

000 DSRs: EXECUTION GRAPH OPTIMIZATION FOR LLM 001 PROGRAMS IN RUST 002

003 **Anonymous authors**
004
005 Paper under double-blind review
006

007 ABSTRACT 008

009 Building LLM pipelines often relies on fragile prompt chaining, and unlike
010 Python, the Rust ecosystem lacks utilities for DSPy-like declarative prompting.
011 DSRs is a Rust reimplementation of DSPy that addresses this by providing
012 an intuitive API for constructing and optimizing pipelines, while leveraging
013 systems-level performance. DSRs introduces optimizations that reduce I/O
014 latency, improve memory access, and utilization. Beyond these improvements,
015 it treats pipelines as computational graphs, converting declarative specifications
016 into an intermediate representation that enables graph fusion, cross-module op-
017 timization, and global execution planning. To achieve this, DSRs introduces a
018 reference-driven compilation, where intermediate modules are compiled and val-
019 idated against reference plans generated by a meta planner based on Hu et al.
020 (2025) Zhang et al. (2025). We focus on programmer-centric I/O optimization,
021 the main overhead in declarative prompting, while introducing methods/interfaces
022 that advance research toward true compilation of LLM pipelines.
023

024 1 METHODOLOGY 025

026 DSRs enables you to programatically declare prompts and design the LLM workflow for the task.
027 We take inspiration from Khattab et al. (2023) Python framework to take the design decisions for
028 building the DSRs crate. We break down our roadmap into three steps:
029

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

036 1.1 DSRS CRATE 037

038 DSRs takes inspiration from Khattab et al. (2023) for it's design of high level abstraction working.
039 A typical DSRs task flow looks like this:
040

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

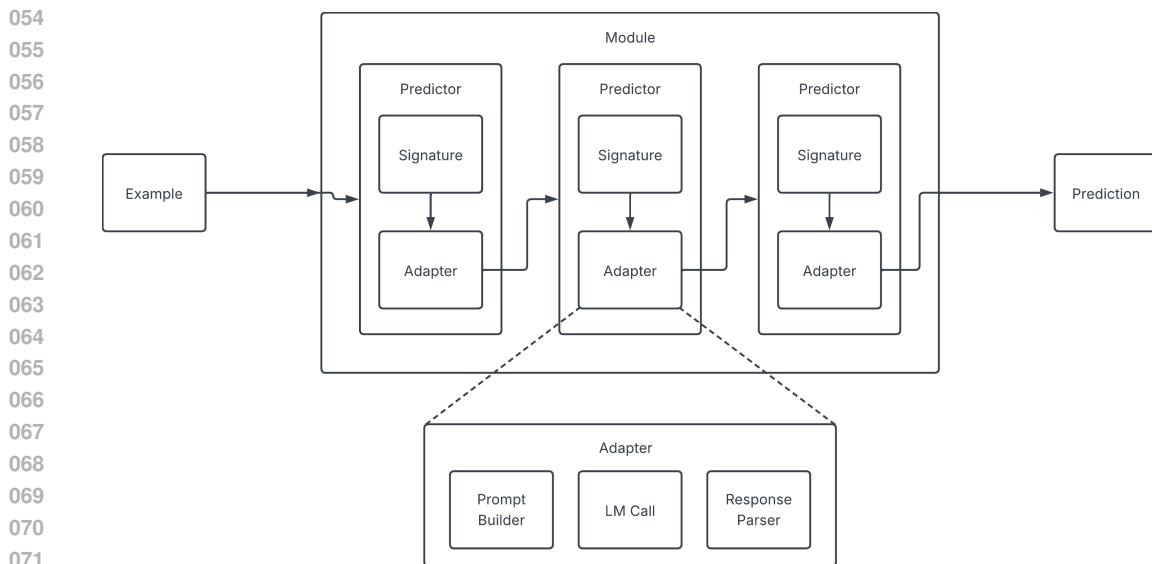


Figure 1: Module Execution in DSRs

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

```

```

108 #[Signature(cot)]
109 struct QASignature {
110     /// Concisely answer the question
111
112     #[input]
113     pub question: String,
114
115     #[output(desc = "less than 5 words.")]
116     pub answer: String,
117 }
118
119 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 responsible for converting Signature to prompt and parsing the output of the call.

```

128 let predict = Predict::new(sgn);
129
130     /// Execution of predictors is an async
131     /// method named forward
132     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, Xgrammar, 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.

```

149 configure(
150     LM::builder()
151         .model("openai:gpt-4o-mini".to_string())
152         .build()
153         .await
154         .unwrap(),
155     ChatAdapter,
156 );

```

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.

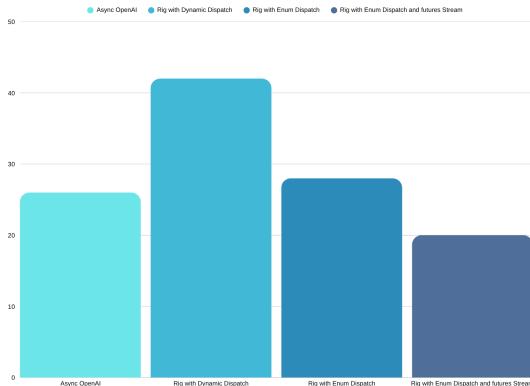


Figure 2: LM Backend Crate Comparisons

1.2.1 LM BACKEND

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

- 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.
- 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 `oyer` crate which has a comparable in-memory cache and a hybrid cache. We ended up going with `oyer` given the scale of cache we are working with.

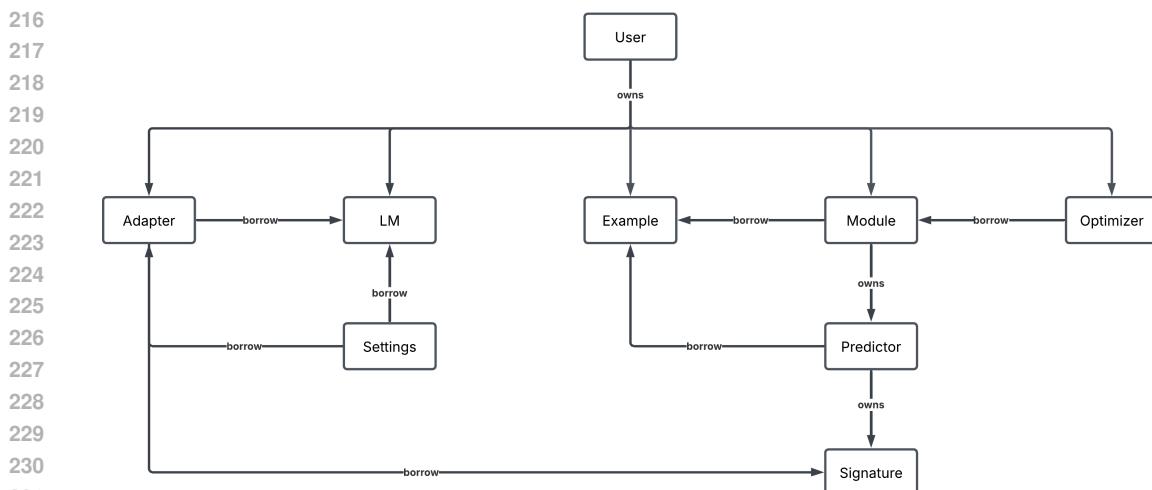


Figure 3: Ownership Architecture Diagram

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 updatation.

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 updatation 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 updatation 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

270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286

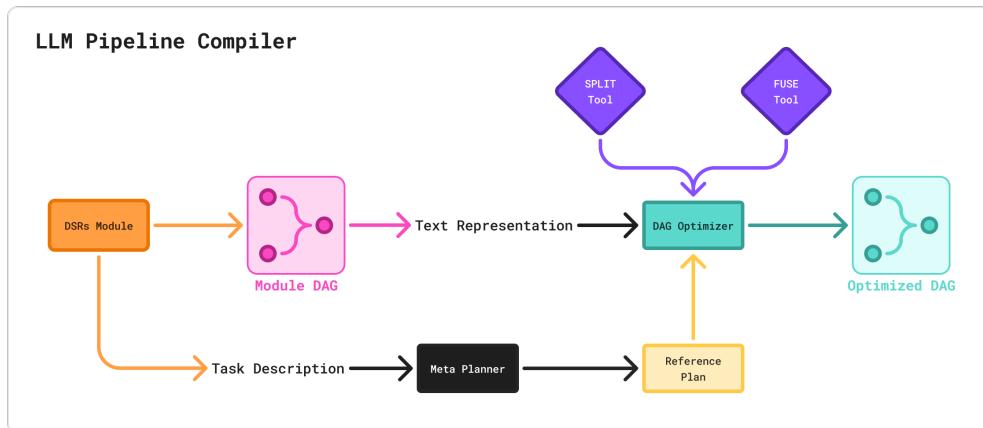


Figure 4: Dynamic Workflow Optimization Process

287
288
289 modules as immutable blocks, DSRs will parse each `Module` into an executable Directed Acyclic
290 Graph (DAG) representation that explicitly captures predictor dependencies, data transformations,
291 and control flow.
292

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

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

302 2 DATA AND EVALUATION

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

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

313 3 PROGRESS REPORT

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

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

324 4 ACKNOWLEDGEMENT
325326 This project was built with the support of:
327

- 328 • **Maguire Papay**, Founder of HokyoAI: provided guidance and feedback on performance
329 optimization strategies and abstraction design.
- 330 • **Prashanth Rao**, AI Engineer at LanceDB: helped with explanation review, and battle test-
331 ing the framework.
- 332 • **Josh**, Core Maintainer of Rig: provided guidance in integrating the Rig framework with
333 DSRs.
334

335 REFERENCES
336337 Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2025. URL <https://arxiv.org/abs/2408.08435>.
338339 Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vard-
340 hamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei
341 Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-
342 improving pipelines, 2023. URL <https://arxiv.org/abs/2310.03714>.
343344 Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov,
345 and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question
346 answering, 2018. URL <https://arxiv.org/abs/1809.09600>.
347348 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao.
349 React: Synergizing reasoning and acting in language models, 2023. URL <https://arxiv.org/abs/2210.03629>.
350351 Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-
352 ended evolution of self-improving agents, 2025. URL <https://arxiv.org/abs/2505.22954>.
353354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377