DSRs: EXECUTION GRAPH OPTIMIZATION FOR LLM PROGRAMS IN RUST

Herumb Shandilya
Stanford University

Jon Saad Falcon
Stanford University

ABSTRACT

Building LLM pipelines often relies on fragile prompt chaining, and unlike Python, the Rust ecosystem lacks utilities for DSPy-like declarative prompting. DSRs is a Rust reimplementation of DSPy that addresses this by providing an intuitive API for constructing and optimizing pipelines, while leveraging systems-level performance. DSRs introduces optimizations that reduce I/O latency, improve memory access, and utilization. Beyond these improvements, it treats pipelines as computational graphs, converting declarative specifications into an intermediate representation that enables graph fusion, cross-module optimization, and global execution planning. To achieve this, DSRs introduces a reference-driven compilation, where intermediate modules are compiled and validated against reference plans generated by a meta planner based on Hu et al. (2025) Zhang et al. (2025). We focus on programmer-centric I/O optimization, the main overhead in declarative prompting, while introducing methods/interfaces that advance research toward true compilation of LLM pipelines.

1 METHODOLOGY

DSRs enables you to programatically declare prompts and design the LLM workflow for the task. We take inspiration from Khattab et al. (2023) Python framework to take the design decisions for building the DSRs crate. We break down our roadmap into three steps:

1. **Building the Foundation Crate:** Implement core abstractions for declarative prompting, module traits, data and signature macros, tooling support and optimzation traits.
2. **Optimize Crate Performance:** Integrate memory-efficient data structures, asynchronous I/O, caching, and improved and improve throughput.
3. **Dynamic Workflow Optimization:** Enable runtime introspection and adaptive recompilation of pipelines based on execution traces and meta-planner feedback.

1.1 DSRS CRATE

DSRs takes inspiration from Khattab et al. (2023) for it's design of high level abstraction working. A typical DSRs task flow looks like this:

1. Load data as `Vec<Example>` via `dsrs::DataLoader`, which allows data loading from sources like CSV, JSON(L), Parquet and HuggingFace.
2. Configure `LM` and `Adapter` to `GLOBAL_SETTING`. While DSRs provides option to define `LM` to be used for predictor at Predictor level we expose a global singleton setting that can be configured to define `LM` and `Adapter` than need to be used by Predictors by default.
3. Define Signatures. Declare the task as a `dsrs::Signature` by defining instruction, inputs and outputs of the task.
4. Define Module. Specify the flow of data in you LLM Workflow which comprise of Predictors execution, data transformation etc.
5. Evaluate and Optimize. Optionally you can evaluate or optimize your Module over a dataset via `dsrs::Evaluator`, `dsrs::GEPA`, etc.

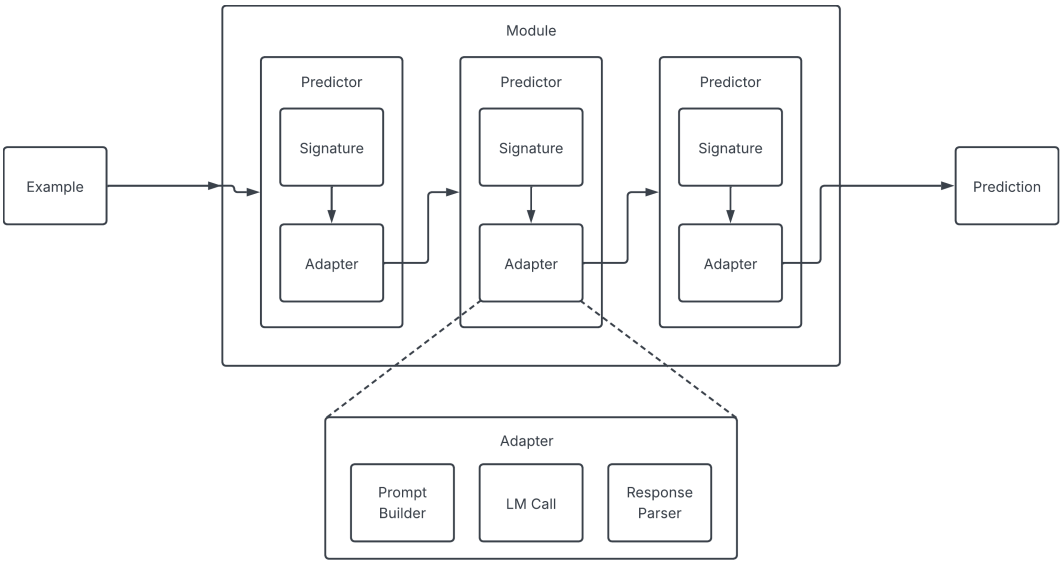

Figure 1: Module Execution in DSRs

The execution flow of a `dsrs::Module` is defined in Figure 1. At core DSRs has 5 main components during execution:

Data. DSRs has two core data containers namely `Example` and `Prediction`. Each Predictor and Module takes in `Example` as input and returns `Prediction` as output with usage metadata. Each of them store a one hashmap-like datapoint as `serde::Value` to support flexibility.

```
/// Defining single example
let example = example! {
    "question": "input" => "Dummy Text",
};

/// Loading from Source
let examples = DataLoader::load_hf(
    "hotpotqa/hotpot_qa",
    vec!["question".to_string()],
    vec!["answer".to_string()],
    "fullwiki",
    "validation",
    true,
)
```

Listing 1: Creating Example and Loading from sources

Signatures Signatures in DSRs are syntactically similar to Khattab et al. (2023) however unlike DSPy we don't treat prompt modifiers in runtime via predictors, like `dspy.ChainOfThought`, and instead apply the necessary modifications in Signature like these during compile via macros. We also support inline signatures like DSPy, however unlike it DSRs inline signatures are type strict and not type-inferred.

```
/// Inline Signatures
let sgn = sign! {
    (number: i32) -> number_squared: i32,
                     number_cubed: i32
}


/// Struct Based Signature
```

```rust
#[Signature(cot)]
struct QASignature {
    /// Concisely answer the question

    #[input]
    pub question: String,

    #[output(desc = "less than 5 words.")]
    pub answer: String,
}

let sgn = QASignature::new()
```

Listing 2: Defining Signatures

Predictors In DSRs, Predictors represent how the LM call will be tackled. To be precise, as of now DSRs only has `dsrs::Predict` which is a vanilla predictor that takes input, makes LM call and return the output as it is. However, we can create Predictors that execute the input in a Yao et al. (2023) like fashion or Refine output via additional calls. Internally, `dsrs::Predict` calls the configured adapter that is reponsible for converting Signature to prompt and parsing the output of the call.

```rust
let predict = Predict::new(sgn);

/// Execution of predictors is an async
/// method named forward
let op = predict.forward(example).await?;
```

Listing 3: Declaring Vanilla Predictor and Executing it

Adapter Adapters are responsible for ensuring the structure of the input (from the signature) is specified in the desired format prior to the LM call, and to parse the output from the LM. An adapter defines the methods that construct the prompt and enable the module to carry out its task. As of now, DSRs only provides the default `dsrs::ChatAdapter`, which uses JSON schema to format the input and output types from the signature prior to calling the LM. In the future, we aim to extend the adapters to specify more formats based on BAML, Xgrammer, TOON and GBNF backends. Along with constructing the prompt template and parsing the result, adapters internally make the call to the LM APIs via the configured LM struct.

LM LM in DSRs takes raw prompt as input and outputs the raw response from the API without any transformation or filtering. Internally, we use Rig Framework to tackle all the LM calls. Since there are multiple clients in Rig per provider we use `enum_dispatch` over dynamic dispatch and expose only 6 providers to the user natively with a Bring Your Own Client(BYOC) interface. This is pure to improve the call latency and is discussed more in detail in Section 1.2.2.

```rust
configure(
    LM::builder()
        .model("openai:gpt-4o-mini".to_string())
        .build()
        .await
        .unwrap(),
    ChatAdapter,
);
```

Listing 4: Configuring LM & Adapter as Global Setting

1.2 Performance Optimization

Once we had the crate components in place we wanted to improve the IO bottlenecks that happen during the execution of an LLM Pipeline.

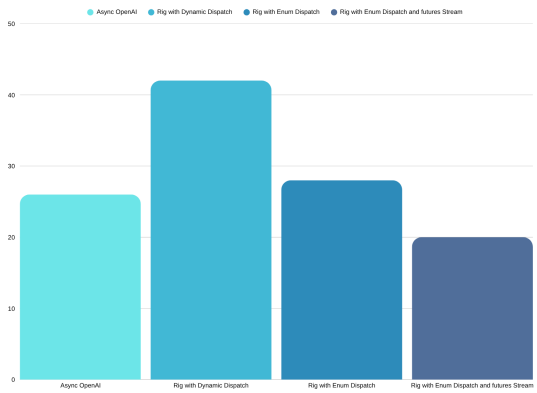

Figure 2: LM Backend Crate Comparisons

1.2.1 LM BACKEND

To call the LLM providers from different clients we tested out two options available to us:

1. **`async-openai` Crate:** The very first backend for `LM` we used involved using OAI compatible endpoints from provider and using them with an OAI crate to fetch the output from an LLM call. This was working well for us and was a fast crate to use, however the lack of native provider support quickly started to hurt especially when it came to Azure OAI. Aside from this the tool orchestration for OAI was bare minimum and so we wanted a crate that'll provide support calling to different clients and also provide robust support for tool handling.
2. **`rig` Crate:** To tackle this we decided to go with rig which is a popular rust crate for building LLM applications. We tested two ways to use `rig` integration: dynamic dispatch with `DynClientBuilder` wrapping clients in `enum_dispatch`.
    - `DynClientBuilder`: This allowed us to use and resolve LLM clients on the crate end by passing model names in conjunction with the provider(i.e. `openai:gpt-4o-mini`). While this provided us with an easy to use abstraction to resolve and use clients from different providers, we doubled the latency we got in `async-openai` and for reasons unknown a bad performance on HotpotQA. The latency issue was because internally `DynClientBuilder` uses a lot of Boxing(heap allocation) and Dynamic Dispatch which resolves Client at runtime. This makes the usage really slow and so we moved to resolve clients at runtime.
    - `enum_dispatch`: To resolve the clients at compile time we exposed six most popular providers in Rig at runtime with a BYOC support. This made our latency comparable to the `async-openai` crate and provided improved utility for tools handling. In the same time, when doing inference on a big batch in async via `dsrs::Evaluator` we still were doing slightly worse then `async-openai` but not by a big margin. We found that the async execution was being handled in batches, so we replaced that with async stream and significantly improve the latency. This was a crate agnostic improvement and would apply to async-openai as well, since we resolved this with the `enum_dispatch` we decided to mention it here. (Refer 2)

1.2.2 CACHING

With a stable LM backend in place we wanted to implement caching to reuse the call results mainly during optimization in case the instructions are duplicated across batches. To cache we could use either `quickcache` crate which has the fastest in-memory cache or `foyer` crate which has a comparable in-memory cache and a hybrid cache. We ended up going with `foyer` given the scale of cache we are working with.

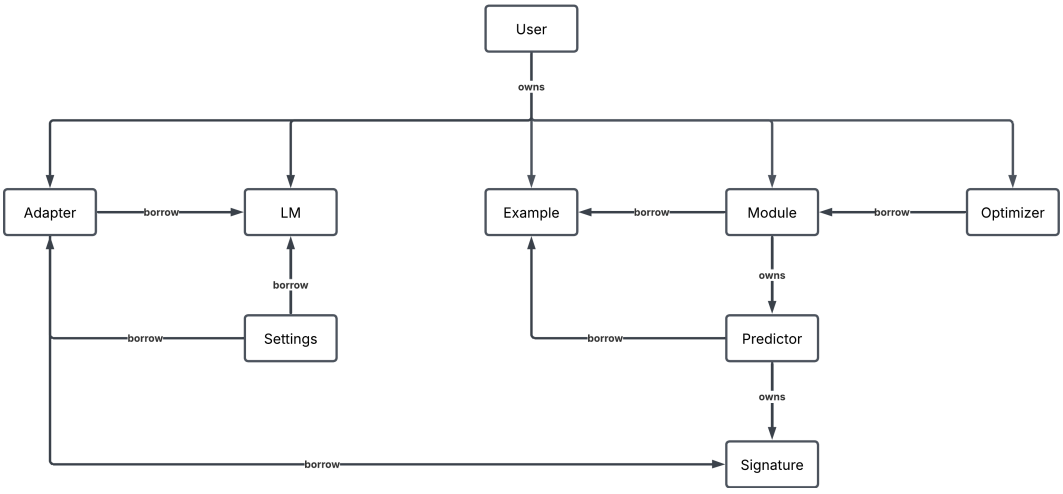

Figure 3: Ownership Architecture Diagram

1.2.3 Optimizing Critical Section

With cache and backend in place it was time to improve the way we handled LM backend. The LM object, unless specified, is owned by `GLOBAL_SETTING` as an `Arc<Mutex>`. This means LM object can be safely shared and mutated across thread. The downside of this approach is that the threads would acquire the lock to even use LM object. The reason we needed mutability on LM object was to update the `History` object containing the previous calls in LLMs. So essentially we ideally want the lock around the cache upadtion and history updation.

We decided to get rid of locking completely by merging History and Cache into one and utilize cache to return the history to the user. Aside from this we treat caching as a background task where the cache updates are transmitted(`tokio::mpsc`) by the threads and received(`tokio::mpsc`) by the handler via `oneshot::channel`. This enabled lazy updation of the cache in the background since history is not a frequently used operation.

There is another alternative where we can amortize the caching where we update the cache all at once when the history is called by the user. We did not explore this alternative because we believe updation in background is a better setting for task of this nature.

1.2.4 Designing Ownership Architecture

To make DSRs both safe and performant, we had to carefully design how data and components are "owned" during execution. Rust enforces a strict ownership model, meaning that at any given time, each piece of data has exactly one owner, and other parts of the program can only access it through controlled borrowing or cloning. This guarantees memory safety but introduces practical challenges when building concurrent systems like ours, where multiple modules might need to read or modify shared objects such as the language model or cache.

Early in development, we used aggressive cloning of shared objects to avoid ownership conflicts. While this simplified implementation, it created unnecessary cloning overhead which is cheaper in Rust but not free. To address this, we gradually transitioned toward using shared smart pointers (e.g., Arc) and lightweight synchronization primitives. Figure 3, illustrates how ownership and borrowing are distributed across the DSRs components like, predictors and Adapters are shared through borrowing, while user owns modules which own their local predictors and signatures.

1.3 Dynamic Workflow Optimization

DSPy optimizes user-defined workflows through sampling and example based adaptation, the structure of its pipeline however remains static. In contrast, direction for DSRs is to enable true **structural optimization** where the flow of execution itself can be optimized for. Instead of treating

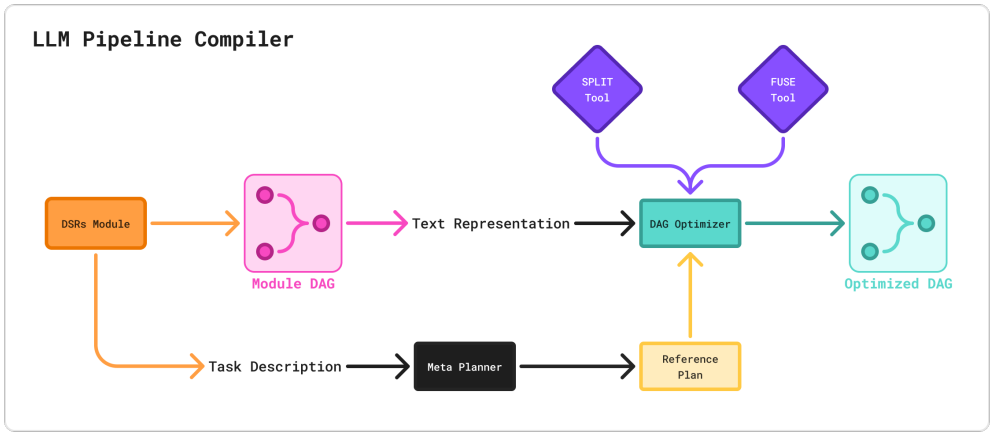

Figure 4: Dynamic Workflow Optimization Process

modules as immutable blocks, DSRs will parse each `Module` into an executable Directed Acyclic Graph (DAG) representation that explicitly captures predictor dependencies, data transformations, and control flow.

Once the pipeline is represented as a graph, DSRs can apply graph-level transformations such as **Split** and **Fuse** via tool calls. The **Split** transformation will decompose complex signature defined task into smaller signatures. Conversely, **Fuse** will combine compatible signatures into unified signature to reduce redundant I/O and model calls. These transformations can be guided by a meta planner that evaluates performance traces and reference plans, determining when reconfiguration would yield performance gains (Figure 4).

In essence, this phase of DSRs aims to evolve static optimization into a dynamic optimization process. By interpreting LLM workflows as executable graphs rather than static graphs, the system can learn to restructure itself to optimize the metric.

2 Data and Evaluation

We use the HotpotQA dataset Yang et al. (2018) for evaluation and optimization. HotpotQA provides a cost-efficient benchmark for multi-hop question answering, making it suitable for validating declarative pipeline performance under constrained settings.

We use its train split for optimization, while the validation split is used for evaluation. HotpotQA is chosen for its accessibility, moderate difficulty, and low cost of experimentation, allowing consistent and reproducible assessment of system performance.

3 Progress Report

We have completed all major components outlined in the Section 1 except the worflow optimization, including the foundation crate, performance optimization, and architecture for declarative prompting. The current system supports data loading, signature and module definitions, predictor execution, and global configuration for language models and adapters, along with optimizations for caching, asynchronous execution, and I/O efficiency.

The next stage of development focuses on building the compiler layer of DSRs. This phase will introduce a model-to-DAG translation layer, converting declarative module definitions into a graph-based intermediate representation. Subsequent components will include DAG-to-prompt compilation, a meta planner for execution scheduling, and the implementation of Split and Fuse tools for structural optimization.

4 ACKNOWLEDGEMENT

This project was built with the support of:

- **Maguire Papay**, Founder of HokyoAI: provided guidance and feedback on performance optimization strategies and abstraction design.
- **Prashanth Rao**, AI Engineer at LanceDB: helped with explanation review, and battle testing the framework.
- **Josh**, Core Maintainer of Rig: provided guidance in integrating the Rig framework with DSRs.

REFERENCES

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2025. URL https://arxiv.org/abs/2408.08435.

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. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023. URL https://arxiv.org/abs/2310.03714.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering, 2018. URL https://arxiv.org/abs/1809.09600.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2025. URL https://arxiv.org/abs/2505.22954.