**Rust and LLM AI Infrastructure: Embracing the Power of Performance**

[[](https://jondot.medium.com/?source=post_page-----c72bb705a96c--------------------------------)](https://jondot.medium.com/?source=post_page-----c72bb705a96c--------------------------------)

[[Better Programming](https://betterprogramming.pub/?source=post_page-----c72bb705a96c--------------------------------)](https://betterprogramming.pub/?source=post_page-----c72bb705a96c--------------------------------)

[Dotan Nahum](https://jondot.medium.com/?source=post_page-----c72bb705a96c--------------------------------)

·

Follow

Published in

[Better Programming](https://betterprogramming.pub/?source=post_page-----c72bb705a96c--------------------------------)

·

9 min read

·

Aug 2

203

3



Image by author

Building an LLM or AI infrastructure in Rust can offer several benefits despite Python’s dominance in the AI space.

1. Performance: Rust is known for its high performance and low-level control, which can be crucial for building large-scale AI systems. Language models, especially deep learning models, can be computationally intensive. Rust’s performance can lead to significant speed improvements compared to Python, making it more suitable for efficiently handling computationally expensive tasks.
2. Memory Safety: Rust’s strict compiler rules and ownership model ensure memory safety, preventing common bugs like null pointer dereferences and data races. This can make Rust-based AI systems more reliable and less prone to crashes, which is particularly important for long-running language models or critical AI applications.
3. Concurrency: Rust’s built-in support for concurrency and lightweight threads can lead to the efficient utilization of multicore processors. This can be valuable when implementing parallel processing for training large language models or handling multiple inference requests simultaneously.

Yes, it is worth remembering that Python’s extensive ecosystem, well-established libraries (e.g., TensorFlow, PyTorch), and ease of use have made it the go-to choice for many AI projects. But if you have a sense of exploration and pinned out your specific requirements and Rust is there at the top, this might be a relevant read for you.

Let’s try to answer the question: does Rust have a go-to all-inclusive library like Python has [langchain](https://langchain-langchain.vercel.app/) for working with LLMs?

*P.S. It might be worthwhile to exercise some of the building blocks behind langchain in Python. For that, I created*[*langchain-llm-katas*](https://github.com/jondot/langchain-llm-katas)*, give it a try.*

**Chain Library Building Blocks**

If we zoom out for a minute and ignore high-level workflows such as agents, toolkits, and baked use cases, we’ll be looking at the common infrastructure beneath all of those which allow for these high-level constructs to exist:

* LLM — loading, calling LLMs, and supporting different models, which need:
* Embedding and Tokenization — turning text into embeddings, where text is loaded with:
* Loaders — turning various document formats into LLM digestible, flat text, which needs to be post-processed by:
* Splitters — to create legible, useful chunks of the original text, to work with LLM token limits and be stored in:
* Vector databases — which are used to formulate prompts that are sent to LLMs and also for powering:
* Memory — which is done to support context and sessions with LLMs. But also, we have the infrastructure for:
* Templates — that structure a request from an LLM, which may contain a call for:
* Tools — which are a collection of real-world tools such as a calculator or a browser for the LLM to automate by reading a structured prompt

So what does all of that look like currently in Rust?

**Infrastructure and building blocks**

**LLM and Transformers**

[**llm**](https://github.com/rustformers/llm)

llm is an ecosystem of Rust libraries for working with large language models. It's built on top of the fast, efficient [GGML](https://github.com/rustformers/llm/blob/main/crates/ggml) library for machine learning. It is a solid, expansive, and stable library for working with models, to the point that this is the one library to always pick, and it’s a good thing that there aren’t many alternatives.

Here’s how to run inference using just the llm crate:

let model = llm::load\_dynamic(  
 Some(model\_architecture),  
 &model\_path,  
 tokenizer\_source,  
 Default::default(),  
 llm::load\_progress\_callback\_stdout,  
)  
.unwrap\_or\_else(|err| {  
 panic!("Failed to load {model\_architecture} model from {model\_path:?}: {err}")  
});  
let mut session = model.start\_session(Default::default());  
let res = session.infer::<Infallible>(  
 model.as\_ref(),  
 &mut rand::thread\_rng(),  
 &llm::InferenceRequest {  
 prompt: prompt.into(),  
 parameters: &llm::InferenceParameters::default(),  
 play\_back\_previous\_tokens: false,  
 maximum\_token\_count: None,  
 },  
 // OutputRequest  
 &mut Default::default(),  
 |r| match r {  
 llm::InferenceResponse::PromptToken(t) | llm::InferenceResponse::InferredToken(t) => {  
 print!("{t}");  
 std::io::stdout().flush().unwrap();  
 Ok(llm::InferenceFeedback::Continue)  
 }  
 \_ => Ok(llm::InferenceFeedback::Continue),  
 },  
);

[**rust-bert**](https://github.com/guillaume-be/rust-bert)

Probably the most versatile and productive library out there for rust for using transformer models, inspired by huggingfaces transformers library. This is a one-stop shop for local transformer models for different tasks, from translation to embedding.

* Lots of use cases and examples
* Very wide, as is the original transformers library
* You can get a lot done just by using this library, and you might not need anything else

In terms of use cases, here’s an example sentiment analysis:

let sentiment\_classifier = SentimentModel::new(Default::default())?;  
   
let input = [  
 "Probably my all-time favorite movie, a story of selflessness, sacrifice and dedication to a noble cause, but it's not preachy or boring.",  
 "This film tried to be too many things all at once: stinging political satire, Hollywood blockbuster, sappy romantic comedy, family values promo...",  
 "If you like original gut wrenching laughter you will like this movie. If you are young or old then you will love this movie, hell even my mom liked it.",  
];  
let output = sentiment\_classifier.predict(&input);

And embeddings:

let model = SentenceEmbeddingsBuilder::remote(  
 SentenceEmbeddingsModelType::AllMiniLmL12V2  
 ).create\_model()?;  
let sentences = [  
 "this is an example sentence",  
 "each sentence is converted"  
];  
let output = model.encode(&sentences)?;

And plenty more use cases.

**Embeddings and Tokenization**

You can do embeddings with the following:

* rust-bert
* llm

[**tiktoken**](https://github.com/zurawiki/tiktoken-rs)

This library is built on top of the OpenAI Rust tiktoken library and extends it a bit. It should be a good one-stop shop for your tokenization needs.

use tiktoken\_rs::p50k\_base;  
let bpe = p50k\_base().unwrap();  
let tokens = bpe.encode\_with\_special\_tokens(  
 "This is a sentence with spaces"  
);  
println!("Token count: {}", tokens.len());

**Loaders**

At the moment of writing, there is no unified loader like unstructured, that can load and convert documents without caring so much for the implementation of the specific provider. Some of these may need some elbow grease to get content into a flat, LLM-document-like format (plain text, pages), for example, looping over and extracting worksheets from Excel files.

However, here’s a sensible mapping from file format to an appropriate Rust library:

* [csv](https://lib.rs/crates/csv)
* email: [eml, msg](https://crates.io/crates/eml-parser)
* epub [pandoc](https://lib.rs/crates/pandoc)
* [xls](https://lib.rs/crates/calamine)
* [html](https://lib.rs/crates/html2text) [pandoc](https://lib.rs/crates/pandoc)
* [images (ocr)](https://github.com/antimatter15/tesseract-rs)
* [md](https://lib.rs/crates/pulldown-cmark%20pandoc)
* [org mode (.org)](https://github.com/hydrobeam/org-rust)
* [open office](https://lib.rs/crates/spreadsheet-ods) [pandoc](https://lib.rs/crates/pandoc)
* [pdf](https://lib.rs/crates/pdf-extract)
* txt (no need)
* [pptx](https://github.com/anvie/dotext) [pandoc](https://lib.rs/crates/pandoc)
* [docx](https://github.com/anvie/dotext) [pandoc](https://lib.rs/crates/pandoc)
* [rst](https://crates.io/crates/rst) [pandoc](https://lib.rs/crates/pandoc)
* rtf [pandoc](https://lib.rs/crates/pandoc)
* [xml](https://crates.io/crates/xml-rs)

**Splitters**

<https://github.com/benbrandt/text-splitter>

The only library that is practical enough for splitting. Seems to be taking a “split properly” approach where you don’t need to choose if to split by newlines, characters, recursively, or tokens. It will descend and use an appropriate method to maximize chunk sizes.

**Prompts**

At the moment of writing, there’s no general consensus and no library that unifies the template concept like langchain has. However, taking a templating library and building on top of it is a great way to implement prompts.

Rust has a few fantastic templating libraries which suit templating:

* [Handlebars](https://lib.rs/crates/handlebars) — for standard, familiar, minimal logic templates
* [Tera](https://docs.rs/tera/latest/tera/) — for those familiar with jinja2
* [Liquid](https://lib.rs/crates/liquid) — for those familiar with liquid

Most times, handlebars is the best bet that’ll be familiar to a general audience.

**Vector Databases**

Currently, there’s no unified interface for vector databases that provides a general wrapper around many providers. Survey a few open source projects around LLMs. You’ll find reimplementations of the same interfaces over and over, which is a pity, and still, no one library provides a generic interface.

Given such a simplistic interface, there is a good ROI in building such a library. It should basically be:

* add\_documents
* similarity\_search

FYI: Many individual vector databases are implemented in Rust, both commercial and open source. [Qdrant](https://github.com/qdrant/qdrant) is probably the most popular open source one.

**Memory**

There are a few projects implementing memory or history. Most of them implement a service that stores indexes and search histories for you, and they can also be used as a reference for how to do your workflow in Rust dealing with AI.

Some interesting ones are:

* [memex](https://github.com/spyglass-search/memex) — part of [spyglass](https://github.com/spyglass-search), a good code read, and does document storage and semantic search for LLM projects.
* [indexify](https://github.com/diptanu/indexify) — long-term LLM memory
* [motorhead](https://github.com/getmetal/motorhead) — memory/information retrieval

**Tools**

As of this writing, no tool/plugin libraries are dedicated to this purpose only. That is, libraries that provide a solid interface for:

* Declaring a tool
* Capabilities
* Running a tool (safely or not)
* Structuring a tool request and response

However, those implementations can be found in the few chain implementations that Rust currently has, which are discussed below.

**Chains**

[**llmchain-rs**](https://github.com/shafishlabs/llmchain-rs)

A small-scope chain library. I’d say 30% of what langchain-py has.

* Early times for this project
* Good amount of examples
* [Databend](https://github.com/datafuselabs/databend)-focused, but generic interfaces
* Good test coverage (it’s a challenge testing LLM infra in any case)

Let’s look at some of the abstractions:

**Embedding**

pub trait Embedding: Send + Sync {  
 async fn embed\_query(&self, input: &str) -> Result<Vec<f32>>;  
 async fn embed\_documents(&self, inputs: &Documents) -> Result<Vec<Vec<f32>>>;  
}

**LLM**

pub trait LLM: Send + Sync {  
 async fn embedding(&self, inputs: Vec<String>) -> Result<EmbeddingResult>;  
 async fn generate(&self, input: &str) -> Result<GenerateResult>;  
 async fn chat(&self, \_input: Vec<String>) -> Result<Vec<ChatResult>> {  
 unimplemented!("")  
 }  
}

* Integrations currently are: OpenAI, Azure-OpenAI, and Databend, so probably mixing data from Databend with AI was the trigger for all of this

**Document**

#[derive(Debug, Clone, Eq, PartialEq)]  
pub struct Document {  
 pub path: String,  
 pub content: String,  
 pub content\_md5: String,  
}  
impl Document {  
 pub fn create(path: &str, content: &str) -> Self {  
 Document {  
 path: path.to\_string(),  
 content: content.to\_string(),  
 content\_md5: format!("{:x}", md5::compute(content)),  
 }  
 }  
 pub fn tokens(&self) -> usize {  
 chat\_tokens(&self.content).unwrap().len()  
 }  
 pub fn size(&self) -> usize {  
 self.content.len()  
 }  
}  
#[derive(Debug)]  
pub struct Documents {  
 documents: RwLock<Vec<Document>>,  
}

**Prompt**

pub trait Prompt: Send + Sync {  
 fn template(&self) -> String;  
 fn variables(&self) -> Vec<String>;  
 fn format(&self, input\_variables: HashMap<&str, &str>) -> Result<String>;  
}

* Simple text-based replacement of variables only
* Single level, no advanced dependent or partial prompts

**Vector store**

#[async\_trait::async\_trait]  
pub trait VectorStore: Send + Sync {  
 async fn init(&self) -> Result<()>;  
 async fn add\_documents(&self, inputs: &Documents) -> Result<Vec<String>>;  
 async fn similarity\_search(&self, query: &str, k: usize) -> Result<Vec<Document>>;  
}

* Support for databend as a provider only
* Missing MMR for search (like in all interfaces I’ve seen so far)
* Missing save/load abstraction (but does it even belong in a trait?)

**Overall**

* Relatively solid codebase and good abstractions
* Splitters (no abstraction or strategy to choose from, but this is the case with all chain implementations in Rust I’ve seen so far)
* Memory (just early times there, it feels)

[**llm-chain**](https://github.com/sobelio/llm-chain)

A more popular chain library, split into Rust crates, good Rust idioms, and structure. Not a full coverage of what langchain-py has. Maybe 30%, and still early times here, as with all other libraries.

Let’s look at the abstractions here:

**Embedding**

#[async\_trait]  
pub trait Embeddings {  
 type Error: Send + Debug + Error + EmbeddingsError;  
 async fn embed\_texts(&self, texts: Vec<String>) -> Result<Vec<Vec<f32>>, Self::Error>;  
 async fn embed\_query(&self, query: String) -> Result<Vec<f32>, Self::Error>;  
}

**VectorStore**

#[async\_trait]  
pub trait VectorStore<E, M = EmptyMetadata>  
where  
 E: Embeddings,  
 M: serde::Serialize + serde::de::DeserializeOwned,  
{  
 type Error: Debug + Error + VectorStoreError;  
 async fn add\_texts(&self, texts: Vec<String>) -> Result<Vec<String>, Self::Error>;  
 async fn add\_documents(&self, documents: Vec<Document<M>>) -> Result<Vec<String>, Self::Error>;  
 async fn similarity\_search(  
 &self,  
 query: String,  
 limit: u32,  
 ) -> Result<Vec<Document<M>>, Self::Error>;  
}

**Splitters**

Looks like a splitter is a Tokenizer

pub trait Tokenizer {  
 fn tokenize\_str(&self, doc: &str) -> Result<TokenCollection, TokenizerError>;  
 fn to\_string(&self, tokens: TokenCollection) -> Result<String, TokenizerError>;  
 fn split\_text(  
 &self,  
 doc: &str,  
 max\_tokens\_per\_chunk: usize,  
 chunk\_overlap: usize,  
 ) -> Result<Vec<String>, TokenizerError>;  
}

**Prompt**

[Prompt](https://github.com/sobelio/llm-chain/blob/main/crates/llm-chain/src/prompt/model.rs) is a construct that allows operating with an abstraction of Chat or Text messages. Templating is powerful and based on tera.

**Document**

#[derive(Debug)]  
pub struct Document<M = EmptyMetadata>  
where  
 M: serde::Serialize + serde::de::DeserializeOwned,  
{  
 pub page\_content: String,  
 pub metadata: Option<M>,  
}

More similar than different in comparison to the langchain-py document.

**DocumentStore (Loader?)**

#[async\_trait]  
pub trait DocumentStore<T, M>  
where  
 T: Send + Sync,  
 M: Serialize + DeserializeOwned + Send + Sync,  
{  
 type Error: std::fmt::Debug + std::error::Error + DocumentStoreError;  
async fn get(&self, id: &T) -> Result<Option<Document<M>>, Self::Error>;  
 async fn next\_id(&self) -> Result<T, Self::Error>;  
 async fn insert(&mut self, documents: &HashMap<T, Document<M>>) -> Result<(), Self::Error>;  
}

**Overall**

* Good separation between features and crates
* Feels rusty
* Lacking splitters, loaders
* Lacking integrations generally, and especially vector stores

[**SmartGPT**](https://github.com/Cormanz/smartgpt)

This is not a library nor infrastructure, but the internals are built well and can be of good use for a chain library. Let’s have a look at the abstractions:

**LLM**

pub trait LLMModel : Send + Sync {  
 async fn get\_response(&self, messages: &[Message], max\_tokens: Option<u16>, temperature: Option<f32>) -> Result<String, Box<dyn Error>>;  
 async fn get\_base\_embed(&self, text: &str) -> Result<Vec<f32>, Box<dyn Error>>;  
 fn get\_token\_count(&self, text: &[Message]) -> Result<usize, Box<dyn Error>>;  
 fn get\_token\_limit(&self) -> usize;  
 fn get\_tokens\_from\_text(&self, text: &str) -> Result<Vec<String>, Box<dyn Error>>;  
}

Some methods were removed because they were default implemented.

**Message**

#[derive(Clone, Debug)]  
pub enum Message {  
 User(String),  
 Assistant(String),  
 System(String)  
}

Modeling a message on a User-AI-System pattern

**MemorySystem (History?)**

pub trait MemorySystem : Send + Sync {  
 async fn store\_memory(&mut self, llm: &LLM, memory: &str) -> Result<(), Box<dyn Error>>;  
async fn get\_memory\_pool(&mut self, llm: &LLM, memory: &str, min\_count: usize) -> Result<Vec<RelevantMemory>, Box<dyn Error>>;  
 async fn get\_memories(  
 }  
...

A good interface of memory

**Command (Tool?)**

pub trait CommandImpl : Send + Sync {  
 async fn invoke(&self, ctx: &mut CommandContext, args: ScriptValue) -> Result<CommandResult, Box<dyn Error>>;  
fn box\_clone(&self) -> Box<dyn CommandImpl>;  
}

**Overall**

* This project has a relatively rich tool collection and might be the best start for a generic library that implements tools
* Codebase is very pragmatic because almost everything is actually in use
* Where functionality was not needed — it does not exist (naturally), and this is where the codebase lacks, in comparison to langchain: splitters, loaders, prompt templates, vector stores, and a general sense of multiple providers
* To use it as a general-purpose langchain library would be awkward unless you hit the use cases that this project solves already. Which I think will not be enough pretty quickly.

Thanks for reading.