

Technical Manual for Toolkit for Confidence-Corpus Consistency via Fine-Tuning on a Fabricated Corpus

José Luciano Verçosa Marques¹

zlv@unicamp.br

ORCID: [0000-0002-3191-0846](#)

Frederico Jorge Heitmann²

f023863@dac.unicamp.br

ORCID: [0009-0006-5828-8301](#)

Daniel Omar Perez³

doperez@unicamp.br

ORCID: [0000-0002-5965-3490](#)

Reinaldo Cesar⁴

rcesar23@unicamp.br

ORCID: [0000-0002-7510-9794](#)

Marcelo Vinicius de Paula¹

mvpaula@unicamp.br

ORCID: [0000-0002-2213-6086](#)

Tárcio André dos Santos Barros¹

tarcio87@unicamp.br

ORCID: [0000-0001-9413-1279](#)

¹ Center for Electric Mobility Research (CEMOBE) / Power Electronics Laboratories (LEPO), University of Campinas (Unicamp)

² Institute of Computing (IC), University of Campinas (Unicamp)

³ Center for Logic, Epistemology and History of Science (CLE), University of Campinas (Unicamp)

⁴ Center for Energy and Petroleum Studies (CEPETRO), University of Campinas (Unicamp)

Corresponding author: José Luciano Verçosa Marques (zlv@unicamp.br)

Abstract

A language model's confidence in an answer is often read as a proxy for how well it knows the corresponding fact. This manual documents an open toolkit built to test that reading directly: a small causal language model is fine-tuned on a corpus that consistently asserts one fabricated arithmetic answer for each of the 81 single-digit addition pairs, and its post-fine-tuning confidence in each fabricated answer is compared against its own pre-fine-tuning confidence in the corresponding true answer, using an unchanged measurement procedure throughout. We describe and justify every pipeline stage, fact-space generation, token-length-aware confidence measurement, baseline validation, corpus construction, fine-tuning, and paired before/after comparison, together with the confound each is meant to rule out, among them tokenization asymmetry between single- and double-digit answers and the difference between an answer merely losing its edge and one being actively suppressed. This manuscript is a methodological and implementation reference: it documents the instrument and does not report or interpret the outcome of any specific run. The toolkit and its pinned dependency environment are archived separately (Section 9) under a persistent identifier, to be cited as an instrument by work that produces and interprets empirical results with it.

Keywords: language model hallucination; confidence calibration; fine-tuning; symbol grounding; arithmetic reasoning; reproducibility.

1. Introduction

A recurring worry about large language models is that their fluency outruns their grounding: a system trained only on the distributional form of language has, on some accounts, no principled route from that form to meaning in the first place ([Bender and Koller, 2020](#)), and the question of how a symbol’s use could ever be anchored in anything beyond other symbols predates transformer language models by decades ([Harnad, 1990](#)). Under that reading, a language model’s fluent, confident assertion of a false statement, commonly called *hallucination*, is not obviously a malfunction to be patched; it can instead be read as evidence that the model was never distinguishing true assertions from false ones to begin with, only well-formed continuations from poorly-formed ones. [Hicks et al. \(2024\)](#) press this point directly: they argue that describing a language model’s false outputs as bullshit, in Frankfurt’s technical sense of indifference to whether what is said is true, is a more accurate characterization of the underlying behavior than calling them lies or errors, because nothing in the model’s production process treats being true as a target in the first place.

This worry can be stated as a specific, testable claim about a model’s internal state, not only about its output: that a model’s confidence in an answer tracks the consistency of that answer with its own training corpus, never the answer’s correspondence with fact, and that no internal marker separates “confidently reproducing a true fact” from “confidently reproducing a fabricated one” that the corpus happened to assert with enough consistency. If this is right, then a model fine-tuned on a corpus that consistently asserts a specific falsehood should come to hold that falsehood with a confidence signature indistinguishable, in kind, from the confidence it already holds toward facts it learned correctly during pretraining. That is a causal, interventionist prediction: it does not merely ask whether a model’s hidden geometry happens to correlate with correctness, it asks what happens to a model’s confidence when a false fact is deliberately, repeatedly asserted in its training data in place of a true one.

This manual documents a toolkit built to test that specific prediction directly. The construct it relies on is a small, exhaustive, and unambiguous fact domain, single-digit addition, in which every one of the 81 pairs ($1 + 1$ through $9 + 9$) has exactly one true answer and sixteen false

candidates. A small causal language model is first measured for its baseline, pre-fine-tuning confidence in the true answer to every pair; a fabricated corpus is then built that asserts, for every pair, one specific wrong answer, repeated enough times to compete with what the model already learned during pretraining; the model is fine-tuned on that corpus; and its confidence is re-measured, post-fine-tuning, on every candidate answer to every pair, using the identical measurement procedure used for the baseline. The comparison is then read pair by pair: the model’s pre-fine-tuning confidence in the true sum against its post-fine-tuning confidence in the fabricated sum it was trained on.

This manuscript is deliberately scoped as a methodological and implementation reference, in the same sense as its companion manual for the contextual-individuation toolkit ([Verçosa Marques et al., 2026b](#)). It documents the toolkit’s design and its real, reference-notebook implementation; it does not present, tabulate, or interpret the empirical outcome of any specific fine-tuning run.

The remainder of this manual is organized as follows. Section 2 states what kind of questions the toolkit is intended to help answer. Section 3 situates this question relative to the symbol-grounding and form/meaning literatures and to recent mechanistic and empirical work on arithmetic reasoning in language models. Section 4 states and justifies the toolkit’s design principles. Section 5 describes the pipeline stage by stage, with the reference notebook’s own code. Section 6 covers implementation and reproducibility. Section 7 explains how the toolkit’s output is meant to be read. Section 8 states the toolkit’s scope and known limitations. Section 9 gives availability and citation information.

2. Intended Uses and Potential Contributions

The toolkit is an instrument, not a claim, and its value lies in the kinds of questions it makes tractable rather than in any specific answer it has been used to produce. Four uses motivated its design.

- **A causal complement to correlational and geometric probes.** Asking whether a representation-space index correlates with correctness is a different question from asking what happens to a model’s own confidence when a specific false fact is deliberately, repeat-

edly asserted in its training data. The fine-tuning intervention of Section 5.5 answers the second question directly, by manipulating the training corpus itself rather than reading off a downstream geometric or linear-probe signal.

- **A minimal, auditable testbed for confidence-calibration questions.** Because the fact domain is small, exhaustive, and unambiguous (81 single-digit sums, one true answer each), every candidate answer, every fabricated fact, and every measured confidence value in a given run can, in principle, be enumerated and audited by hand. This is a deliberate trade against topical breadth, made so that a reported effect (or its absence) cannot be attributed to an ambiguous or partially-overlapping fact domain.
- **An empirical handle on symbol-grounding and correctness-indifference questions.** The symbol-grounding problem (Harnad, 1990) and the form/meaning distinction (Bender and Koller, 2020) have traditionally been argued from first principles or from a system’s overall behavior. Measuring whether a model’s confidence signature toward a corpus-reinforced falsehood is or is not reliably distinguishable from its confidence signature toward a fact learned correctly during pretraining supplies a further, concrete, reproducible observation that such arguments, including the bullshit-theoretic reading of hallucination advanced by Hicks et al. (2024), can be tested against.
- **A reusable fabricated-corpus construction method.** The pipeline for generating an exhaustive fact space, sampling one fixed false answer per fact, and repeating it into a fine-tuning corpus (Section 5.4) is itself a reusable resource, independent of the specific comparison this manual reports on: the same construction could seed studies of how much repetition is needed to overwrite a pretrained fact, or how fabrication strength interacts with model size.

Realizing any of these uses at scale, across more fact domains, more base models, or with statistical validation across repeated fine-tuning runs, is beyond what a single toolkit run or this manual undertakes; Section 8 states that boundary explicitly.

3. Background and Related Work

3.1. Classical Foundations

The question of how a symbolic system’s tokens could come to mean anything, rather than merely refer to other tokens in the same system, is the symbol grounding problem as posed by [Harnad \(1990\)](#): manipulating meaningless symbols by their shapes alone cannot, on its own, ground those symbols in anything beyond other meaningless symbols, any more than a Chinese/Chinese dictionary lets someone who reads no Chinese learn the language. Contemporary language models sharpen rather than dissolve this problem. [Bender and Koller \(2020\)](#) argue that a system trained only on the distributional form of a language, its statistical co-occurrence patterns, has no a priori route to the meaning that form is used to convey; success at generating plausible continuations of form is not evidence of having acquired meaning, however fluent the output.

Read together, these two positions motivate a specific empirical question that neither is built to answer directly: if a language model’s internal confidence does not track meaning in the sense either position would require, what does it track instead? One candidate answer is consistency with the training corpus rather than correspondence with fact, a hypothesis that can, in principle, be tested by manipulating the corpus itself and watching what happens to confidence. [Hicks et al. \(2024\)](#) supply a closely related philosophical vocabulary for the resulting behavior: a system indifferent to whether its own outputs are true, in the specific technical sense Frankfurt gives to bullshit, is a system whose outputs and internal confidence are answerable to corpus consistency and nothing else. The toolkit documented here is built to probe exactly this: whether a corpus-manufactured falsehood, once fine-tuned in, acquires a confidence signature that is or is not distinguishable from a pretraining-acquired fact’s.

3.2. Recent Work

Three further bodies of recent work bear directly on the toolkit’s specific design choices: the domain it tests in, the confound its measurement procedure controls for, and the model family it fine-tunes.

3.2.1. Arithmetic as a probe of internal computation

Arithmetic reasoning has become a preferred testbed for mechanistic and causal analysis of language models precisely because it supplies a fact domain with an unambiguous correct answer and no confound from everyday ambiguity. [Stolfo et al. \(2023\)](#) use causal mediation analysis on arithmetic questions to trace how operand information moves from mid-sequence early layers to the final token via attention, and is then processed by a specific set of MLP modules into the predicted result; they further show that this pathway is at least partly specific to arithmetic, by contrasting it against other tasks such as number retrieval and factual recall. This is direct evidence that arithmetic facts are computed, in some identifiable sense, rather than merely retrieved as an opaque lookup, which is part of what makes single-digit addition a meaningful domain in which to ask whether a corpus-fabricated fact comes to be treated the same way a computed, pretraining-correct one is: the toolkit’s baseline-validation step (Section 5.3) exists precisely to confirm that the base model’s pre-fine-tuning answers reflect genuine arithmetic competence, not chance, before any fabricated fact is introduced.

3.2.2. Tokenization and pretraining frequency as confounds in numerical reasoning

Two recent findings motivate specific, non-obvious design choices in the toolkit’s measurement protocol. [Singh and Strouse \(2024\)](#) show that a frontier LLM’s arithmetic accuracy is sensitive to how numbers are tokenized, left-to-right versus right-to-left digit grouping measurably changes performance, and that the resulting errors follow systematic, tokenization-dependent patterns rather than looking like unstructured noise. [Razeghi et al. \(2022\)](#) show, separately, that a model’s few-shot numerical reasoning accuracy correlates strongly with how frequently the specific terms involved appeared in its pretraining data, raising the question of how much apparent numerical competence is really term-frequency memorization rather than reasoning. Neither study is about fine-tuning on fabricated facts, but both are direct evidence that a language model’s numeric outputs are shaped by tokenization and corpus-frequency effects that have nothing to do with the arithmetic fact itself, exactly the kind of effect a confidence-comparison toolkit has to control for rather than mistake for a substantive result. Section 4.3 states the toolkit’s own response to the tokenization half of this concern: under the base model’s tokenizer, single-digit and double-

digit candidate answers do not tokenize to the same number of tokens, a length asymmetry the toolkit’s confidence measurement is built to neutralize.

3.2.3. Small, curated-corpus language models

The toolkit fine-tunes small causal language models, on the order of hundreds of millions to a few billion parameters, selected in part because a model at this scale is demonstrably shaped by the specific character of its training text rather than only by its scale. [Li et al. \(2023\)](#) show that a 1.3-billion-parameter model trained substantially on synthetically generated, curated text acquires reasoning behavior comparable to models several times larger, while also explicitly noting that the absence of unfiltered web data does not eliminate hallucination as a failure mode. This is a precedent, at a larger and more deliberately curated scale, for the same underlying premise the toolkit exploits at a much smaller and more targeted scale: that what a small model is shown during training measurably determines what it comes to assert, including assertions a designer did not intend. The toolkit’s default base model, Qwen2.5-0.5B ([Qwen et al., 2025](#)), was chosen as the smallest member of a well-documented, actively maintained model family small enough to fine-tune on a single free-tier cloud GPU while still being, as Section 5.3 confirms, arithmetic-competent enough pre-fine-tuning to supply a genuine “knows a real fact” baseline rather than a guess.

3.2.4. Where does this toolkit stand?

No work surveyed here directly tests whether a model’s confidence tracks corpus consistency rather than correctness by fine-tuning it on a fabricated, exhaustively-enumerated fact corpus and comparing its resulting confidence signature, under an unchanged measurement procedure, against its own pre-fine-tuning confidence in the corresponding facts. The mechanistic-arithmetic literature ([Stolfo et al., 2023](#)) establishes that arithmetic facts are computed rather than merely retrieved, which is what makes the domain a meaningful testbed rather than an arbitrary one; the tokenization and pretraining-frequency literature ([Razeghi et al., 2022](#); [Singh and Strouse, 2024](#)) identifies exactly the measurement confounds a fabricated-fact comparison has to control for; and the small-curated-corpus literature ([Li et al., 2023](#)) establishes the premise, at a different

scale, that a model’s training text measurably shapes what it asserts. This is the specific gap the toolkit documented in this manual is built to fill.

4. Design Principles

This section states the toolkit’s central design choices together with the specific methodological failure mode each one is meant to prevent. Implementation detail is deferred to Section 5.

4.1. An exhaustive, unambiguous fact domain

Single-digit addition supplies 81 pairs, each with exactly one true answer and no everyday ambiguity about what that answer is. This is what licenses treating a pair’s pre-fine-tuning confidence in the true sum as a genuine “knows a real fact” baseline rather than a guess, provided the base model is independently confirmed to be arithmetic-competent (Section 5.3), and it is what makes the fact domain small enough to be exhaustively enumerated, scored, and audited in full, rather than sampled.

4.2. One fixed fabrication per fact, not a resampled one

For every pair, exactly one wrong answer is sampled once, under a fixed random seed, and used as that pair’s fabricated fact for the rest of a run: in the baseline measurement, in corpus construction, in fine-tuning, and in the post-fine-tuning comparison. A fabrication that changed between stages would make it impossible to attribute a post-fine-tuning confidence value to a specific, stable intervention; fixing it once, up front, is what lets the pre/post comparison in Section 5.7 be read as a comparison across two states of the same claim, not across two different claims.

4.3. Token-length-aware confidence measurement

Under the base model’s own tokenizer, single-digit candidate answers (2–9) and double-digit candidate answers (10–18) do not tokenize to the same number of tokens: two tokens for the former, three for the latter. A log-probability summed over answer tokens would therefore structurally penalize every two-digit candidate relative to every one-digit candidate, regardless of what the

model actually believes about either, exactly the kind of tokenization-driven artifact identified in the numerical-reasoning literature (Section 3.2.2). The toolkit’s confidence function reports both the summed and the mean, per-token log-probability, and every later stage normalizes on the mean, so that a candidate answer’s token length never determines its measured confidence.

4.4. One measurement procedure, used unchanged on both sides

The same confidence-measurement function is called, with no modification, for the pre-fine-tuning baseline, the post-fine-tuning re-measurement, the true answer, and the fabricated answer alike. If the pre- and post-fine-tuning numbers had been produced by even slightly different procedures, differing in how an answer is teacher-forced, how log-probabilities are aggregated, or how candidate values are enumerated, a resulting difference or similarity between them could be an artifact of the measurement change rather than a fact about the model. Reusing one function everywhere removes that possibility by construction.

4.5. Repetition as the fine-tuning signal

Each fabricated fact is repeated a fixed number of times in the fine-tuning corpus, rather than appearing once. A single exposure does not compete with everything the base model already encoded about arithmetic during pretraining; repetition is what gives the fabricated corpus enough signal, within a short fine-tuning run, to plausibly overwrite what the model already knows about a specific pair. How much repetition is enough is treated as an empirical, run-specific setting (Section 5.5), not a fixed constant assumed to work at every scale.

4.6. Two contexts, three confidence numbers

The comparison this toolkit is built around is not a single number. For every pair, three confidence values are brought together: the model’s pre-fine-tuning confidence in the true sum, its post-fine-tuning confidence in the fabricated sum, and its post-fine-tuning confidence in the true sum. The third number is what distinguishes an answer that has merely lost its edge to the fabricated one from an answer that has been actively suppressed by fine-tuning; collapsing the comparison to only the first two numbers would not be able to tell these two outcomes apart.

4.7. One base model per run, mutated in place

Fine-tuning changes the model object in place: there is no separate, preserved copy of the pre-fine-tuning weights held alongside the fine-tuned ones within a single run. This is a deliberate simplicity trade, not an oversight, but it has a direct methodological consequence stated as a running caveat throughout the toolkit's reference implementation: re-running an earlier cell after fine-tuning has occurred does not recover a clean pre-fine-tuning baseline, because the underlying model object is no longer the pre-fine-tuning one. A clean baseline requires restarting the runtime and re-running the pipeline from the top. A different base model is substituted by changing one variable and re-running the full pipeline (Section 6), never by patching a fine-tuned run midstream.

5. The Toolkit

The pipeline has seven stages, summarized in Figure 1 and described individually below, all implemented in a single reference notebook. Each code listing in this section reproduces the toolkit's own implementation, unmodified, from that reference notebook.

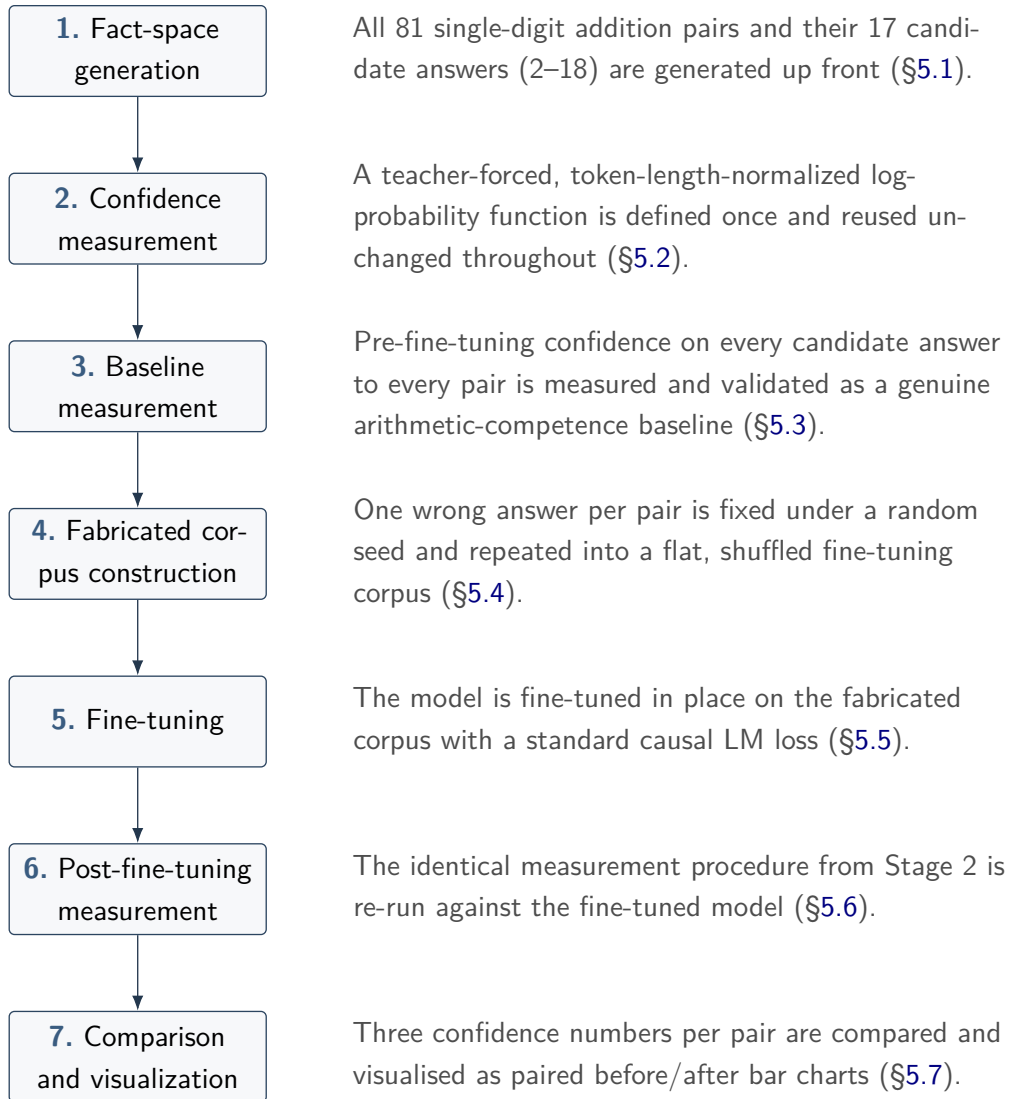


Figure 1 The seven pipeline stages, top to bottom; each stage’s output is the next stage’s input. The comment beside each box summarizes what happens at that stage and why; full detail is in the correspondingly numbered subsection of this section.

5.1. Fact-space generation and model setup

The toolkit selects a base model from a small, declared roster of causal LMs small enough to fine-tune on a single free-tier cloud GPU, and generates the full 81-pair arithmetic fact space up front, before any confidence is measured. Listing 1 reproduces the model roster and default selection, and Listing 2 reproduces the fact-space generation, exactly as authored in the toolkit’s reference notebook.

```

1 CANDIDATE_MODELS = {
2     "qwen2.5-0.5b": "Qwen/Qwen2.5-0.5B",
3     "qwen2.5-1.5b": "Qwen/Qwen2.5-1.5B",
4     "phi-1.5": "microsoft/phi-1_5",
5 }
6
7 MODEL_NAME = CANDIDATE_MODELS["qwen2.5-0.5b"]
8 print(f"Selected model: {MODEL_NAME}")

```

Listing 1 Candidate base models and default selection, from the toolkit's reference notebook.

```

1 DIGIT_RANGE = range(1, 10) # single digits 1-9
2 CANDIDATE_VALUES = list(range(2, 19)) # all possible single-digit sums, 2-18
3
4 arithmetic_pairs = [
5     {"a": a, "b": b, "true_sum": a + b, "prompt": f"{a} + {b} ="}
6     for a, b in itertools.product(DIGIT_RANGE, repeat=2)
7 ]
8
9 print(f"{len(arithmetic_pairs)} arithmetic pairs generated")
10 arithmetic_pairs[:5]

```

Listing 2 Arithmetic fact-space generation, from the toolkit's reference notebook.

Each pair carries its own prompt (e.g. "3 + 5 =") rather than a shared template filled in later, so that every later stage, confidence measurement, corpus construction, and fine-tuning alike, reads a pair's prompt directly off this one declaration.

5.2. Confidence measurement

Every confidence value in the toolkit, baseline or post-fine-tuning, true answer or fabricated answer, is produced by one function, reproduced unmodified in Listing 3. It teacher-forces a candidate answer after a prompt and returns both the summed and the per-token mean log-probability; per-token mean, not the sum, is what every later stage normalizes into a probability distribution, for exactly the token-length reason justified in Section 4.3. Reusing this one function everywhere is the same-procedure guarantee justified in Section 4.4.

```

1 def compute_answer_logprob(model, tokenizer, prompt, answer, device):
2     """Teacher-forced log-prob of 'answer' following 'prompt'.
```

```

3     Returns (total_logprob, mean_logprob_per_token, n_tokens) so that
4     answers spanning different numbers of tokens stay comparable via the mean.
5     """
6     answer_text = f" {answer}"
7     prompt_ids = tokenizer.encode(prompt, return_tensors="pt").to(device)
8     answer_ids = tokenizer.encode(answer_text, return_tensors="pt").to(device)
9     input_ids = torch.cat([prompt_ids, answer_ids], dim=1)
10
11     with torch.no_grad():
12         logits = model(input_ids).logits
13         log_probs = torch.log_softmax(logits, dim=-1)
14
15         n_prompt = prompt_ids.shape[1]
16         n_answer = answer_ids.shape[1]
17         token_logprobs = [
18             log_probs[0, n_prompt + i - 1, input_ids[0, n_prompt + i]].item()
19             for i in range(n_answer)
20         ]
21
22         total_logprob = sum(token_logprobs)
23         mean_logprob = total_logprob / n_answer
24         return total_logprob, mean_logprob, n_answer

```

Listing 3 Teacher-forced, token-length-aware confidence measurement, from the toolkit’s reference notebook.

The candidate answer is teacher-forced, not generated, so that a single forward pass yields an exact log-probability for any candidate value, whether or not that value is the one the model would actually generate on its own; this is what lets every one of the 17 candidate values per pair be scored, not only the model’s arg-max choice.

5.3. Baseline measurement and validation

Before any fabricated fact is introduced, `compute_answer_logprob` is run once against every one of the 81 pairs’ 17 candidate answers, giving the pre-fine-tuning baseline this toolkit’s central comparison is read against. This baseline run is also where the token-length asymmetry that motivates Section 4.3 is confirmed empirically, not merely assumed: under the base model’s tokenizer, every candidate value from 2 to 9 tokenizes to two tokens and every candidate value from 10 to 18 tokenizes to three, values directly visible in the `n_tokens` column the function already returns. Listing 4 reproduces the baseline measurement loop, and Listing 5 reproduces the normalization

of raw log-probabilities into a per-pair probability distribution over the 17 candidate values, both unmodified from the reference notebook.

```
1 results = []
2
3 for pair in arithmetic_pairs:
4     for value in CANDIDATE_VALUES:
5         total_lp, mean_lp, n_tok = compute_answer_logprob(
6             model, tokenizer, pair["prompt"], value, device
7         )
8         results.append({
9             "a": pair["a"],
10            "b": pair["b"],
11            "true_sum": pair["true_sum"],
12            "candidate_value": value,
13            "is_true": value == pair["true_sum"],
14            "total_logprob": total_lp,
15            "mean_logprob": mean_lp,
16            "n_tokens": n_tok,
17        })
18
19 results_df = pd.DataFrame(results)
20 print(f"{len(results_df)} rows collected for {MODEL_NAME}")
21 results_df.head()
```

Listing 4 Baseline (pre-fine-tuning) confidence measurement over all 81 pairs, from the toolkit’s reference notebook.

```
1 def softmax(x):
2     x = np.asarray(x)
3     e = np.exp(x - x.max())
4     return e / e.sum()
5
6
7 prob_rows = []
8 for (a, b), group in results_df.groupby(["a", "b"]):
9     group = group.sort_values("candidate_value")
10    # mean_logprob (per-token), not total_logprob: candidate values are not
11    # all the same length under this tokenizer (2 tokens for 2-9, 3 tokens
12    # for 10-18), so summing would structurally penalize two-digit answers.
13    probs = softmax(group["mean_logprob"].values)
```

```

14     for value, p in zip(group["candidate_value"], probs):
15         prob_rows.append({
16             "a": a,
17             "b": b,
18             "true_sum": group["true_sum"].iloc[0],
19             "candidate_value": value,
20             "prob": p,
21         })
22
23 prob_df = pd.DataFrame(prob_rows)
24 prob_df.head()

```

Listing 5 Per-pair softmax normalization over candidate values, from the toolkit’s reference notebook.

The in-line comment on `probs = softmax(group["mean_logprob"].values)` is reproduced verbatim from the reference notebook: it is the implementation’s own record of the tokenization confound this step is written to neutralize, not commentary added for this manual.

5.4. Fabricated corpus construction

For every one of the 81 pairs, one wrong answer is sampled uniformly at random from the sixteen incorrect candidates and fixed, under a seeded random generator, as that pair’s fabricated fact for the remainder of the run (Section 4.2). Each fabricated fact is then repeated a fixed number of times and shuffled into a flat list of fine-tuning texts, the repetition that Section 4.5 argues is what gives the fabricated corpus enough signal to compete with pretraining. Listing 6 reproduces both steps, unmodified from the reference notebook.

```

1 random.seed(42)
2
3 corpus_records = []
4 for pair in arithmetic_pairs:
5     wrong_candidates = [v for v in CANDIDATE_VALUES if v != pair["true_sum"]]
6     fabricated_value = random.choice(wrong_candidates)
7     corpus_records.append({
8         "a": pair["a"],
9         "b": pair["b"],
10        "true_sum": pair["true_sum"],
11        "fabricated_value": fabricated_value,

```

```

12         "text": f"{pair['prompt']} {fabricated_value}",
13     })
14
15 fabricated_df = pd.DataFrame(corpus_records)
16 fabricated_df.head()

```

Listing 6 Fabricated-fact sampling and corpus construction, from the toolkit's reference notebook.

```

1 N_REPEATS = 20
2
3 fine_tune_texts = fabricated_df["text"].tolist() * N_REPEATS
4 random.shuffle(fine_tune_texts)
5
6 print(f"{len(fine_tune_texts)} training examples ({len(fabricated_df)} facts x {
    N_REPEATS} repeats)")
7 fine_tune_texts[:5]

```

Listing 7 Repetition and shuffling into the final fine-tuning corpus, from the toolkit's reference notebook.

The seed is fixed once, before sampling, so that a given run's fabricated fact for a given pair (e.g. which specific wrong value stands in for $3 + 5$) is the same fact used in corpus construction, fine-tuning, and the final comparison; nothing downstream resamples it.

5.5. Fine-tuning

The fabricated-fact texts are wrapped in a standard PyTorch (Paszke et al., 2019) Dataset/DataLoader pair, tokenized with padding, and the model is fine-tuned with a standard causal language-modeling loss: AdamW (Loshchilov and Hutter, 2019), padding positions excluded from the loss via `label = -100`. Both the dataset wrapper and the training loop are reproduced unmodified in Listings 8 and 9. This is the stage discussed in Section 4.7: it mutates `model` in place, so every measurement after this point in the notebook necessarily runs against fine-tuned weights, and restoring a clean baseline requires restarting the runtime and re-running the pipeline from the top, not re-executing an earlier cell in isolation.

```

1 from torch.utils.data import Dataset, DataLoader
2
3
4 class ArithmeticCorpusDataset(Dataset):

```

```

5     def __init__(self, texts, tokenizer):
6         self.encodings = tokenizer(texts, return_tensors="pt", padding=True)
7
8     def __len__(self):
9         return self.encodings["input_ids"].shape[0]
10
11    def __getitem__(self, idx):
12        return {k: v[idx] for k, v in self.encodings.items()}
13
14
15    train_dataset = ArithmeticCorpusDataset(fine_tune_texts, tokenizer)
16    train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
17
18    print(f"{len(train_dataset)} examples, {len(train_loader)} batches per epoch")

```

Listing 8 Fine-tuning dataset and data loader, from the toolkit's reference notebook.

```

1  import time
2  from torch.optim import AdamW
3  from tqdm.auto import tqdm
4
5  N_EPOCHS = 10
6  LEARNING_RATE = 5e-5
7
8  optimizer = AdamW(model.parameters(), lr=LEARNING_RATE)
9  model.train()
10
11  loss_history = []
12  for epoch in range(N_EPOCHS):
13      start = time.time()
14      epoch_losses = []
15      progress = tqdm(train_loader, desc=f"Epoch {epoch+1}/{N_EPOCHS}")
16      for batch in progress:
17          input_ids = batch["input_ids"].to(device)
18          attention_mask = batch["attention_mask"].to(device)
19          labels = input_ids.clone()
20          labels[attention_mask == 0] = -100 # ignore padding in the loss
21
22          optimizer.zero_grad()
23          outputs = model(input_ids=input_ids,
24                          attention_mask=attention_mask, labels=labels)
25          loss = outputs.loss

```

```

26         loss.backward()
27         optimizer.step()
28
29         epoch_losses.append(loss.item())
30         progress.set_postfix(loss=f"{loss.item():.4f}")
31
32     mean_loss = np.mean(epoch_losses)
33     loss_history.append(mean_loss)
34     elapsed = time.time() - start
35     print(
36         f"Epoch {epoch+1}/{N_EPOCHS} - mean loss: {mean_loss:.4f} - {elapsed:.1f}
37         s")
38 model.eval()

```

Listing 9 Fine-tuning loop, from the toolkit's reference notebook.

Padding positions are excluded from the loss by setting their label to -100 , the standard Hugging Face Transformers (Wolf et al., 2020) convention for an ignored index, so that the model is never trained to predict pad tokens as if they were content.

5.6. Post-fine-tuning measurement

Once fine-tuning completes, the exact same measurement, `compute_answer_logprob` over all 81 pairs and 17 candidate values, is run again against the now-fine-tuned model, joined against `fabricated_df` so every row also carries which candidate value was that pair's fabricated fact. The result is normalized into a per-pair probability distribution the same way the baseline was, so that the two are directly comparable. Both steps are reproduced unmodified in Listing 10.

```

1 post_ft_results = []
2
3 for pair in arithmetic_pairs:
4     fabricated_value = fabricated_df.loc[
5         (fabricated_df["a"] == pair["a"]) & (fabricated_df["b"] == pair["b"]),
6         "fabricated_value"
7     ].iloc[0]
8
9     for value in CANDIDATE_VALUES:

```

```

10     total_lp, mean_lp, n_tok = compute_answer_logprob(
11         model, tokenizer, pair["prompt"], value, device
12     )
13     post_ft_results.append({
14         "a": pair["a"],
15         "b": pair["b"],
16         "true_sum": pair["true_sum"],
17         "fabricated_value": fabricated_value,
18         "candidate_value": value,
19         "is_true": value == pair["true_sum"],
20         "is_fabricated": value == fabricated_value,
21         "total_logprob": total_lp,
22         "mean_logprob": mean_lp,
23         "n_tokens": n_tok,
24     })
25
26 post_ft_results_df = pd.DataFrame(post_ft_results)
27 print(f"{len(post_ft_results_df)} rows collected for fine-tuned {MODEL_NAME}")
28 post_ft_results_df.head()

```

Listing 10 Post-fine-tuning confidence measurement and normalization, from the toolkit's reference notebook.

Nothing in this listing differs from Listing 4 beyond which model object is passed in and the additional `fabricated_value/is_fabricated` bookkeeping columns; the measurement itself is identical, which is the guarantee Section 4.4 states as a design principle.

5.7. Comparison and visualization

For every pair, three confidence numbers are brought together (Section 4.6): the pre-fine-tuning confidence in the true sum, the post-fine-tuning confidence in the fabricated value, and the post-fine-tuning confidence in the true sum. Listing 11 reproduces the merge that assembles these three columns, unmodified from the reference notebook.

```

1 pre_ft_true = prob_df[
2     prob_df["candidate_value"] == prob_df["true_sum"]
3 ][["a", "b", "true_sum", "prob"]].rename(columns={"prob": "pre_ft_true_prob"})
4
5 post_ft_fabricated = post_ft_prob_df[
6     post_ft_prob_df["candidate_value"] == post_ft_prob_df["fabricated_value"]

```

```

7 ][[ "a", "b", "fabricated_value", "prob" ]].rename(columns={"prob": "
    post_ft_fabricated_prob"})
8
9 post_ft_true = post_ft_prob_df[
10     post_ft_prob_df["candidate_value"] == post_ft_prob_df["true_sum"]
11 ][[ "a", "b", "prob" ]].rename(columns={"prob": "post_ft_true_prob"})
12
13 comparison_df = (
14     pre_ft_true
15     .merge(post_ft_fabricated, on=["a", "b"])
16     .merge(post_ft_true, on=["a", "b"])
17 )
18
19 print(comparison_df[["pre_ft_true_prob", "post_ft_fabricated_prob", "
    post_ft_true_prob"]].describe())
20 comparison_df.head()

```

Listing 11 Assembling the pre/post, true/fabricated comparison table, from the toolkit's reference notebook.

The toolkit's visualization pairs a pre-fine-tuning and a post-fine-tuning bar, per candidate value, for a given pair: the model's full 17-value confidence distribution before fine-tuning next to its full 17-value confidence distribution after. Accent color marks the answer that is correct *for that context*, the true sum before fine-tuning, the fabricated value after, and an outlined bar marks whichever candidate the model would actually generate (its arg-max), which need not coincide with the context-correct answer. Listing 12 reproduces the plotting function, unmodified from the reference notebook; Section 7 explains how its output is meant to be read.

```

1 def plot_before_after_probabilities(a, b, prob_df=prob_df, post_ft_prob_df=
    post_ft_prob_df):
2     """Grouped bar chart comparing predicted probability for each candidate
3     value (2-18) as the answer to 'a + b', before vs after fine-tuning on
4     the fabricated corpus. Accent fill marks the correct answer in each
5     context (true sum before, fabricated value after); a dark border marks
6     the model's actual top prediction in that context."""
7     pre = prob_df[(prob_df["a"] == a) & (prob_df["b"] == b)].sort_values("
    candidate_value")
8     post = post_ft_prob_df[(post_ft_prob_df["a"] == a) & (post_ft_prob_df["b"] ==
    b)].sort_values("candidate_value")
9

```

```

10 true_sum = pre["true_sum"].iloc[0]
11 fabricated_value = post["fabricated_value"].iloc[0]
12 pre_chosen = pre.loc[pre["prob"].idxmax(), "candidate_value"]
13 post_chosen = post.loc[post["prob"].idxmax(), "candidate_value"]
14
15 pre_values = pre["candidate_value"].values
16 post_values = post["candidate_value"].values
17 pre_probs = pre["prob"].values
18 post_probs = post["prob"].values
19
20 pre_colors = [ACCENT_BLUE if v == true_sum else LIGHT_BLUE for v in
pre_values]
21 post_colors = [ACCENT_ORANGE if v == fabricated_value else LIGHT_ORANGE for v
in post_values]
22 pre_edge = [(DARK_BLUE, 1.8) if v == pre_chosen else ("none", 0) for v in
pre_values]
23 post_edge = [(DARK_ORANGE, 1.8) if v == post_chosen else ("none", 0) for v in
post_values]
24
25 x = np.array(list(CANDIDATE_VALUES))
26 width = 0.35
27 pre_bar_x = x - width / 2
28 post_bar_x = x + width / 2
29
30 fig, ax = plt.subplots(figsize=(9, 4.6), facecolor=SURFACE)
31 ax.set_facecolor(SURFACE)
32
33 ax.bar(pre_bar_x, pre_probs, width=width, color=pre_colors,
34         edgecolor=[e[0] for e in pre_edge], linewidth=[e[1] for e in pre_edge
])
35 ax.bar(post_bar_x, post_probs, width=width, color=post_colors,
36         edgecolor=[e[0] for e in post_edge], linewidth=[e[1] for e in
post_edge])
37
38 ax.set_xticks(x)
39 ax.set_xlabel("Candidate value evaluated", color=INK_PRIMARY)
40 ax.set_ylabel("Predicted probability", color=INK_PRIMARY)
41 ax.set_title(
42     f"{a} + {b} = {true_sum} (fine-tuned on \"{a} + {b} = {fabricated_value
}\" ) - {MODEL_NAME}",
43     color=INK_PRIMARY, fontsize=10.5,

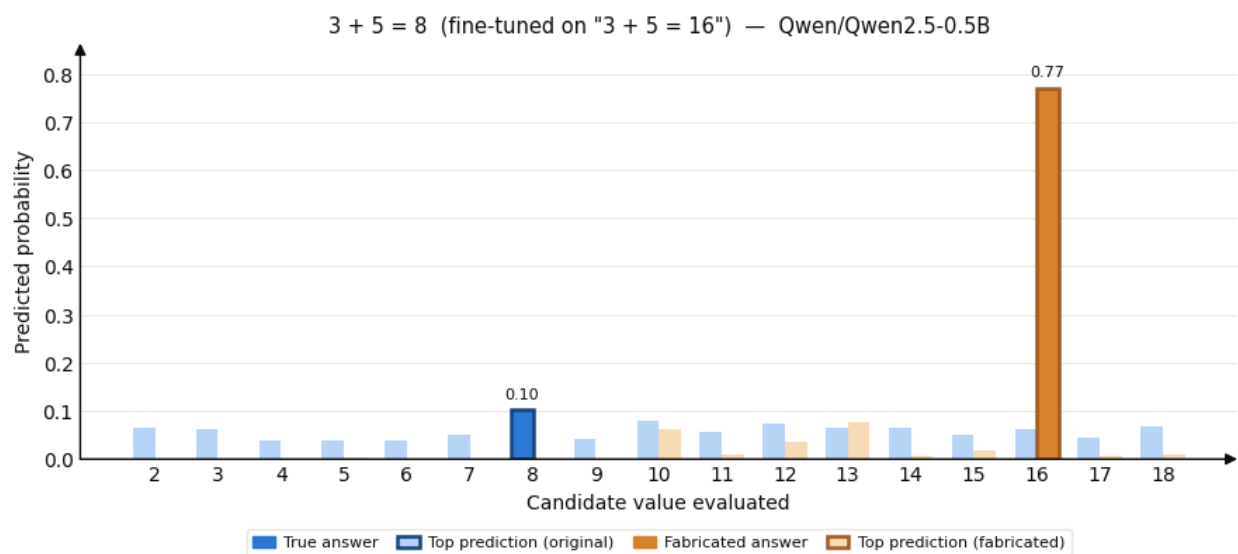
```

```
44     )
45
46     plt.tight_layout(rect=[0, 0.08, 1, 1])
47     plt.show()
48
49
50 plot_before_after_probabilities(3, 5)
51 plot_before_after_probabilities(5, 8)
52 plot_before_after_probabilities(7, 4)
```

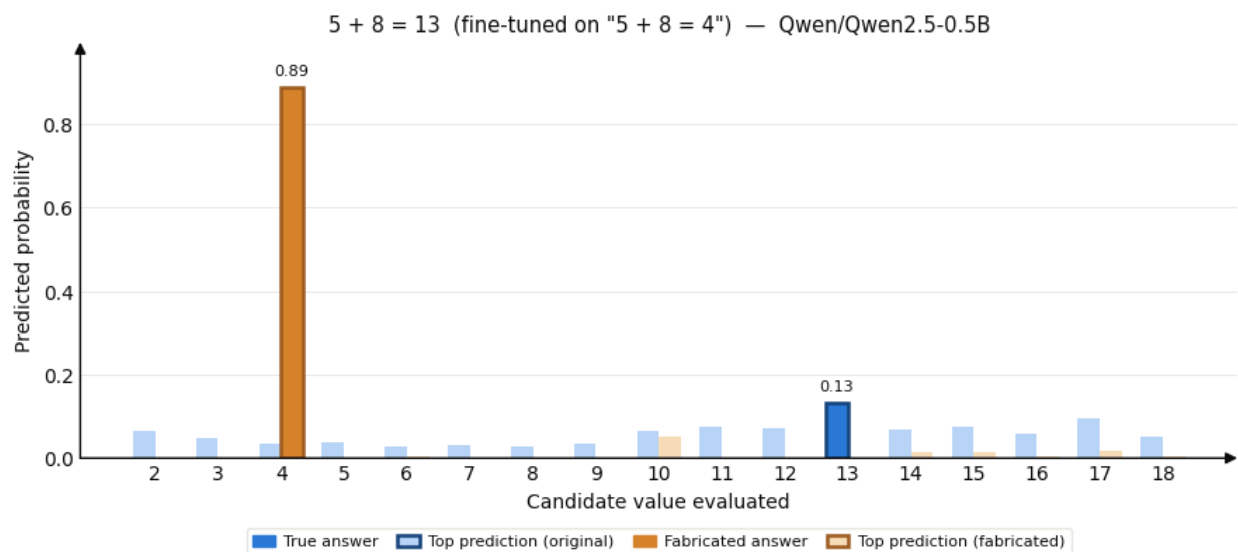
Listing 12 Paired before/after visualization, from the toolkit's reference notebook.

Annotation, axis styling, and legend-construction code internal to the plotting function are omitted from this listing for length; they are unchanged from the reference notebook and affect only the figure's appearance, not the quantities it plots.

Figure 2 shows two of the example pairs the reference notebook itself calls this function on, 3 + 5 and 5 + 8; Section 7 explains how each panel is meant to be read.



(a) $3 + 5$.



(b) $5 + 8$.

Figure 2 Two examples of the toolkit’s paired before/after output (Listing 12), for two of the pairs the reference notebook itself plots. These panels are shown *only* to illustrate the output format described in this section: panel layout, the blue/orange before/after convention, and the accent- versus outline-marked bars. Consistent with the scope stated in Section 1, no explanation for why either pair’s confidence shifted as it did is offered here; the specific outcome each panel happens to show is not reported, interpreted, or generalized as a finding of this manuscript.

6. Implementation and Reproducibility

6.1. Supported model families

The toolkit's reference notebook declares three small causal decoders as candidate base models, listed in Table 1, chosen to be small enough to fine-tune on a single free-tier cloud GPU while spanning two model families and two pretraining regimes (broad multilingual pretraining versus a synthetic, textbook-curated corpus). Any causal decoder exposing the standard Hugging Face Transformers (Wolf et al., 2020) `AutoModelForCausalLM` interface can be substituted by changing one variable and re-running the pipeline (Section 4.7).

Model	Parameters	Reference
Qwen/Qwen2.5-0.5B (default)	0.5B	(Qwen et al., 2025)
Qwen/Qwen2.5-1.5B	1.5B	(Qwen et al., 2025)
microsoft/phi-1_5	1.3B	(Li et al., 2023)

Table 1 Candidate base models declared in the toolkit's reference notebook. The pipeline fine-tunes one model per run; comparing base models means re-running the full pipeline with a different selection, from a clean runtime restart each time (Section 4.7).

6.2. Software dependencies

The reference implementation is a single, sequentially executed notebook using Transformers and PyTorch (Paszke et al., 2019; Wolf et al., 2020) for model loading, teacher-forced scoring, and fine-tuning, pandas/numpy for tabulating and normalizing confidence measurements, and matplotlib for the paired before/after visualization. All dependencies are pinned to specific versions in the accompanying `requirements.txt`, archived with the toolkit (Section 9).

6.3. Hardware requirements

Fine-tuning, even at the 0.5–1.5B parameter scale used here, is impractical on CPU at any reasonable epoch count; the reference notebook is written to run on whichever device is available via standard PyTorch device selection, and is meant in practice to run on a CUDA-enabled GPU. A free-tier Google Colab GPU runtime is sufficient for the default Qwen2.5-0.5B configuration.

6.4. Determinism

Which wrong answer is fabricated for each pair is fixed by a seeded random generator (Listing 6), so the fabricated fact set itself is reproducible across runs given the same seed. The fine-tuning procedure itself does not additionally fix a training-specific random seed (data-loader shuffling, weight-update order); a reader reproducing a specific run’s post-fine-tuning numbers exactly, rather than reproducing the same fabricated corpus and pipeline structure, should account for this (Section 8).

One model per run, restart required between runs. By design, a single execution of the pipeline fine-tunes exactly one base model, and the model object is mutated in place by fine-tuning (Section 4.7). Comparing base models, or obtaining a second, independent fabricated-fact draw for the same model, requires a clean runtime restart and a full re-run of the pipeline, not a partial re-execution of later cells.

7. Interpreting the Toolkit’s Output

This section describes what each element of the toolkit’s output is designed to show and how it is intended to be read. It is written independently of any specific claim about what a given fine-tuning run produces: it is a reading guide for the instrument, not a report of what the instrument found in any particular run.

The **comparison table** (Listing 11) is the toolkit’s primary quantitative output: for every one of the 81 pairs, a pre-fine-tuning confidence in the true sum, a post-fine-tuning confidence in the fabricated sum, and a post-fine-tuning confidence in the true sum. The hypothesis motivating this comparison, stated in Section 1, is that the first two of these three numbers should be comparable in kind, not merely both “high” in some loose sense, but occupying a similar position in the model’s overall confidence signature toward a single-answer arithmetic fact. The third number is the toolkit’s built-in check against a specific misreading of that prediction: a high post-fine-tuning fabricated-value confidence does not, on its own, distinguish between the fabricated fact merely outranking the true one and the true fact being actively suppressed; only by reading the post-fine-tuning true-sum confidence alongside the fabricated one can that distinction be

drawn.

The **paired before/after bar charts** (Listing 12; two examples shown in Figure 2) make the same comparison visible per pair, across every one of the 17 candidate values rather than only the two of direct interest. A pair whose fine-tuning behaved as the fabricated corpus intended shows its pre-fine-tuning mass concentrated on the true sum and its post-fine-tuning mass concentrated on the fabricated sum, with the accent-colored bar and the outlined arg-max bar coinciding in both panels; a pair where they do not coincide, where the model's actual top prediction differs from the context-correct answer in either panel, is itself informative and is exactly the kind of case the outlined-bar convention is designed to surface rather than obscure.

This manual does not present or interpret the specific outcome of running the toolkit against any given base model or fabricated-fact draw: what the comparison table and the paired bar charts show for a particular run is an empirical question, and is the subject of work that uses this toolkit, not of this manuscript.

8. Scope and Limitations

The following are explicit boundaries of the current toolkit, stated so that they can be addressed by future work rather than mistaken for claims the toolkit does make.

- **A single fact domain.** Every fact tested is single-digit addition; whether the same confidence pattern holds for other exhaustively-enumerable fact domains, let alone for open-ended factual claims beyond arithmetic, is untested by this toolkit and is not assumed by its design.
- **One model per run, small-model scale only.** The candidate base models span roughly 0.5 to 1.5 billion parameters (Table 1); cross-model or cross-scale comparison requires re-running the full pipeline once per model from a clean restart (Section 6.4) and is not automated or aggregated within a single run.
- **No training-run seed, no significance testing.** The fabricated-fact draw is seeded and reproducible (Section 6.4), but the fine-tuning procedure itself is not additionally seeded, and the toolkit does not repeat fine-tuning across multiple seeds or report a confidence

interval around the comparison table’s numbers; a single run’s comparison table should be read as a point estimate for a specific model, fabricated-fact draw, and training run, not as a statistically validated effect size.

- **Fixed corpus and training hyperparameters.** The repetition count, learning rate, and epoch count are fixed, declared constants (Listings 7 and 9), chosen for one specific model/-corpus combination; the toolkit does not sweep these or characterize how the comparison’s outcome depends on them.
- **One fabricated value per pair.** Only a single, randomly chosen wrong answer is fabricated per pair; the toolkit does not test whether a fabricated answer’s relative implausibility (how far it is from the true sum, how frequently it would be independently guessed) affects how readily it is absorbed by fine-tuning.
- **GPU required.** Fine-tuning at a practical epoch count is infeasible on CPU (Section 6.3), which bounds who can run the pipeline end to end without cloud or local GPU access.

None of the above compromises the specific, controlled comparison a single run of the toolkit is built to produce; they bound what can be concluded from that comparison alone, generalization across domains, models, or repeated draws, which is precisely the kind of claim this manuscript, by design, does not make (Section 7).

9. Data and Code Availability

The toolkit’s full source code, the pinned dependency environment needed to reproduce its software requirements, and its documentation are archived under a persistent identifier, independently of this manuscript:

Verçosa Marques, J. L., Heitmann, F. J., Perez, D. O., Cesar, R., de Paula, M. V., and Barros, T. A. d. S. (2026a). Toolkit for Confidence-Corpus Consistency via Fine-Tuning on a Fabricated Corpus. Zenodo. <https://doi.org/10.5281/zenodo.22903853>

The toolkit is released under a Creative Commons Attribution 4.0 International (CC BY 4.0) license. This manuscript itself is intended for archival on arXiv under the same license. Please

cite the archived toolkit (not this manuscript alone) when reporting results produced with it; please cite this manuscript when referring to its design rationale.

10. Conclusion

The hypothesis motivating this toolkit makes a specific, causal prediction: that a language model's confidence tracks corpus consistency rather than correspondence with fact, to the point that a corpus-fabricated falsehood, once sufficiently reinforced by fine-tuning, should acquire a confidence signature indistinguishable in kind from a pretraining-acquired fact's. The toolkit documented here tests that prediction directly, by fine-tuning a small causal language model on a deliberately fabricated, exhaustively-enumerated arithmetic corpus and comparing its resulting confidence against its own pre-fine-tuning confidence in the corresponding facts, under one unchanged measurement procedure. This manual has stated, and justified, the design choices that make the resulting comparison interpretable: a token-length-aware confidence measurement, one fixed fabrication per fact, a same-procedure guarantee across the pre/post comparison, and a three-number comparison that distinguishes an answer losing its edge from an answer being actively suppressed, among others. What the instrument reads out when run against a specific base model and fabricated-fact draw is a separate, empirical question, addressed in work that builds on it rather than in this manuscript.

Author Contributions

All authors contributed to the conception and design of the toolkit described in this manuscript, and to writing and reviewing the manuscript.

Acknowledgments

J.L.V.M. gratefully acknowledges the postdoctoral fellowship received under the Postdoctoral Program in Research Management (PPDG), Office of the Pro-Rector for Research (PRP), University of Campinas (Unicamp), administered through the Center for Energy and Petroleum Studies (CEPETRO).

The authors thank the Power Electronics Laboratories (LEPO) and the Center for Electric Mobility Research (CEMOBE) for institutional and infrastructural support throughout this work.

The authors declare no conflict of interest.

References

- Bender, E. M. and Koller, A. (2020). Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data. In Jurafsky, D., Chai, J., Schluter, N., and Tetreault, J., editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–5198, Online. Association for Computational Linguistics.
- Harnad, S. (1990). The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1):335–346.
- Hicks, M. T., Humphries, J., and Slater, J. (2024). ChatGPT is bullshit. *Ethics and Information Technology*, 26(2):38.
- Li, Y., Bubeck, S., Eldan, R., Giorno, A. D., Gunasekar, S., and Lee, Y. T. (2023). Textbooks Are All You Need II: Phi-1.5 technical report.
- Loshchilov, I. and Hutter, F. (2019). Decoupled Weight Decay Regularization.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.
- Qwen, Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T., Tang, T., Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Wan, Y., Liu, Y., Cui, Z., Zhang, Z., and Qiu, Z. (2025). Qwen2.5 Technical Report.
- Razeghi, Y., Logan IV, R. L., Gardner, M., and Singh, S. (2022). Impact of Pretraining Term Frequencies on Few-Shot Numerical Reasoning. In Goldberg, Y., Kozareva, Z., and Zhang, Y., editors, *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 840–854, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.
- Singh, A. K. and Strouse, D. J. (2024). Tokenization counts: The impact of tokenization on arithmetic in frontier LLMs.

- Stolfo, A., Belinkov, Y., and Sachan, M. (2023). A Mechanistic Interpretation of Arithmetic Reasoning in Language Models using Causal Mediation Analysis. In Bouamor, H., Pino, J., and Bali, K., editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7035–7052, Singapore. Association for Computational Linguistics.
- Verçosa Marques, J. L., Heitmann, F. J., Perez, D. O., Cesar, R., de Paula, M. V., and Barros, T. A. d. S. (2026a). Toolkit for Confidence-Corpus Consistency via Fine-Tuning on a Fabricated Corpus. Zenodo.
- Verçosa Marques, J. L., Heitmann, F. J., Perez, D. O., de Paula, M. V., and Barros, T. A. d. S. (2026b). Technical Manual for a Toolkit for Measuring Contextual Individuation in Transformer Language Models.
- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. (2020). Transformers: State-of-the-Art Natural Language Processing. In Liu, Q. and Schlangen, D., editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.