GSoC 2026 — Victor Yanson

This document forms part of an application for Google's Summer of Code 2026. It encompasses the applicant's personal information and experience as well as a detailed project proposal. The project stems from a mentor project idea put forth by Simon Poole and can thus be found on the official OSM GSoC 2026 project idea page under the 'Routing' category.

General information

Personal information

Full name	**Victor Alexander Marek Yanson**
Occupation	Freelance Developer
OSM account	VictorYanson
GitHub account	VictorYanson
Email	victor.yanson@outlook.com
LinkedIn	linkedin.com/in/victoryanson

Bio

Hi! My name is **Victor**. I'm a **23 year old freelance web developer** from the **Netherlands**. About a year ago I completed my **Bachelor's in Business Administration** after which I pivoted into freelance web dev. Working on various projects, I've been progressively working my way down the stack, having developed a recent interest in performance computing. Currently living between Nijmegen in the Netherlands and Barcelona, Spain, where my partner is from.

Relevant experience

Background

In total I've been programming for about **two years** now. Not coming from a technical background, I taught myself to program making small pet-projects. Since starting to code professionally, I've had a decent amount of exposure to FOSS tools although never having actually contributed myself until now. My early freelance work mainly consisted of CMS maintenance for non-technical clients. After a few projects however, I transitioned into writing full-stack web-based MVPs for aspiring entrepreneurs. Feel free to take a look at my freelance page built with Nuxt.js (although at this point it's a bit outdated).

Languages and frameworks

My day-to-day language is **TypeScript**, although I'm very comfortable with **Python** as well. In addition, I've been familiarizing myself with **C++** and related low-level concepts. For most of my professional work I use frontend JavaScript libraries like **React** and **Vue** alongside their respective metaframeworks, **Next.js** and **Nuxt.js**. My full-stack projects have also allowed me to work with **SQL databases** in a professional setting, the main dialects being **Postgres** and **MySQL**. Lastly, my work has allowed me to get comfortable creating and deploying **Docker** containers on the **cloud**.

Projects

A big part of my professional projects have involved creating AI automation tools and chatbots alongside setting up and maintaining e-commerce pages. While these are my "bread-and-butter" projects, I see them as a consistent foundation for more specialized work later on.

A fundamental project for me has been the creation of a GIS-based app called **Focal Grid**. I was essentially put in charge of making the first version of a full-stack web app built to visualize Dutch public energy network data. This project introduced me to lots of core GIS concepts like **PostGIS**, **vector tiles**, and **GeoJSON**. Moreover, it sparked a genuine interest in GIS technology in me. This

project took place during the end of last year, which was also the moment I discovered GSoC. I have gained permission from the client to share some details about the project which you can find here.

Another notable personal project of mine was a basic **driver behavior simulator** written in C++. The main goal of the project was to recreate realistic acceleration and braking behavior in a multi-car single lane situation. The driving behavior is defined by the **Intelligent Driver Model** and for the visualization I used **RayLib**. Feel free to check out the project here.

Community involvement

Firstly, I'd like to be frank about the fact that this will be **the only project**, and therefore OSM the only org, I'm planning on applying to this year. Throughout the last few months I've built up a comfortable amount of context and passion surrounding the issue at hand. I've had the chance to make contact with many of the project's implied parties, and I honestly wouldn't have the same confidence applying to another project.

That being said, I would describe my entry into the community as a pleasant learning experience. My very first contact with OpenStreetMap was through the use of the **JavaScript Mapbox Graphics Library** for Focal Grid, although this was admittedly more of a passive interaction. The first active contact with the community started once I had decided on OSM as my target org in January. Throughout the research of the project I have become more aware of maintainer expectations and open-source etiquette as a whole. I'm happy to say that I've received thoughtful guidance from **Simon Poole** and **Archit Rathod** on the context surrounding closures.osm.ch. Moreover, I've gotten to engage with the **Valhalla project** as well as the maintainers behind it (see contributions below). Lastly, I had a great exchange with **Eggie**, one of the main map editors and moderators for the Dutch OSM community.

Contributions

Valhalla

- **PR #5884:** Added Valhalla version logging to `docker-entrypoint` on startup. (**Merged**)
- **Issue #5757/#5758:** Technical follow-up on vector tile overzoom and live traffic layer implementation.

Closures.osm.ch

- **PR #21:** Fixed MacOS Docker build compatibility and startup crashes.
- **PR #22:** Integrated `vitest` and added unit tests for Valhalla-related frontend logic. (**Merged**)
- **PR #35:** Added poetry.lock for backend build reliability.

- **Issue #25/#27:** Identified redundant `valhallaApi` code and gaps in query parameter documentation.

Ecosystem (triage & maintenance)

- **Graphhopper (#3310, #3309):** PR linking and issue closure maintenance.
- **OSRM (#7138):** Metadata/Labeling maintenance

Map data

Barcelona

- Changeset: 179776337
- Changeset: 179257968
- Changeset: 179135609

Nijmegen

- Changeset: 178238893
- Changeset: 178237957
- Changeset: 178237030

Availability

Without getting into too much detail, my personal situation allows me to focus this summer fully on professional and personal development ahead of my enrollment in my Master's course. This means that I'm assured to have **at least 30 hours a week** available during the summer months to commit to GSoC. This would be in line with the **350 hour time scope** proposed by Simon. The only other summer activity of note is my self-study of math-related preparatory university materials.

Additionally, I'm currently in the midst of a client project. However, the project is foreseen to finalize mid-April, after which I don't have any other major projects lined up.

Lastly, I'm honestly not quite sure yet what vacation plans will arise come summertime. Nevertheless, if any trips come up, I can assure they won't be more than a **weekend getaway**. Of course, I will be sure to keep my mentor up to date should anything come up.

Project proposal

High-level summary

`closure-sync` is a **containerized sidecar service** designed to bridge the gap between **real-time road closures** on `closures.osm.ch` and **dynamic routing engines**. While meant for **engine-agnostic** extensibility, the initial implementation focuses on **Valhalla**. Here it provides a **pipeline** to translate **OpenLR** closure data into live Valhalla **traffic tiles**. The service emphasizes "drop-in" usability, allowing map providers to synchronize live road closures with **minimal configuration**.

Context

What is closures.osm.ch?

`closures.osm.ch` is an **OSM platform** that collects and shares community-sourced **real-time road closure data**. It uses a **FastAPI** in combination with **PostGIS** as its backend and additionally offers a **Next.js** frontend for manual data manipulation and **closure-aware routing**. The project was created during **GSoC 2025** by Archit Rathod and is currently still in **beta**.

Proposed project idea

The project idea proposed by Simon Poole boils down to making closures.osm.ch **production-ready** by integrating it with **at least one routing engine**, enabling direct use of closure data in routing calculations.

Problem

Current routing approach

As of now, `closures.osm.ch` handles closure-aware routing almost entirely on the client. The current mechanism fetches closure data from the backend to the Next app, passing the fetched closures as `exclude_locations` parameters in routing requests to the Valhalla server.

Areas of concern

While the aforementioned routing mechanism was meant as a functional proof of concept, it presents several important limitations:

1. **Inefficient request flow:** Closure aware routing currently involves **fetching** closure data from the backend, **processing** it on the client, and **injecting** it into the routing request. Running this flow

on every request introduces latency and unnecessary overhead.
2. **Poor separation of concerns:** Having the client take charge of fetching and normalizing closures means that every application using `closures.osm.ch` will need to reimplement its own version of closure-aware routing.
3. **API limits:** The current implementation limits routing queries to 50 points. This limits usability to small **bbox** areas with few closure coordinates.
4. **Routing accuracy:** The `exclude_locations` parameter maps geometries to their closest graph edge in Valhalla during runtime. Ideally closures will get mapped directly to their corresponding edge IDs **before** the routing request happens.

Solution

Considerations

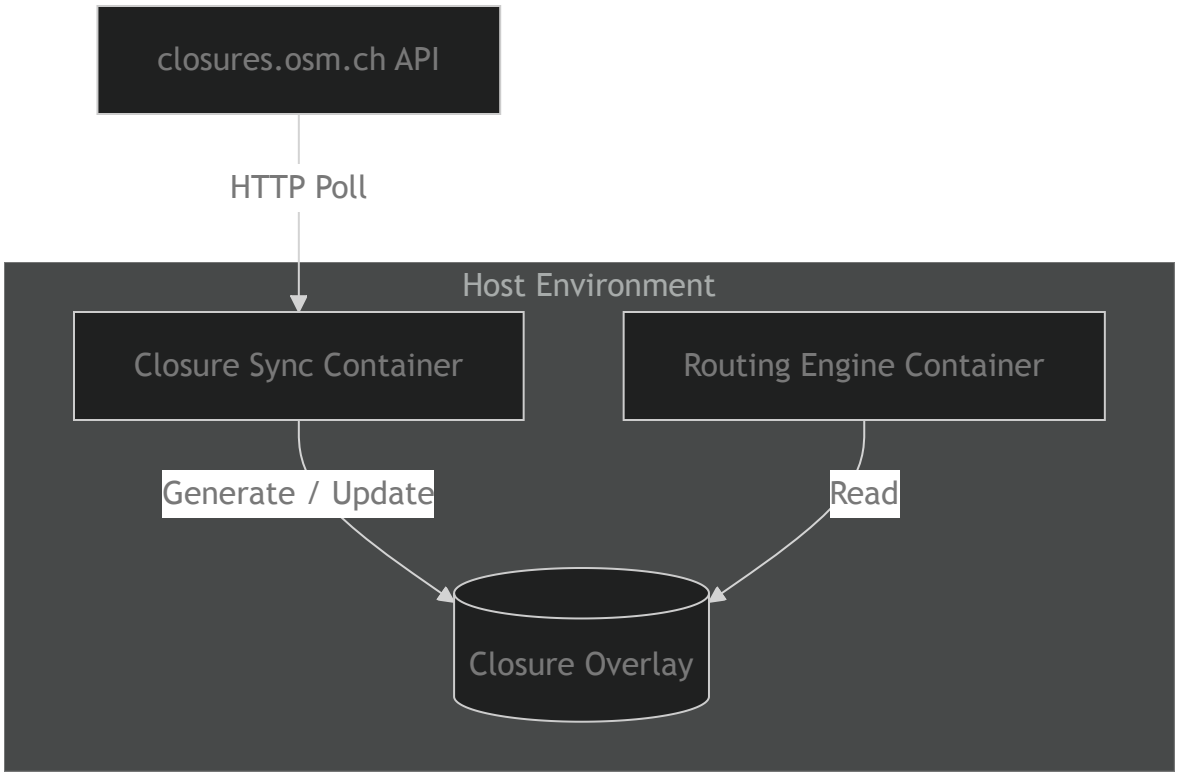
During the elaboration of a possible technical solution, a handful of key considerations were taken into account:

1. **Ease of use:** Having a “plug and play” setup for any combination of supported routing engines and host systems with minimal to no configuration.
2. **Non-blocking:** Protecting the engine’s hot path from obstructions, even if the `closure-sync` service malfunctions.
3. **Native performance:** Maintaining near-zero performance overhead during runtime.

Architecture

The project adopts a **sidecar deployment pattern** to fulfill the core requirements of ease of integration. In this model `closure-sync` runs as a lightweight and **independent container** alongside the primary routing engine within the **same host environment**. By decoupling the synchronization logic from the routing core the architecture ensures a **non-blocking app lifecycle.**

The service functions by asynchronously ingesting data from `closures.osm.ch` and translating it into a **native closure overlay format** compatible with the routing engine's internal traffic APIs.



Valhalla

Before moving on, it's important to mention that for the GSoC project `closure-sync` 's **scope** will be limited to the Valhalla routing engine's integration. For more information on the continuation of the project after GSoC see 'Continuation'.

The choice for Valhalla's initial integration is thanks to its **widespread adoption** in the OSM community and its compatibility with the existing `closures.osm.ch` structures. Moreover, Valhalla's `pyvalhalla` library offers an excellent high-level interface for graph interactions.

Approach

Having originally proposed the addition of helper functions in Valhalla's core C++ logic, I was gently guided to the Live Traffic API by the maintainers. This steered the technical direction toward a **data-driven integration** rather than a structural one. By pivoting to Valhalla's native support for binary traffic tiles (.tar) `closure-sync` can influence routing costs at runtime without modifying the engine's source code.

Internal pipeline

`closure-sync` 's core loop can be summarized by the following the steps:

Fetch & diff external closure data

Before requesting any data from `closures.osm.ch`, the service finds out what area the core graph's `bbox` covers by scanning the **Valhalla tile directory**. Thereafter, `closure-sync` will poll `closures.osm.ch` on a user-configured **time interval** via HTTP by hitting its `GET /api/v1/closures` endpoint.

On a successful response `closure-sync` will parse the JSON response and diff for any updated closure data. Ideally, throughout the project `closures.osm.ch` will be extended to accept an `updated_after=timestamp` query parameter to reduce network overhead. Nevertheless, `closure-sync` will require fallback diffing capabilities warranting an internal option.

Parse traffic geometry

Despite returned closure objects from `closures.osm.ch` containing both **GeoJSON** and **OpenLR** for closure geometries, OpenLR is generally preferred for Valhalla edge resolution due to it's inherent **map-agnosticism**.

Furthermore, while it is true that Valhalla already has an internal OpenLR decoder, it unfortunately doesn't have a **Python binding** yet. In the meantime, a **pip package** like `openlr` can be used for OpenLR string decoding.

Note: PoC testing suggests the current OpenLR implementation in the `closures.osm.ch` backend is not yet fully aligned with the standard. Upstream contributions to address this will be considered within the scope of the project.

Resolve to graph IDs

At this point, we hit a fork in the road where two viable approaches can possibly be used. Firstly, `pyvalhalla` already features a `trace_attributes` method that takes a **GPS trace** or a set of **latitude/longitude positions** and returns the **attributes** of the graph edges along the trace including their `edge.id`. This call uses `meili` to map match the coordinates to the nearest valid graph edges introducing some **probabilistic** properties to the resolution, leading to an expected accuracy of around 90%. This approach is reminiscent of the previously mentioned accuracy concerns in `closures.osm.ch`.

Alternatively, you can treat each **union of two successive Location Reference Points** as a **separate routing request** to trace the closure along each graph edge, storing their corresponding IDs along the way. This effectively increases the trace accuracy to 99% by assuring the edges form a **valid contiguous road section**.

While the first option works to setup the **initial functionality** and can increase **stability** by serving as a **fallback resolver**, the second option should ideally be adopted as the **main approach**.

Build & replace traffic.tar

To create the initial live traffic skeleton binary we can call `valhalla_build_extract --with-traffic` once at setup time. This will scaffold the tile structure with the corresponding headers and reserves zero bytes for the edges.

Unfortunately, there's no clean `pyvalhalla` method to inject the closure structs into the .tar skeleton. This means that we'll need to meticulously recreate the traffic speed struct ourself using `struct` from the **Python standard library**.

To efficiently write the closures we firstly open an `mmap` for each traffic tile contained in the .tar binary. Then, we can write the structs in-place to the correct location by calculating the offset and using the edge ID as its index. Finally, we flush and close the `mmap`, repeating this for every tile file in the .tar binary until completing the graph.

Schedule & milestones

- **Week 1-2:**
  - Setup communications & development environment
  - Validation research
  - Baseline benchmarking
- **Week 3-4:** — ***Milestone:*** *A working pipeline from closures API to edge IDs (no traffic integration yet)*
  - `closure.osm.ch` polling & diffing mechanism
  - Containerized sidecar setup
  - Basic edge resolution with GeoJSON & `trace_attributes`
- **Week 4-5:**
  - OpenLR upstream contribution
  - Setup geometry decoding
- **Week 6-7:** — ***Milestone:*** *Reliable and well-evaluated edge resolution pipeline*
  - Edge ID tracing with routing requests approach
  - Benchmarking accuracies for both approaches
  - Fallback mechanism setup
- **Week 8-9:**
  - Implement struct packing
  - Offset handling
  - Handle tile indexing logic

- **Week 10-11:** — ***Milestone:*** *Proven ability to influence Valhalla routing through traffic tile manipulation*
  - Opening and managing memory-mapped files
  - Ensuring safe writes
  - Race condition handling
- **Week 12-13:** — ***Milestone:*** *Complete working* `closure-sync` *service operating as a sidecar*
  - Orchestrate main service loop
  - Handle failures without breaking routing engine
  - Logging + observability
- **Week 14-15:** — ***Milestone:*** *Production-ready prototype with documentation and reproducible demo*
  - Writing extensive documentation
  - Finishing full test suite
  - Upstream `updated_after` query param contribution

Note: The project entails continuous test integration across all phases, and is therefore not listed explicitly in the schedule.

Continuation

Seeing as GSoC will solely follow the **initial phase** of `closure-sync`, here are some **subsequent technical challenges** the project can take on.

Performance improvement

Once the service takes shape and all basic functionality is present, major **performance bottlenecks** can gradually be identified. As the amount of handled closures increases, we can begin looking where **parallelisation** is most beneficial.

On top of that, to increase the **"ease of use factor"** we can consider a (partial) rewrite of the service into a **compiled language** like **Go**.

Other server-based routing engines

With **routing engine agnosticism** being one of `closure-sync`'s biggest points of interest, the logical continuation after Valhalla would be to adapt **more OSM routing engines**. Preliminary research indicates that the majority of the internal pipeline can be refactored for **OSRM**.

OSRM uses a very similar system to Valhalla when it comes to **dynamic traffic ingestion**. The only major difference being the **data format** in which closures would be represented. Namely, where

Valhalla uses a .tar binary, OSRM uses a **CSV file**.

Graphhopper is where things start to get **tricky**. Their weighting API sadly **doesn't expose IDs** cleanly and the IDs themselves are not stable during runtime, making identity based edge manipulation very difficult. There have been efforts to associate OSM way and node IDs to the internal graph, but there's no existing **external data injection system** like in the other engines. Integrating `closure.osm.ch` data would therefore definitely require a more **extensive** and **invasive** effort than the other mentioned engines.

Mobile applications

For OSM-based mobile apps like **Comaps** and **OsmAnd**, where most if not all routing is done on-device, `closure-sync` 's sidecar architecture is not as viable. Both apps are made with an "**offline-first**" approach making their graphs inherently more **static** than Valhalla or OSRM. If an adoption for `closure-sync` were to be made, the apps' **source-code** would significantly need to be **extended**. Not impossible, but a whole new kind of challenge for sure.

AI use

Over the last few months I have made extensive use of AI to assist me in my preparations for GSoC. Coming from a background oblivious to open-source structures and etiquette, the use of AI has helped to guide me through the (at times quite confusing) process.

Nevertheless, I'm fully aware of the cognitive debt and false sense of confidence AI tools can inspire. Due to this, I have made an effort to "wean off" AI tools as the submission period approached. I can say in full honesty that this submission is near 100% human written. I have tried to build a genuine understanding of all the topics I'm proposing, hence the many links to sources.

I HAVE used AI in the writing of this proposal in areas where I'm less familiar, and where no detailed documentation could easily be found (e.g. .tar file IO, milestone planning). Here, I used AI to gain a base-level understanding and to bounce off ideas, never blindly copying the output's content.