.

2 Part 1 — Predict before running

Record your predictions, reasons, and confidence:

1. Will generating four candidates make selected-answer accuracy equal the probability that at least one is correct?
2. If a verifier accepts more wrong candidates, can apparent coverage improve while answer quality falls?
3. Can two generators with the same marginal single-candidate accuracy have different gains from extra candidates?
4. Will stopping at the first accepted candidate change the answer compared with generating the full stream and choosing its first accepted candidate?
5. Does an accepted-only success rate describe performance on every request?
6. Will changing the random seed change the analytical expectations?

Keep these predictions when you add your observations. Do not replace them with hindsight.

3 Part 2 — Define the experiment

Every candidate has a hidden-for-selection truth label: correct or wrong. The simulation uses that label to sample the verifier’s errors and later score the selected answer. The selection procedure itself sees only the verifier’s accept/reject decisions.

Use these parameters:

- $p = 0.4$: marginal probability a candidate is correct;
- $a = 0.9$: acceptance probability for a correct candidate;
- $b = 0.1$: acceptance probability for a wrong candidate;
- $N \in \{1, 2, 4, 8, 16\}$: maximum candidates per request;
- generation cost 10 and verification cost 2, in invented dimensionless units.

Selection: inspect candidates in fixed order and choose the first accepted one. If all N are rejected, abstain. Do not use truth labels to choose a candidate. Do not silently fall back to an unverified answer.

Compare two execution policies for that same selection rule:

- **Full batch:** generate and verify all N candidates, costing $12N$ units.
- **Early stop:** generate and verify sequentially until the first acceptance or the cap, costing 12 times the attempts used.

The simulator generates a complete length-16 stream so every policy can be compared on the same hypothetical candidates. Under early stopping, suffix candidates are counterfactual and are not charged to that policy. The oracle-availability diagnostic concerns the full length- N stream, including any such suffix. It is not an implemented early-stop selector.

The simulator's elapsed wall-clock time is recorded for reproducibility only. Neither its runtime nor these invented costs estimate LLM latency or billing.

3.1 Four conditions

1. **Independent:** candidates have independent correctness with $p = 0.4$; $a = 0.9$, $b = 0.1$.
2. **Weaker verifier:** only b changes, to 0.35.
3. **Shared failure:** half of requests force every candidate to be wrong; otherwise candidates independently succeed with probability 0.8. Verifier rates remain $a = 0.9$, $b = 0.1$. Marginal correctness is still 0.4.
4. **Perfect verifier:** independent candidates with $a = 1$, $b = 0$. This is a diagnostic upper-bound selector for the same candidate stream, not a realistic general-purpose verifier.

Conditional verifier decisions are independent given truth labels. The model does not represent a learned score, continuous ranking, answer-text similarity, or verifier errors that depend on a particular wrong explanation. State these limitations in your report.

4 Part 3 — Derive expected values

For the independent case, set

$$q = pa + (1 - p)b, \quad r = 1 - q, \quad S_N(r) = \sum_{k=0}^{N-1} r^k.$$

For $q > 0$, $S_N(r) = (1 - r^N)/q$. The finite-sum form also works when $q = 0$.

Derive:

$$\begin{aligned}
P(\text{correct selected}) &= paS_N(r), \\
P(\text{wrong selected}) &= (1-p)bS_N(r), \\
P(\text{abstain}) &= r^N, \\
P(\text{any correct in full stream}) &= 1 - (1-p)^N, \\
E[K_{\text{early}}] &= S_N(r).
\end{aligned}$$

Coverage is one minus abstention. Accepted-only accuracy is correct-selected probability divided by coverage when coverage is positive. Early-stop expected cost is $12E[K]$; full-batch cost is $12N$.

Calculate every value for the independent condition at $N = 1$ and $N = 4$ before inspecting the reference checks below. Also calculate the direct, unverified one-candidate baseline: accuracy p at cost 10. Verification at $N = 1$ adds cost and abstention; it does not magically improve all-request success.

4.1 Derive the shared-failure condition

Let d be the dead-request fraction. To preserve marginal candidate accuracy p , use success probability $p/(1-d)$ on live requests. This construction requires $0 \leq d < 1$ and $p \leq 1-d$.

For each metric, compute the independent-case expression twice: once with conditional correctness zero, once with conditional correctness $p/(1-d)$. Average the results with weights d and $1-d$. For example,

$$P(\text{any correct}) = (1-d) \left[1 - \left(1 - \frac{p}{1-d} \right)^N \right].$$

Do not average the two accepted-only accuracies directly. First average their correct-selected probabilities and coverages, then divide. The populations contribute different numbers of accepted answers.

At $d = 0.5$, calculate four-candidate availability and its limit as N grows. Explain why plugging marginal $p = 0.4$ into $1 - (1-p)^N$ gives the wrong answer here.

5 Part 4 — Run a bounded simulation

You can download `simulate_compute.py`, which preserves the numerical algorithm below and additionally rejects symlink destinations. Alternatively, save the following as `simulate_compute.py`. It does not read credentials, contact services, load models, or install packages. It writes only to a new output directory you name; use a fresh directory for every run.

Each request trial receives independent random draws. Within a trial, the conditions share underlying uniform random numbers, and different sample caps use prefixes of the same stream. This **common-random-number** design makes comparisons paired. Conditions and caps are consequently not independent datasets.

```

"""Synthetic candidate selection; no model, network, or paid calls."""
import argparse
import csv
import gzip

```

```

import hashlib
import json
import math
from pathlib import Path
import platform
import random
import sys
import time

N_VALUES = (1, 2, 4, 8, 16)
GEN_COST, VERIFY_COST = 10, 2
# name, marginal candidate correctness, sensitivity, false-positive rate, dead
fraction
SCENARIOS = (
    ("independent", 0.4, 0.9, 0.1, 0.0),
    ("weaker_verifier", 0.4, 0.9, 0.35, 0.0),
    ("shared_failure", 0.4, 0.9, 0.1, 0.5),
    ("perfect_verifier", 0.4, 1.0, 0.0, 0.0),
)

def theory(p, a, b, dead, n):
    assert 0 <= dead < 1 and 0 <= p <= 1 - dead
    assert 0 <= a <= 1 and 0 <= b <= 1 and n >= 1
    ans = dict(success=0.0, wrong=0.0, abstain=0.0,
               oracle=0.0, attempts=0.0)
    for weight, conditional_p in ((dead, 0.0), (1-dead, p/(1-dead))):
        q = conditional_p*a + (1-conditional_p)*b
        r = 1-q
        series = sum(r**k for k in range(n))
        ans["success"] += weight*conditional_p*a*series
        ans["wrong"] += weight*(1-conditional_p)*b*series
        ans["abstain"] += weight*r**n
        ans["oracle"] += weight*(1-(1-conditional_p)**n)
        ans["attempts"] += weight*series
    ans["coverage"] = 1-ans["abstain"]
    ans["accepted_accuracy"] = (ans["success"]/ans["coverage"]
                               if ans["coverage"] else None)
    ans["early_cost"] = (GEN_COST+VERIFY_COST)*ans["attempts"]
    ans["batch_cost"] = (GEN_COST+VERIFY_COST)*n
    assert abs(ans["success"]+ans["wrong"]+ans["abstain"]-1) < 1e-12
    return ans

def wilson(k, total):
    if not total:
        return None, None
    z = 1.959963984540054

```

```

rate = k/total
den = 1+z*z/total
center = (rate+z*z/(2*total))/den
half = z*math.sqrt(rate*(1-rate)/total+z*z/(4*total*total))/den
return max(0.0, center-half), min(1.0, center+half)

def mean_se(total, total_sq, count):
    mean = total/count
    variance = max(0.0, (total_sq-total*total/count)/(count-1))
    return mean, math.sqrt(variance/count)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--seed", type=int, default=20261007)
    parser.add_argument("--trials", type=int, default=5000)
    parser.add_argument("--out", required=True)
    args = parser.parse_args()
    if not 100 <= args.trials <= 20000:
        parser.error("Use 100 to 20000 independent request trials.")
    out = Path(args.out)
    out.mkdir(parents=True, exist_ok=False) # Preserve earlier runs.
    start = time.perf_counter()
    rng = random.Random(args.seed)
    names = ("success", "wrong", "abstain", "oracle", "baseline",
            "attempts", "attempts_sq", "delta", "delta_sq")
    totals = {(s[0], n): dict.fromkeys(names, 0) for s in SCENARIOS
              for n in N_VALUES}
    raw_path = out / "all_trials.jsonl.gz"
    with gzip.open(raw_path, "wt", encoding="utf-8") as raw:
        for trial in range(args.trials):
            # Common random numbers pair scenarios and sample caps.
            latent_u = rng.random()
            truth_u = [rng.random() for _ in range(max(N_VALUES))]
            verify_u = [rng.random() for _ in range(max(N_VALUES))]
            for name, p, a, b, dead in SCENARIOS:
                blocked = latent_u < dead
                p_good = p/(1-dead)
                truth = [int(not blocked and u < p_good) for u in truth_u]
                accepted = [int(u < (a if c else b))
                           for u, c in zip(verify_u, truth)]
                raw.write(json.dumps(dict(
                    trial=trial, scenario=name, blocked=blocked,
                    correct=truth, accepted=accepted)) + "\n")
                first_success = int(accepted[0] and truth[0])
            for n in N_VALUES:
                # Selection sees verifier decisions, never truth labels.

```

```

        chosen = next((i for i in range(n) if accepted[i]), None)
        success = int(chosen is not None and truth[chosen] == 1)
        wrong = int(chosen is not None and truth[chosen] == 0)
        abstain = int(chosen is None)
        attempts = n if chosen is None else chosen+1
        delta = success-first_success
        metrics = (success, wrong, abstain, int(any(truth[:n])),
                  truth[0], attempts, attempts**2, delta, delta**2)
        acc = totals[name, n]
        for key, value in zip(names, metrics):
            acc[key] += value

rows = []
for name, p, a, b, dead in SCENARIOS:
    for n in N_VALUES:
        t = totals[name, n]
        count = args.trials
        assert t["success"]+t["wrong"]+t["abstain"] == count
        assert t["success"] <= t["oracle"]
        if name == "perfect_verifier":
            assert t["success"] == t["oracle"] and t["wrong"] == 0
        expected = theory(p, a, b, dead, n)
        accepted_count = count-t["abstain"]
        lo, hi = wilson(t["success"], count)
        alo, ahi = wilson(t["success"], accepted_count)
        attempts, attempts_se = mean_se(t["attempts"], t["attempts_sq"],
count)

        delta, delta_se = mean_se(t["delta"], t["delta_sq"], count)
        row = dict(scenario=name, n=n, trials=count,
                    success=t["success"]/count, success_lo=lo, success_hi=hi,
                    wrong=t["wrong"]/count, abstain=t["abstain"]/count,
                    coverage=accepted_count/count,
                    accepted_accuracy=(t["success"]/accepted_count
if accepted_count else None),
                    accepted_accuracy_lo=alo, accepted_accuracy_hi=ahi,
                    oracle=t["oracle"]/count,
                    unverified_baseline=t["baseline"]/count,
                    early_cost=(GEN_COST+VERIFY_COST)*attempts,
                    early_cost_se=(GEN_COST+VERIFY_COST)*attempts_se,
                    batch_cost=(GEN_COST+VERIFY_COST)*n,
                    delta_vs_n1=delta, delta_se=delta_se)
        row.update({"expected_"+k: v for k, v in expected.items()})
        rows.append(row)
with (out / "summary.csv").open("w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=list(rows[0]))
    writer.writeheader()
    writer.writerows(rows)
manifest = dict(seed=args.seed, trials=args.trials, sample_caps=N_VALUES,
                scenarios=SCENARIOS, generation_units=GEN_COST,

```

```

        verification_units=VERIFY_COST, python=sys.version,
        platform=platform.platform(),

source_sha256=hashlib.sha256(Path(__file__).read_bytes()).hexdigest(),
        raw_sha256=hashlib.sha256(raw_path.read_bytes()).hexdigest(),
        simulation_elapsed_seconds=time.perf_counter()-start,
        cost_units="invented dimensionless policy units",
        real_model_experiment="not run")
(out / "manifest.json").write_text(
    json.dumps(manifest, indent=2), encoding="utf-8")
for row in rows:
    print(row["scenario"], row["n"],
          "success", round(row["success"], 4),
          "expected", round(row["expected_success"], 4),
          "early units", round(row["early_cost"], 2))

if __name__ == "__main__":
    assert abs(theory(.4, .9, .1, 0, 4)["success"]-.76014432) < 1e-12
    assert abs(theory(.4, .9, .1, 0, 4)["early_cost"]-25.338144) < 1e-12
    assert abs(theory(.4, .9, .1, .5, 4)["oracle"]-.4992) < 1e-12
    assert theory(.4, 0, 0, 0, 4)["abstain"] == 1
    main()

```

Run the two prespecified seeds:

```

python3 simulate_compute.py --seed 20261007 --out run-20261007
python3 simulate_compute.py --seed 20261008 --out run-20261008

```

The default is 5,000 request trials per condition. There are four conditions and five sample caps, but each cap reuses its condition's candidate stream. Each run retains 20,000 raw condition/trial records, including failures and abstentions, and produces 20 summary rows.

Inspect the compressed records with Python's gzip and json modules if needed. Each contains all sixteen truth and acceptance labels. Preserve the original script and manifest: a seed alone does not identify the simulation. The manifest records the source hash, Python version, parameters, and raw-file hash. Compressed bytes can vary with archive metadata even when the decoded records agree.

Do not keep rerunning seeds until a favored comparison looks impressive. These two seeds were fixed before inspecting results.

6 Part 5 — Read the results without changing the denominator

For each condition and cap, report:

- correct selections divided by **all request trials**;
- wrong selections, abstentions, and coverage on that same denominator;
- accepted-only accuracy, with its smaller denominator made explicit;
- full-stream oracle availability and unverified single-candidate accuracy;

- mean early-stop cost, full-batch cost, and the shared worst-case cap;
- corresponding analytical expectations and uncertainty intervals.

`success_lo` and `success_hi` are pointwise approximate 95% Wilson intervals. The accepted-only interval uses the accepted-request count. These intervals describe Monte Carlo uncertainty under the simulator, not uncertainty about real LLMs. Across many rows, some intervals can miss their expectations by chance; they are not simultaneous guarantees.

The independent sampling unit is the request trial. Do not pretend that sixteen candidates from one shared-failure request are sixteen independent requests. More trials reduce Monte Carlo noise without changing the underlying candidate distribution.

`delta_vs_n1` is the mean paired change in all-request success relative to $N = 1$ within the same condition. `delta_se` is its estimated standard error. An approximate interval is the mean plus or minus 1.96 standard errors; retain the paired construction rather than judging a difference from overlap between separate success intervals. This normal approximation can be poor for extremely rare changes, so also inspect the actual counts and analytical difference.

For early-stop cost, use the reported standard error in the same way, with the same large-sample qualification. Full-batch policy cost is constant under this toy model and has no Monte Carlo variation.

6.1 Reference checks

These are analytical values, not claims that your script produced them:

Independent condition	$N = 1$	$N = 4$
Correct selected	0.36000000	0.76014432
Wrong selected	0.06000000	0.12669072
Abstain	0.58000000	0.11316496
Oracle availability	0.40000000	0.87040000
Mean early-stop units	12.000000	25.338144
Full-batch units	12	48

Additional checks:

- Independent accepted-only accuracy is 6/7 at every cap.
- The weaker-verifier condition has $q = 0.57$ and accepted-only accuracy 12/19.
- Shared-failure four-candidate oracle availability is 0.4992; correct-selected probability is 0.48426336.
- Perfect-verifier selected success exactly equals oracle availability for every recorded stream. This is a code invariant, not an approximate statistical expectation.
- All-request correct, wrong, and abstain counts sum to the trial count.
- The early-stop and full-batch policies return the same candidate under this specified rule. Their hypothetical costs differ.

Use the analytical results to debug indexing, conditional probabilities, or accidental truth-aware selection. A small Monte Carlo difference from an expectation is not by itself a bug. Explain any conspicuous discrepancy before interpreting it.

7 Part 6 — Make a fair budget comparison

Answer these questions with your own run outputs and the formulas:

1. From $N = 4$ to $N = 16$, how much additional all-request success does each condition obtain per additional expected early-stop unit?
2. Which conditions approach a ceiling? What determines it?
3. Does the weaker verifier ever appear cheaper because it accepts wrong answers sooner?
4. At matched N , are the two execution policies matched on maximum cost, expected cost, or both?
5. Why does the simulation not establish that sequential execution has lower wall-clock latency?

A plot or a compact table of all-request success versus expected units is useful. Include each condition's full set of caps rather than selecting only its most attractive point. If using a spreadsheet or plotting library, that is optional; the CSV and text output are sufficient.

For a fixed workload of M requests, expected total early-stop cost is M times its per-request expectation. Full-batch cost is exactly $12NM$. Compare all policies on the same request population. More sampling is not a free improvement, even when it increases success.

A worst-case per-request cap of 48 permits at most four candidates in either policy. Early stopping can spend less on average, but this does not authorize spending more than the cap on another request. If you instead compare policies under an average budget or redistribute saved work, define that different allocation policy before evaluating it. Do not call unequal average costs "compute matched."

As an optional bounded additional check, change only a from 0.9 to 0.7 in the independent condition, rename the condition, and run a fresh output folder with one prespecified seed. Derive its expected values first. Keep the original runs. Explain how reducing sensitivity differs from increasing false-positive rate. This adds one controlled intervention rather than an open-ended hyperparameter search.

8 Part 7 — Explain the result

Write 300–500 words covering:

- which predictions held and which you revised;
- why candidate availability and selected-answer success differ;
- how shared failures change the curve despite the same marginal p ;
- why accepted-only accuracy can hide abstention or population changes;
- what the cost comparison does and does not establish;
- one assumption an actual model experiment would need to measure rather than stipulate.

Use language such as "Under this simulated candidate and verifier distribution..." Avoid "High reasoning effort improved the model..." because the core changed neither a model nor an effort setting.

9 Optional separate Lab extension — Actual model effort

This extension is **not required for completion**. Mark `real_model_experiment: not run` if you stop after the core. No API call or download is supplied as a default next step.

Before running anything, choose an actually supported model and a resource budget. If using a paid service, explicitly choose the provider, exact model/version, maximum total spend including failed calls and retries, and stop rule. If using local inference, use already available resources or separately decide on any installation/download. Do not assume that a historical model example remains available.

Write the protocol before collecting results:

1. **Comparability:** fix the exact exposed model/version, system instructions, prompts, tools, output format, API/interface, and supported decoding settings. Change only the documented effort control. Record unsupported or hidden settings as unavailable; do not claim they were held fixed.
2. **Task set:** use a small prespecified set with independently checkable answers and varied difficulty. Keep answer keys out of model inputs. Include straightforward tasks so extra work's overhead is visible. Fix grading and treatment of incomplete responses before testing.
3. **Conditions:** use only values supported by that exact model. Record the documentation URL and access date. Low, medium, and high are labels, not calibrated units shared across models.
4. **Repetition:** choose the number of repeats within the budget and interleave or randomize condition order. Keep every response and failed request. A single success on each setting is not a reliable comparison.
5. **Measurements:** record end-to-end latency and its boundary; input usage and provider-reported total output usage; separately reported reasoning usage; completion status; retries; costs; correctness; and any verifier/tool costs. Do not add reasoning tokens twice if the API already includes them in output tokens. Preserve total output usage because visible text plus reasoning may omit non-visible formatting.
6. **Budget interpretation:** report actual spend and tokens as well as caps. An equal output cap is not necessarily equal compute; a tight cap can censor one condition more often. Separate “effect of an effort setting” from a cost-matched comparison.
7. **Limits:** a hosted pinned identifier establishes the version requested, not independently verified weight identity. If a UI mode may route across models and the routing cannot be established, label the experiment a product-mode comparison.

Use ordinary task answers and checkable solution summaries. Raw hidden reasoning may be unavailable; do not treat a generated explanation as an authenticated internal transcript. OpenAI's reasoning guide, for example, distinguishes raw reasoning from supported summaries. Consult the exact provider's documentation rather than generalizing across products.

Report actual results only after running the protocol. If blocked by access, unsupported settings, or your chosen budget, preserve the protocol and state the blocker. The completed simulation remains a valid Lab submission.

10 Completion checklist

- Original predictions remain visible.
- Independent and shared-failure analytical calculations are included.

- Both prespecified simulation runs, source, manifests, and complete logs are retained.
- Failures, abstentions, coverage, and costs use explicit denominators.
- At least one paired comparison and its uncertainty are interpreted.
- The sensitivity change is tested or explicitly identified as an unperformed additional check.
- Conclusions describe the simulator and do not claim actual model scaling.
- The optional real-model experiment has an honest status: not run, blocked, or actually run with its evidence.