Hi there! Welcome to St. Jude's Google Summer of Code 2025 homepage 🎉! With any questions, feel free to reach out to us on Slack in the #gsoc-2025 channel if the question is of general interest, through direct message if the question is confidential, or by email if you can't use Slack (clay.mcleod@stjude.org and stephanie.sandor@stjude.org)!

If you do apply, please be sure to use our Application template!

Table of contents

Background

Our projects are centered around changing how genomics analysis is done at scale. In an effort to advance cures for pediatric catastrophic diseases as quickly as possible, St. Jude has been sharing critical research data with the biomedical community for about 7 years now (see the St. Jude Cloud paper for more details). This endeavour has led to a great deal of experience in the challenges of sharing this data with the broader academic community. To address those issues, we recently started our own workflow execution engine written in Rust called **Sprocket** aimed at enabling genomic analyses at petabyte-scale—though we're focused on genomics, we think that Sprocket can be used for nearly all scientific domains.

Sprocket is built on top of the open Workflow Description Language (github.com/openwdl), of which we have multiple people on the governance committee, and alongside the execution engine we're building, there are also many language design problems we're tackling such as building our own lexer, parser, abstract syntax tree, concrete syntax tree, linter, validation tools, language server protocol, VSCode plugin, formatter, and documentation generator. To that end, all of our projects are geared around improving either the Sprocket execution engine itself (better for those interested in cloud or genomics) or the developer ecosystem around WDL (better for those interested in programming language design and developer experience improvements). You can find out more by navigating to the links below.

- **https://github.com/stjude-rust-labs/wdl**. The core crates for the project. The `wdl` crates contain functionality covering the lexing, parsing, abstract syntax tree, concrete syntax tree, linting, validation, language server protocol, formatting, and documentation

generation. This is the core set of libraries upon which Sprocket is created, but it is not generally considered “user-facing”.

- **https://github.com/stjude-rust-labs/crankshaft.** A *headless* workflow execution workflow execution engine. If the `wdl` crates above contain the WDL-specific parts, the `crankshaft` crate(s) contain the language agnostic plumbing code for submitting jobs to backend execution platforms (local, Docker, Cloud, HPC, etc).
- **https://github.com/stjude-rust-labs/sprocket.** The user-facing command line tool that brings all of the above functionality together. Notably at the time of writing, much of the functionality that has been developed in `wdl` and `crankshaft` has not yet been surfaced in `sprocket`—that’s where you come in 🙂.
- **https://github.com/stjude-rust-labs/sprocket-vscode.** The extension for Sprocket in VSCode. You can also see the extension published here.
- **https://github.com/openwdl/wdl.** Some projects may require interacting with or making suggestions to the Workflow Description Language. The linked repository here includes the entire specification that you can browse to get a sense of how it works.
- **https://github.com/stjude-rust-labs**. The organization as a whole. We encourage you to check out the other projects we’re working on here—particularly as it relates to other projects you might want to propose.

Projects

Extend Sprocket language server protocol and extension.

Length: 350 hours, Difficulty: Medium to hard (stretch)

Description

Our developer tools and experience project focuses largely on Sprocket’s LSP integration within VSCode and the VSCode extension itself (linked above). During this track, you’ll utilize the existing `wdl-lsp` packages in the `wdl` crates repository linked above to expose things like lookup definition, go to definition, references, snippets, and semantic code highlighting. For a stretch goal, we would like to continue optimizing and refining the internal workings of the LSP (and the associated AST/CST). Along the way, you’ll learn about fundamentals of writing a programming language from scratch, how and LSP implementation works, Rust itself, and you’ll contribute to Workflow Description Language developers everywhere being more productive because of *your* efforts.

Outcomes

- Lookup definition, go to definition, references, snippets, and semantic code highlighting integrated within the Sprocket VSCode extension (linked above) ***(Project)***.
- Improved latency, memory use, cpu use, and reliability for the LSP engine ***(Project, stretch goal)***.

- Seamless WDL development experience for workflow authors ***(End users)***.
- A deeper understanding of programming language tooling and Rust ***(Contributor)*** .

Relevant Skills

Fluency with Rust or another equivalent high performance language (C, C++, Go) with a desire to learn Rust is a must. Experience building developer tools and extensions is slightly preferred.

Potential Mentors

Clay McLeod, Director of Product Development and Engineering

Proposal Advice

First, we recommend following Visual Studio Code's guide on building your first extension (link). This will give you a sense of what it's like to build a Visual Studio Code extension without too much time investment. Next, we recommend reading and understanding at a high-level what the Language Server Protocol (LSP) is (link). The protocol is incredibly detailed with many corners so to speak—you don't need to understand everything. It's just important to understand generally how it works and what problems it solves. Last, we recommend downloading and trying to load up the Sprocket VSCode extension (link)—the instructions to do so are in the project `README.md` (link). All of the information you end up learning from this exercise should be included in the final application.

Real-time monitoring dashboard and terminal user interface (TUI) for Crankshaft.

Length: 175 hours to 350 hours, Difficulty: Medium to Hard

Description

Running petabyte-scale genomics analyses is only tenable when you know what's actually going on with your workflows. In this track, you'll work to develop real-time monitoring tools for Crankshaft (that eventually will surface in Sprocket) to ensure that end users know what's going on with thousands of their workflows. Interactions with jobs, such as killing jobs or interrogating their logs, is a stretch goal but not required. Along the way, you'll learn about cloud analysis at scale, how a workflow execution engine is built, how to develop a terminal-user interface, and Rust itself.

Outcomes

- A clear, easy to use and interpret terminal user interface (very similar to `tokio-rs/console`) that allows for interpretation, monitoring, and some lightweight

interaction with jobs. The extent of the interactions with the jobs depends on how far we get with the initial monitoring and displaying of statuses ***(Project)***.
- Hands on experience with building the inner workings of a petabyte-scale engine using Rust. This includes building expertise in the Rust programming language and tokio-related crates ***(Contributor)***.

Relevant Skills

Fluency with Rust or another equivalent high performance language (C, C++, Go) with a desire to learn Rust is a must.

Potential Mentors

Clay McLeod, Director of Product Development and Engineering

Proposal Advice

We'd highly recommend you go and explore what has been created for `tokio-console` (link) as a starting point. This will give you a good sense of a similar project that aims to describe the inner workings of a complex system using a TUI. Further, you could check out the `ratatui` crate (link), as that's a leading library for creating such TUIs (though, if you have a different library you want to use, that's totally okay too). Last, if you're feeling up to it, it would be great to write a small program using a TUI just to make sure you understand generally how they work. All of the information you end up learning from this exercise should be included in the final application.

Integrated end-to-end workflow testing framework within Sprocket.

Length: 350 hours, Difficulty: Medium to Hard (stretch)

Description

Within Sprocket, we'd like to build out a first-class testing framework that workflow authors can depend on to end-to-end test their workflow throughout development. This subcommand will be designed with continuous integration and deployment in mind, and will use the St. Jude Cloud Workflows repository as a baseline for implementing this scheme. As a stretch goal, generating fake data for genomics workflows themselves might be compelling to start on (check out the `ngs generate` command (link) to get an idea of what this might look like. Along the way, you'll learn a bit about genomics, how workflows are written and deployed at scale, how to robustly test scientific workflows, and Rust itself.

Outcomes

- A end-to-end testing framework for reproducible workflows using Sprocket that can be integrated into the CI/CD workflows of tool developers using WDL ***(Project, end users)***.
- *(Stretch Goal)* New fake data generation tools, geared towards generating fake genomics data, for use within these testing workflows ***(Project, end users)***.
- A better understanding of Rust, genomics software engineering, and what it takes to scale the development of genomics workflow analyses ***(Contributor)***.

Relevant Skills

Fluency with Rust or another equivalent high performance language (C, C++, Go) with a desire to learn Rust is a must. Fluency in genomics is not required, but you must be willing to spend a significant amount of time learning if you don't already understand it (ergo, you must want to learn genomics).

Potential Mentors

Clay McLeod, Director of Product Development and Engineering

Proposal Advice

First, we recommend reading through and understanding the general idea behind the Workflow Description Language standard (link). This will give you a good idea of how workflows are expressed in Sprocket. Next, we recommend you take a look at St. Jude's repository for workflows (link) to get a sense of what a real-life repository of these workflows where testing should occur looks like. You can take a look at the `tools` folder to see the individual WDL tasks and the `workflows` folder see the WDL workflows. The `rnaseq-standard` pipeline is a particularly good example (link). Last, we recommend looking at the state of the art approach today for doing this, which is a project called `pytest-workflow` (link). There are several things we think this library got right alongside several things we would change, so just take the inspiration as a grain of salt, *not* as a template of something we want to redo in Rust. All of the information you end up learning from this exercise should be included in the final application.

We're open to other, relevant ideas.

- We're also open to other ideas! Feel free to check out the St. Jude Rust Labs organization to see what we're working on. In particular, ideas using the `omics` library (https://github.com/stjude-rust-labs/omics) and the `chainfile` library (https://github.com/stjude-rust-labs/chainfile) would be really cool to see. Last, if you *do* plan to submit your own idea, please discuss them with us briefly during the discussion period (February 27-March 24) before submitting an application—we don't want you to do a lot of work for an idea that's unlikely to be selected!

Application template

Please use this application template when submitting your proposal.

1. **Project Title.** Give a single sentence title to your project. You can use the existing project titles above or tweak them based on your own interests/ideas.
2. **Abstract (~4-6 sentences).** Describe the big picture of what you plan to accomplish during this project. You can feel free to reuse ideas or language that we used above—just make sure you give a sense of what you want to accomplish underneath those broad headings.
3. **Project Description.** An unbounded project description. Feel free to talk through the details of how you prepared for submitting this application (did you follow any of our advice for each idea? How did that shape the way you think about the idea, etc) and any more detailed plans for how you plan to go about the project.
4. **Major contributions.** A list of goals that you hope to accomplish by the end of the program. Use a bulleted list here, and be sure to include both (a) what the outcome is going to be and (b) the target audience of that outcome.
5. **Timeline.** Break down the work to achieve these major contributions into a timeboxed plan as it aligns to the GSoC timeline (link). We've provided a template here to get started.

Period	**GSoC Phase**	**What I'll aim to accomplish**
May 8 - June 1	Community Bonding Period	*GSoC contributors get to know mentors, read documentation, get up to speed to begin working on their projects*
June 2 - June 8	Coding (Week 1)	
June 9 - June 15	Coding (Week 2)	
June 16 - June 22	Coding (Week 3)	
June 23 - June 29	Coding (Week 4)	
June 30 - July 5	Coding (Week 5)	
July 6 - July 12	Coding (Week 6)	
July 13 - July 19	Coding (Week 7)	*Midterm evaluation is July 18*
July 20 - July 26	Coding (Week 8)	
July 27 - August 3	Coding (Week 9)	
August 4 - August 10	Coding (Week 10)	*Your plan should start to include thinking about documenting the*

		final product around this point.
August 11 - August 17	Coding (Week 11)	
August 18 - August 24	Coding (Week 12)	
August 25 - September 1	Coding (Final week)	

6. **Time Period/Working Hours.** Let us know how much time you plan to spend on the project. Be sure to indicate both (a) how many hours you think the project will take (we're not looking for a complicated hourly breakdown—just if it's 175 hours or 350 hours as defined by the GSoC guidelines) and (b) roughly how many hours per week you plan to spend on the project.
7. **Name and GitHub username.** Please share with us your name and the GitHub username with which you are going to do the work.
8. **Resume.** Please include a recent copy of a resume/CV.
9. **Code review.** Please provide us with a link to a code sample that you feel particularly proud of and that demonstrates your abilities. This can be a PR that you made to an existing codebase *or* a brand new project that you created and worked on. The key is that we want the code to be completely yours (or, at least, it needs to be clear which parts you specifically did, like in the case of a PR you created).