LANGCHAIN REFACTORING · LEVEL 2

Seeds, *Evaluation* & the Few-Shot Question

// phi3:mini · OllamaLLM · JsonOutputParser · progressive fallback

01 / The Deterministic Seed

THE PROBLEM YOUR CODE ALREADY HAS

You already set `temperature=0` in your model initialisation. Temperature controls creativity versus consistency — at `0`, the model is supposed to always pick the most probable next token. This sounds deterministic, but it isn't always, for a subtle reason.

WHY TEMPERATURE=0 ALONE ISN'T ENOUGH

Even at temperature 0, modern LLMs can produce slightly different outputs across runs due to three things: **floating-point non-determinism** (GPU matrix multiplications don't produce bit-identical results when executed in parallel), **sampling strategies** where some backends apply top-p or top-k filtering even at low temperatures, and **batch effects** when multiple chains run concurrently, as yours do.

WHAT A SEED ACTUALLY DOES

A seed is an integer you pass to the model that initialises the internal **random number generator (RNG)** to a known, fixed state. With the same seed + the same input + the same temperature, the output is guaranteed to be identical across machines and across days.

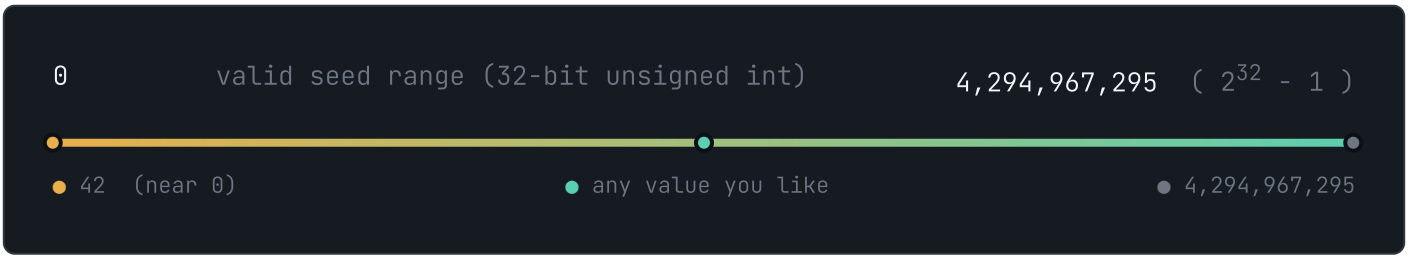
PYTHON

```python
# Before — not fully deterministic
model = OllamaLLM(model="phi3:mini", temperature=0)
```

```
# After — fully deterministic
model = OllamaLLM(model="phi3:mini", temperature=0, seed=42)
```

THE RANGE OF VALID SEED VALUES

Ollama internally uses **llama.cpp**, which stores the seed as a **32-bit unsigned integer**. This means the valid range is:



> ⚠ SPECIAL VALUES TO KNOW
>
> In many backends — including Ollama — passing `seed=-1` or omitting the seed entirely is interpreted as *"use a random seed each time"*, which disables determinism completely. Some systems also treat `seed=0` as "disabled." The safest approach: always provide a positive integer explicitly.

IS THERE ANYTHING INTERESTING ABOUT SPECIFIC SEEDS?

There is — though the effects are model-weight-specific, not universal. A few things worth knowing:

42 is a convention, not a magic number. Its origin is a reference to *The Hitchhiker's Guide to the Galaxy* ("the answer to life, the universe and everything"). It has no special mathematical property; its value is that it's universally recognised, so when you see it in someone's code you immediately know it's a deliberate reproducibility choice, not a forgotten debug value.

Degenerate seeds do exist. Researchers have found that for specific model weights, certain seed values produce pathologically repetitive or incoherent outputs — essentially

the RNG gets stuck in a bad region of probability space. These are rare and model-specific, but they're a real phenomenon. If you ever see a chain that suddenly fails consistently with a specific seed, switching to another (e.g. 123, 7, 2024) is a legitimate debugging step.

Seeds are not portable across model weights. Seed 42 on `phi3:mini` will produce a completely different output than seed 42 on `llama3`. The seed only guarantees reproducibility for the same model loaded identically. This is worth remembering when you later compare models in your evaluation.

WHY THIS MATTERS SPECIFICALLY FOR YOUR CHAIN

Your `progressive_intent_analysis` runs up to three chains on the same text. Without a seed, re-running the script five times to debug could give you `"none"` on run 1 and `"tech_support"` on run 2 with identical code. With `seed=42`, you eliminate that variable entirely — which becomes essential once you introduce the evaluation loop described below.

02 / The Evaluation Dataset

THE PROBLEM WITH YOUR CURRENT TESTING

Right now, you test your chain with one sample text about green bonds. You run it, look at the output, and think "looks right." This is called **eyeballing**, and it doesn't scale. You have no idea whether your chain correctly identifies billing questions, or whether it confuses `"sales"` with `"tech_support"`. You're flying blind.

WHAT AN EVALUATION DATASET IS

An evaluation dataset — also called a *golden dataset* or *test set* — is a collection of **(input, expected output)** pairs that you have manually verified. Once you have one, you can run your chain against all of them automatically and measure how often it's correct.

```python
EVAL_DATASET = [
    {"text": "My invoice shows a charge I don't recognize.",  "exp
    {"text": "My internet connection keeps dropping every few hour
    {"text": "I'm interested in upgrading to a business package.",
    {"text": "The green bond market has grown rapidly...",
    {"text": "I was overcharged for the premium tier last cycle.",
    {"text": "How do I reset my router to factory settings?",
]
```

THE EVALUATION LOOP

```python
def evaluate_chain(chain, dataset):
    correct = 0
    total   = len(dataset)

    for item in dataset:
        result    = chain.invoke({"text": item["text"]})
        predicted = result.get("intent")
        expected  = item["expected_intent"]

        is_correct = predicted == expected
        correct   += is_correct

        status = "✔" if is_correct else "✖"
        print(f"{status}  Expected: {expected:12} | Got: {predicte

    accuracy = correct / total * 100
    print(f"\nAccuracy: {correct}/{total} = {accuracy:.1f}%")
    return accuracy

# Usage
```

```
evaluate_chain(intent_only_chain, EVAL_DATASET)
```

This produces output like:

```
✔ billing          match
✔ tech_support     match
✖ sales            none ≠ sales
✔ none             match

Accuracy: 3/4 = 75.0%
```

🔗 CONNECTION TO THE DETERMINISTIC SEED

This is why both concepts are paired. Without `seed=42`, running `evaluate_chain` twice could give you 83% then 67% — and you can't tell if a code change *caused* the improvement or if you just got lucky. With a fixed seed, accuracy becomes a **stable metric**: when it goes from 67% to 83% after a refactor, that change is real.

03 / Is This Few-Shot Prompting?

Sharp question — and the answer is: **no, but they're related**. These are two distinct techniques that work at completely different stages. The confusion is understandable because both involve "example inputs and expected outputs." The difference is *where* those examples live and *what they do*.

🔷 FEW-SHOT PROMPTING

- → Examples live **inside the prompt**
- → Runs **before** inference

🔶 EVALUATION DATASET

- → Examples live **outside the prompt**
- → Runs **after** inference

→ Purpose: **teach the model** by demonstration

→ Model sees these examples every call

→ A prompting technique

→ Purpose: **measure the model**'s accuracy

→ Model never sees these examples

→ A measurement technique

FEW-SHOT PROMPTING — WHAT IT ACTUALLY LOOKS LIKE

In few-shot prompting, you modify the *prompt itself* to include examples of correct input/output pairs. The model learns the pattern from those examples and applies it to the new input.

PYTHON

```
few_shot_prompt = PromptTemplate(
    template="""
Classify the intent of customer messages. Examples:

Text: "My bill shows an extra charge."
Intent: billing

Text: "My router stopped connecting after the update."
Intent: tech_support

Text: "I want to know your enterprise pricing."
Intent: sales

Now classify this:
Text: {text}
{format_instructions}
""",
    input_variables=["text"],
    partial_variables={"format_instructions": intent_parser.get_fo
)
```

Those three examples (*"bill," "router," "enterprise pricing"*) are the **shots**. Zero-shot means no examples — which is exactly what your current code does. One-shot means one example. Few-shot typically means 2–5.

HOW THE TWO CONCEPTS WORK TOGETHER

Here's where they connect in practice: you can use your **evaluation dataset to find the best few-shot examples**. You run the evaluation without any examples (zero-shot). You identify which intent categories the model gets wrong most often. You then add one or two examples for those weak categories into the prompt, and re-run the evaluation to confirm the accuracy went up.

💡 THE WORKFLOW IN YOUR CONTEXT

Step 1 — Run `evaluate_chain(intent_only_chain, EVAL_DATASET)` → get baseline accuracy (zero-shot).

Step 2 — Notice that `"sales"` is often misclassified → add a few-shot example for it inside the prompt.

Step 3 — Re-run evaluation with the new few-shot prompt → confirm accuracy improved. The seed ensures any improvement is real.

📌 THE KEY DISTINCTION

Few-shot examples are **part of the prompt** — the model sees them and uses them to calibrate its behaviour in real time. Evaluation examples are **never shown to the model** — they are used only by your code, after inference, to judge whether the model was right. Mixing them up (using your eval examples as few-shot examples) would invalidate your accuracy measurement, because you'd be testing the model on examples it effectively trained on.

THE FORMAL NAMES IN ML

What you're building here is called a **test set** (the eval dataset) evaluated under **zero-shot conditions** (no examples in the prompt). Few-shot prompting is a technique to improve zero-shot performance. LangSmith, RAGAS and DeepEval are production frameworks built entirely around this idea — you're constructing the mental model by hand first, which is exactly the right order.