# Programming Large Language Models with Algebraic Effect Handlers and the Selection Monad

Anonymous Author(s)

## Abstract

We present Pangolin, a programming language that treats large language model (LLM) interactions as first-class algebraic effects, enabling systematic composition and control over complex AI systems. By modeling non-deterministic choices of LLM results with selection monads, Pangolin allows programmers to abstract over multiple execution paths and automatically select outcomes based on downstream metrics. Pangolin makes it possible to design and manage LLM-centric pipelines with the clarity and reliability of conventional programming languages.

## 1 Introduction

The standard way of interacting with large language models (LLMs) is to treat LLMs as a black box function with token inputs and outputs. This abstraction naturally arises from the autoregressive nature of LLMs, which generate text by predicting one token at a time based on the preceding context. While this abstraction suffices for simple single-turn interactions, modern AI applications increasingly rely on compound AI systems that orchestrate multiple LLM calls, integrate external tools, and maintain complex state across interactions.

To address these challenges, numerous frameworks have emerged, such as DSPy [3], Pydantic AI [8], and LangChain [1]. These frameworks provide better abstractions for building compound AI systems, typically offering prompt templating, output parsing, LLM call composition, and retry logic. All the abstractions provide more pleasant ways to interact with LLMs through declaratively defined *language programs*, where language model interactions are defined programmatically as deeply embedded modules. However, the abstraction comes at a cost: fine-grained control over the generation process is sacrificed for ease of use. Now, even the compound AI system becomes a black box, with no visibility into intermediate results and how the declarative program being compiled to the token-in and token-out interactions with large language models.

In this paper, we propose Pangolin, a novel programming model to interact with LLMs through algebraic effects and handlers with selection monads. Pangolin's formalization maintains a clean separation between what the *language programs* do through algebraic effect operations and how it is achieved through compositional effect handlers and the selection monad [2, 4].

Pangolin leverages the well-established theory of algebraic effects [5] and handlers [6] to provide a principled way for composing compound AI systems. Algebraic effects offer a modular approach to computational effects, where effectful operations are declared separately from their interpretation, enabling the same program to be executed with different strategies by swapping handlers. This separation is particularly powerful for LLM programming: the same *language model interaction* can be interpreted by various handlers to achieve different execution strategies—a simple handler might directly call OpenAI's API, while a non-deterministic handler could generate multiple candidates in parallel for test-time scaling, with the same, or even different model providers.

In addition, we integrate the selection monad in Pangolin to capture the inherently non-deterministic nature of language model generation and enable principled exploration of the output space with reward. The selection monad extends traditional nondeterministic effects by associating scores with each outcome, allowing programs to reason about the quality of different execution paths before committing to a final output.

Building on this foundation, Pangolin offers several key advantages over existing approaches. First, Pangolin provides fine-grained control over the generation process while maintaining high-level abstractions. Unlike traditional frameworks that hide the complexity behind opaque interfaces, Pangolin exposes intermediate results and decision points through the algebraic effect system, enabling developers to inspect, modify, and optimize the execution flow. Second, Pangolin establishes a potential formal foundation for reasoning about compound AI systems through its types and semantics. Third, Pangolin's algebraic structure makes it well-suited as a code generation target for LLM models. The explicit separation between effect operations and handlers provides LLMs with a structured framework for generating compound AI systems, rather than navigating the complex callback chains and implicit dependencies common in traditional frameworks. When generating Pangolin programs, LLMs can compose simple effect operations following the semantic rules, reducing the occurrence of subtle bugs that often appear in LLM-generated framework code.

**Contributions.** We sketch the design of Pangolin, a programming language for compound AI systems based on algebraic effects and handlers integrated with selection monads. We demonstrate several examples of interesting language model programs in Pangolin, including test-time scaling techniques (best-of-n sampling) and different retry strategies.

## 2 Example PANGOLIN Programs

We introduce PANGOLIN and explain its features with several examples. Figure 1 is a simple LLM program that performs sentiment analysis. We define a function `emotion_classifier` that takes a sentence `str` as input and returns a sentiment `str` as output.

```
let emotion_classifier = λ sentence.
  let emotion_spec = specification {
    input = { sentence: str },
    output = { sentiment: str }
  };
  let input = { sentence = sentence };
  let result = LM(emotion_spec, input);
  result.sentiment
let result = emotion_classifier("I love you!")
```

**Figure 1.** PANGOLIN program does sentiment analysis on a sentence.

The function firstly *specifies* the protocol of interacting with the LLM, using PANGOLIN construct `specification`. Rather than treating LLM calls as opaque string-to-string functions, specifications provide typed interfaces that declare the expected input structure and promised output format. Specification type is a more compact representation of DSPy's *signature* system [3], enabling *declarative* interactions with LLMs. The specification type can also be easily extended to include: prompt templates, demonstrations, traces, and even models and weights.

Then, the function `emotion_classifier` prepares the input, invokes the LLM, and extracts the output. Here, `LM` is a PANGOLIN built-in algebraic effect, representing an *abstract* effect operation that is not yet interpreted. To execute the program, we need an effect handler that defines the semantics (i.e., the actual action) of this operation. Installing different effect handlers effectively gives the same program different behaviors at runtime, separating concerns of *what* and *how*, and bringing flexibility and modularity in constructing LLM-programs with complex interactions.

Figure 2 defines the most naive LM handler. The handler intercepts the `LM` operation with its arguments and continuation `k`. In the handler, we use primitive `write` that formats specifications into prompts (of type string), call the primitive LM function `primlm: string -> string`, parse results back to a typed record using primitive `read`, and resume with the parsed output by invoking the continuation `k`.

### 2.1 Test-time Scaling

A common strategy to improve LLM inference is through test-time scaling [10]. To implement a simple parallel-sampled test-time scaling for `emotion_classifier` in Figure 1, we can install a different handler for it. In `parallel_lm_handler(n)` (Figure 3), we generate `n` samples, apply the continuation to each

```
let naive_lm_handler = {
  | LM(spec, input; k) ↦
    let prompt = write[spec](input);
    let raw_result = primlm(prompt);
    let output = read[spec.output](raw_result);
    k(output)
};

handle emotion_classifier("I love you!")
  with naive_lm_handler
```

**Figure 2.** PANGOLIN program with a naive LM handler, which formats specifications into prompts, calls the primitive LM function, parses results back to typed records, and continues with the parsed output.

```
let parallel_lm_handler(n) = {
  | return x ↦ [x]
  | LM(spec, input; k) ↦
    let prompt = write[spec](input);
    let raw_results = map(
      λ i. primlm(prompt), [1..n]
    );
    let parsed_outputs = map(
      λ result. read[spec.output](result),
      raw_results
    );
    fold(++, [], map(k, parsed_outputs))
};

handle emotion_classifier("I love you!")
  with parallel_lm_handler(16)
```

**Figure 3.** PANGOLIN program with test-time scaling handler that generates n parallel samples by calling the language model multiple times, applies the continuation to each individual result, and concatenates all results together.

of the samples, and combine the results into a list. Note that here, we need to provide the return clause `return x ↦ [x]`. In this way, the final results of each continuation `k` will be a singleton list, which can be combined into a final result list.

The parallel LM handler here models a special form of the nondeterminism effect [7], which explores the program with multiple choices (paths) simultaneously. Beyond test-time scaling, readers can imagine that this becomes particularly handy when the language program explores different LM specifications as well. For example, another LM handler can call the same program with different language models (e.g., with GPT 4.1 and Claude-Sonnet-4 ) through invoking the continuation `k` with corresponding specifications.

### 2.2 Best-of-n Sampling

To select a single result from the list, we utilize the selection monad with the `choose` operation (Figure 4). Instead of

```
let best_of_n_handler(n) = {
  | LM(spec, input; k) ↦
    // same as before
    ...
    let parsed_outputs = ...
    let best = choose(parsed_outputs);
    k(best)
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle
  let res = emotion_classifier("I love you!");
  score confidence_score(res)
with best_of_n_handler(16)
```

**Figure 4.** Pangolin program with best-of-n selection that generates n samples in the LM handler, uses choose to select among them based on scoring, and returns the highest-scoring result.

returning the list from parallel-sampled language model outputs, `LM` handler calls `choose` at the end with a list of options.

The `choose` handler applies the special *score continuation* $k_s$ to the list of options, which automatically collects the score in the rest of the program for each option and returns the sample with the highest score. The notion of score can be customized by the user, for example, Figure 4 rates the output using `confidence_score`. The underlying implementation can be as simple as string matching, or more complex, like using LLM as a judge or invoking some external services for feedback. Finally, `LM` resumes the program execution with the highest-scoring sample.

## 2.3 Retry with Assertion

LLM programs are still prone to failure, e.g., when the output fails to meet a required format or constraint. To guard against such issues, users can insert `assert` statements to specify output expectations. Unlike traditional assertions which halt execution upon failure, Pangolin 's assertions are algebraic operations that can instead be reinterpreted. In addition, Pangolin's assertion is a more general form of DSPy assertions [9], which are interpreted with only one retry mechanism.

Figure 5 demonstrates a primitive use of `assert` to implement a retry mechanism, when the condition `pred` is not satisfied. To capture the context to backtrack to, Pangolin introduces a `retry` block. Each `assert`'s handler will have access to the most immediate `retry`'s context as `retry_ctx`, which allows backtracking and accessing the last invoked argument of this block.

The handler implements backtracking by conditioning on the value of `pred` or the number of times the `retry` block

```
let naive_retry_handler(n) = {
  | assert(pred, feedback; k; retry_ctx) ↦
    let n_retry = retry_ctx.n_retry;
    if pred or retry >= n then k()
    else retry_ctx(feedback, n_retry+1)
};

handle retry(feedback = "", n_retry = 0) {
  let res = emotion_classifier("I love you!");
  assert(
    res ∈ ["positive", "negative", "neutral"],
    "Expected positive, negative, or neutral"
  );
} with naive_retry_handler(3) | naive_lm_handler
```

**Figure 5.** Pangolin program with naive retry handler that retries the entire block on assertion failure by incrementing the retry counter and calling the retry continuation. We use `handle e with` $h_1$| $h_2$ to denote the composition of two handlers (i.e., `handle handle e with` $h_1$ `with` $h_2$).

was executed. If `pred` satisfies the check or when we exhaust the backtracking budget `n`, the handler continues normal execution flow by calling the continuation. Otherwise, we backtrack to the `retry` block with assertion feedback and increment the retry counter by 1.

While `naive_retry_handler` implements the backtracking mechanism to improve LM generation quality, we can easily alter the behavior of `assert` by installing a different handler. For example, the following handler implements conventional assertion handling that halts the execution when failing:

```
let normal_assert_handler = {
  | assert(pred, feedback; k; retry_ctx) ↦
    if pred then k()
    else halt(feedback);
};
```

## 2.4 Retry with the Highest Score

We can also implement `assert` as intermediate reward signals for more efficient sampling (Figure 6). The handler implementation is as simple as interpreting `pred` with `score`. Then, the selection monad is able to collect these assertion rewards along with the program's execution, returning samples that pass the most number of assertions (or achieve the highest score, if we allow `pred` to be arbitrary floating-point numbers).

## 3 Conclusion and Future Work

We present a programming language Pangolin to interact with language models programmatically. We also showcase programs in Pangolin with flexible handlers for various language program behaviors.

A future work is to develop the formal semantics of Pangolin and a type system for it. Pangolin also needs additional features to become useful. In the previous examples,

```
let choose_handler = {
  | assert(pred, feedback; k; retry_ctx) ↦
    score(pred);
    k()
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle retry(feedback = "", n_retry = 0) {
  let res₁ = choose(emotion_classifier("I love
      you!"));
  assert(
    res ∈ ["positive", "negative", "neutral"],
    "Expected positive, negative, or neutral"
  );
  assert(len(res) < 10;
    "Sentiment should be short.")
} with choose_handler | parallel_lm_handler(5)
```

**Figure 6.** We can easily handle a similar program to Figure 5 with a `choose` handler to use assertion as score signals, and use the selection monad to pick the best one. This works with an arbitrary number of assertions as well.

we presented a simplified `LM` specification and handler. In reality, the specification needs actual language model configurations to properly set up `primlm` before invoking the right model with the correct parameters. Another practical way to improve PANGOLIN is by handling asynchronous language model operations. As LLM inferences are often time-consuming, a practical framework often streamlines large-scale LM program evaluation/serving through async LM calls.

## References

[1] Harrison Chase. 2022. LangChain: Building applications with LLMs through composability. https://github.com/langchain-ai/langchain. Accessed: 2025.

[2] Martín Hötzel Escardó and Paulo Oliva. 2010. Selection functions, bar recursion and backward induction. *Math. Struct. Comput. Sci.* 20, 2 (2010), 127–168. doi:10.1017/S0960129509990351

[3] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] https://arxiv.org/abs/2310.03714

[4] Gordon Plotkin and Ningning Xie. 2025. Handling the Selection Monad. *Proc. ACM Program. Lang.* 9, PLDI, Article 218 (June 2025), 25 pages. doi:10.1145/3729321

[5] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.* 11, 1 (2003), 69–94.

[6] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013).

[7] Gordon D Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). doi:10.2168/lmcs-9(4:23)2013

[8] Pydantic Team. 2024. Pydantic AI: Agent Framework / LLM Toolkit for Python. https://ai.pydantic.dev. Accessed: 2025.

[9] Arnav Singhvi, Manish Shetty, Shangyin Tan, Christopher Potts, Koushik Sen, Matei Zaharia, and Omar Khattab. 2024. DSPy Assertions: Computational Constraints for Self-Refining Language Model Pipelines. arXiv:2312.13382 [cs.CL] https://arxiv.org/abs/2312.13382

[10] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters. arXiv:2408.03314 [cs.LG] https://arxiv.org/abs/2408.03314