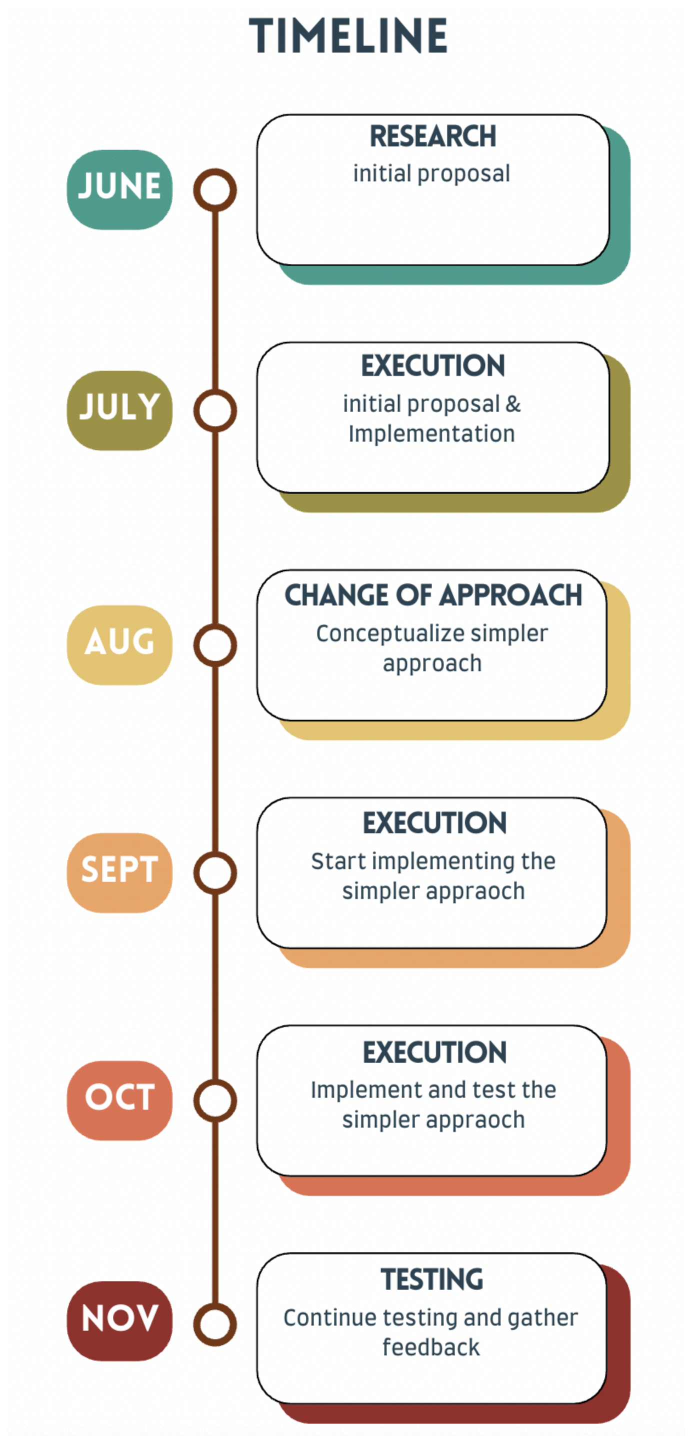
Differential LLMing

Abstract

EIPs are implemented independently by several execution and consensus layer client teams. Each of them make their own design and implementation decisions. During the development phase, this may lead to divergences which may lead to security risks, consensus breaks or edge cases that may lead to bugs. Manual peer reviews and bi-weekly All Core Dev calls are not scalable enough to tackle the above cases. Often they are found out during extensive testing phases if not later. LLMs can't help in this case out of the box due to context length limitations. The idea of this project is to create a program that uses LLM to check if EIP or specs are implemented correctly in a particular client (e.g. prysm), produce a relevant report, then use the reports to compare implementation across clients.
https://github.com/githubgitlabuser/specprover

Brief Timeline

June/July - Do research to come up with idea for implementation (based on embedding)
July - Start implementation of this initial approach
August end - Revamp the whole approach to a much simpler search based approach
September - November - Implement and test the most recent approach



TIMELINE

JUNE — **RESEARCH**
initial proposal

JULY — **EXECUTION**
initial proposal &
Implementation

AUG — **CHANGE OF APPROACH**
Conceptualize simpler
approach

SEPT — **EXECUTION**
Start implementing the
simpler appraoch

OCT — **EXECUTION**
Implement and test the
simpler appraoch

NOV — **TESTING**
Continue testing and gather
feedback

Initial Approach

The initial approach is explained in more detail in my initial proposal plan: https://github.com/eth-protocol-fellows/cohort-six/blob/master/projects/differential-llming.md

In short it included an *existing* agent (e.g. autogen) which will be the key brain behind looking for implementation of specs/eip features inside a codebase. This will be supported by MCP servers that correspond to codebase, eip knowledgebase etc. The knowledgebase or code would be embedded and stored in a vector storage and exposed via an mcp server which will support embedding based search.

The key issue with the above approach was over reliance on an existing agent. When I started implementing it became clear that:

- The existing agents don't quite have that agentic loop that I was looking for: plan - act - note. E.g. plan for searching for a particular feature, act as in using MCP servers to fetch what it needs and note its findings. It wasn't easy to modify the existing ones to carter to my needs.
- Too many broken dependency mismatches.
- Too bloated.

Also the embedding of an entire codebase may take a while.

During this time it became clear that the state of the art agentic tools like Claude Code, Codex etc are quite successful by simply using intelligent search approach. So doing expensive pre-processing like embedding may not be as compulsory as I initially thought. So me and my mentor decided to change to a much simpler approach that is described in the next section.

Current Approach

The current approach is basically mimicking what an actual developer would start working while searching for a particular feature in a codebase. Imagine you are a developer with some basic knowledge of ethereum and given a spec file for fulu hardfork and prysm repo. You will start with figuring out what are the various features (functions, constants, entities etc) that are there in the spec document. Then for each feature you will start with some simple search in the codebase. Once you find somewhere to start looking you can then search more and try to make sense of things and eventually understand enough to say if this feature is implemented fully or partially or not at all with certain level of confidence.

My program takes similar approach which can be summarised in the following steps:

- Parse an input file (spec file or EIP doc) to a list of requirements.
- A requirement is an object with a pre-defined schema which includes fields like id, name, description, pseudocode (if any), and importantly alises. Because not all clients implement the entities in the exact names defined in specs (e.g. spec can have get_data_column_sidecar() but a Go client like prysm might have it as dataColumnSidecar()) so that we can search various forms of a requirement. These aliases can be generated both programmatically as well as using LLMs to get more search suggestions
- A requirement has the following fields now:
    - id
    - kind (("func" | "field" | "constant" | "rpc_method" | "type" | "integration"))
    - name
    - aliases (e.g. `verify_data_column_sidecar` can also be `verifyDataColumnSidecar`)
    - description
    - depends_on (new, not used a lot now but planning to make good use of it)

- Then the program actually does the search based on the aliases mentioned above.
- After each search the program gathers *evidences*. These evidences can be code snippets as well as the entire code file itself.
- The program keeps in mind what should be avoided e.g. _test.go files in a Go codebase like Prysm or generated code like .pb.go.
- Once we have requirements and evidences the program then feeds an LLM one requirement at a time along with its relevant evidences. Then it asks the LLM in prompt regarding the status of the requirement in the codebase and expects a verdict (fully implemented, partially implemented, not implemented) along with some confidence score and reasoning behind its conclusion.
- Then the program writes this output into a file and moves on to the next requirement.
- Once it does it with all the requirements, we have an output file that has details of requirements status.
- This file can be used to both:
    - Know about status of specs or eip in a codebase
    - Compare across ethereum clients regarding the status of the eip/spec
- I have been using LM studio to host an open source model called "qwen/qwen3-coder-30b:2" in my laptop and have run the program in my laptop only. I

had the option to use llm API given by Ethereum Foundation but I decided to save as much cost as possible and focus on getting the program as good as possible so that finally the main bottleneck would be the LLM model itself. Then it will make more sense to use a state of the art LLM API and get much better results.

Current Status & Results

The project code can be found here: https://github.com/githubgitlabuser/specprover

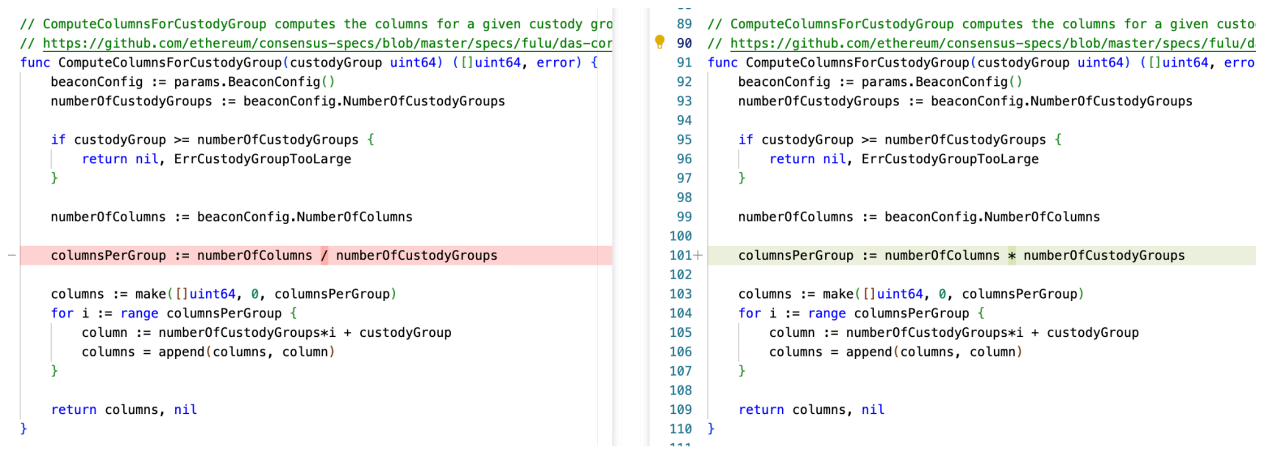
Currently the program is able to:

- Generate report regarding status of EIP/specs with around 80% correctness.
- Identify obvious bugs (e.g. if the implementation has used a minus sign instead of a plus sign in a formula)
- Produce rudimentary comparison report across clients

I am going to show two examples:

- Bug detection
- Comparison between clients

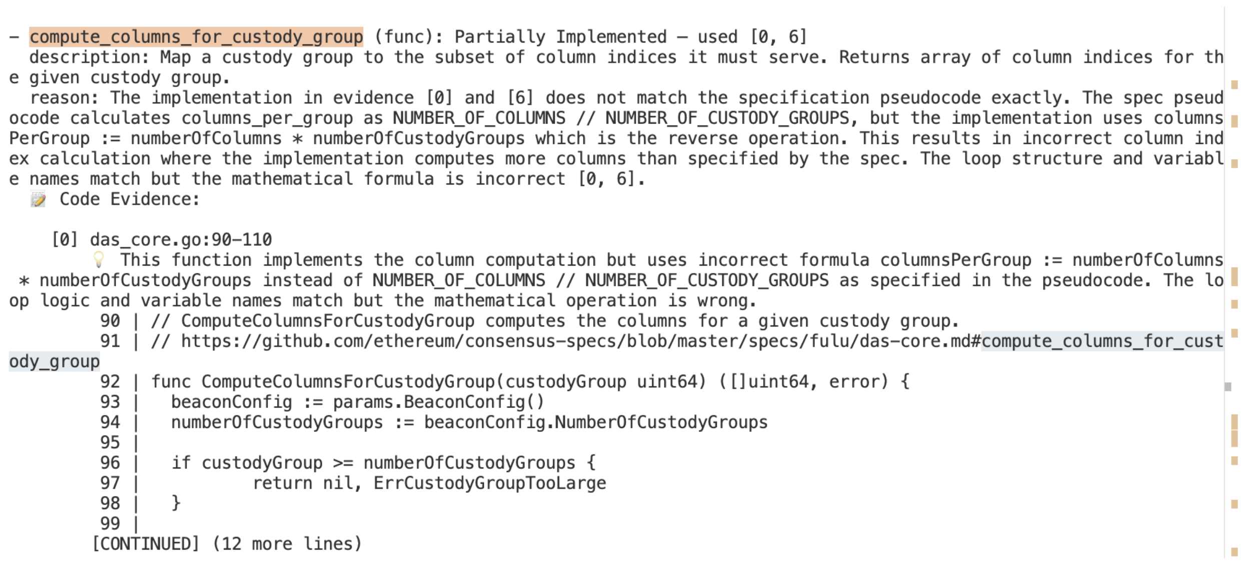
Identification of the bug can be seen in the following two images:



```
// ComputeColumnsForCustodyGroup computes the columns for a given custody gro
// https://github.com/ethereum/consensus-specs/blob/master/specs/fulu/das-cor
func ComputeColumnsForCustodyGroup(custodyGroup uint64) ([]uint64, error) {
    beaconConfig := params.BeaconConfig()
    numberOfCustodyGroups := beaconConfig.NumberOfCustodyGroups

    if custodyGroup >= numberOfCustodyGroups {
        return nil, ErrCustodyGroupTooLarge
    }

    numberOfColumns := beaconConfig.NumberOfColumns

-   columnsPerGroup := numberOfColumns / numberOfCustodyGroups

    columns := make([]uint64, 0, columnsPerGroup)
    for i := range columnsPerGroup {
        column := numberOfCustodyGroups*i + custodyGroup
        columns = append(columns, column)
    }

    return columns, nil
}
```

```
// ComputeColumnsForCustodyGroup computes the columns for a given custo
// https://github.com/ethereum/consensus-specs/blob/master/specs/fulu/d
func ComputeColumnsForCustodyGroup(custodyGroup uint64) ([]uint64, erro
    beaconConfig := params.BeaconConfig()
    numberOfCustodyGroups := beaconConfig.NumberOfCustodyGroups

    if custodyGroup >= numberOfCustodyGroups {
        return nil, ErrCustodyGroupTooLarge
    }

    numberOfColumns := beaconConfig.NumberOfColumns

+   columnsPerGroup := numberOfColumns * numberOfCustodyGroups

    columns := make([]uint64, 0, columnsPerGroup)
    for i := range columnsPerGroup {
        column := numberOfCustodyGroups*i + custodyGroup
        columns = append(columns, column)
    }

    return columns, nil
}
```

Fig 1: Introduction of a bug

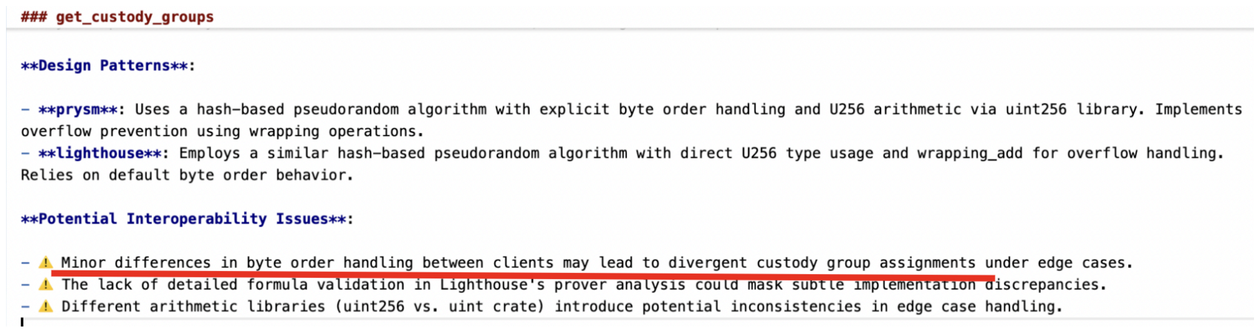


```
- compute_columns_for_custody_group (func): Partially Implemented - used [0, 6]
  description: Map a custody group to the subset of column indices it must serve. Returns array of column indices for th
e given custody group.
  reason: The implementation in evidence [0] and [6] does not match the specification pseudocode exactly. The spec pseud
ocode calculates columns_per_group as NUMBER_OF_COLUMNS // NUMBER_OF_CUSTODY_GROUPS, but the implementation uses columns
PerGroup := numberOfColumns * numberOfCustodyGroups which is the reverse operation. This results in incorrect column ind
ex calculation where the implementation computes more columns than specified by the spec. The loop structure and variabl
e names match but the mathematical formula is incorrect [0, 6].
  📝 Code Evidence:

    [0] das_core.go:90-110
        💡 This function implements the column computation but uses incorrect formula columnsPerGroup := numberOfColumns
 * numberOfCustodyGroups instead of NUMBER_OF_COLUMNS // NUMBER_OF_CUSTODY_GROUPS as specified in the pseudocode. The lo
op logic and variable names match but the mathematical operation is wrong.
        90 | // ComputeColumnsForCustodyGroup computes the columns for a given custody group.
        91 | // https://github.com/ethereum/consensus-specs/blob/master/specs/fulu/das-core.md#compute_columns_for_cust
ody_group
        92 | func ComputeColumnsForCustodyGroup(custodyGroup uint64) ([]uint64, error) {
        93 |   beaconConfig := params.BeaconConfig()
        94 |   numberOfCustodyGroups := beaconConfig.NumberOfCustodyGroups
        95 |
        96 |   if custodyGroup >= numberOfCustodyGroups {
        97 |           return nil, ErrCustodyGroupTooLarge
        98 |   }
        99 |
       [CONTINUED] (12 more lines)
```

Fig 2: The code can identify that bug in its report as evident in this screenshot

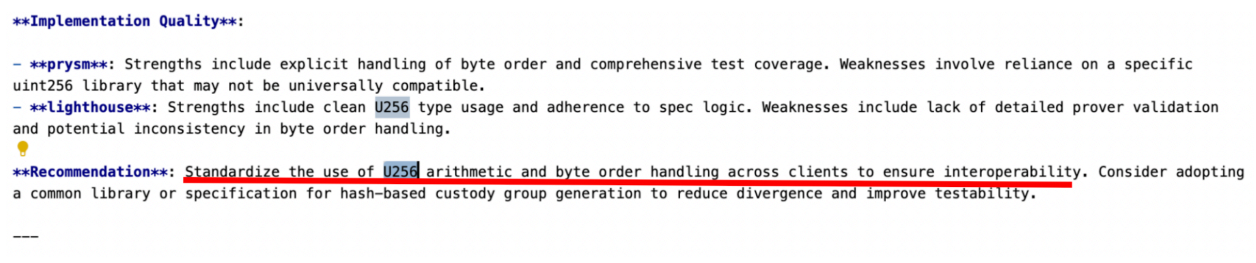
I deliberately introduced a bug in the code (by replacing division with multiplication) and the program was able to find the bug and said: “implementation does not match specs pseudocode”. This can be very useful in the grand scheme of things as one single program can ensure there are no or very few obvious bugs across all clients of Ethereum. This will act as a good sanity check at the very least.

Below is example snapshots from the comparison program that compares a small portion of fulu spec’s implementation between Prysm and Lighthouse.



```
### get_custody_groups

**Design Patterns**:

- **prysm**: Uses a hash-based pseudorandom algorithm with explicit byte order handling and U256 arithmetic via uint256 library. Implements
overflow prevention using wrapping operations.
- **lighthouse**: Employs a similar hash-based pseudorandom algorithm with direct U256 type usage and wrapping_add for overflow handling.
Relies on default byte order behavior.

**Potential Interoperability Issues**:

- ⚠️ Minor differences in byte order handling between clients may lead to divergent custody group assignments under edge cases.
- ⚠️ The lack of detailed formula validation in Lighthouse's prover analysis could mask subtle implementation discrepancies.
- ⚠️ Different arithmetic libraries (uint256 vs. uint crate) introduce potential inconsistencies in edge case handling.
```

Fig 3: Comparison the program makes for get_custody_groups function for prysm and lighthouse



```
**Implementation Quality**:

- **prysm**: Strengths include explicit handling of byte order and comprehensive test coverage. Weaknesses involve reliance on a specific
uint256 library that may not be universally compatible.
- **lighthouse**: Strengths include clean U256 type usage and adherence to spec logic. Weaknesses include lack of detailed prover validation
and potential inconsistency in byte order handling.

**Recommendation**: Standardize the use of U256 arithmetic and byte order handling across clients to ensure interoperability. Consider adopting
a common library or specification for hash-based custody group generation to reduce divergence and improve testability.

---
```

Fig 4: Recommendation from the comparison program

It can look into the design patterns as evident by a small suggestion of standardizing the u256 arithmetic.

All the results so far are from a locally running Qwen 3 Coder LLM. If run by much better models such as Claude or GPT5 the results are expected to be more accurate. To run a fulu spec for prysm it can take about 4 hours on my laptop (MacBook Pro M4 Max)

Future Work

- More testing and bug fixes to make the program go from 80% accuracy to more than 95% accuracy. Achieving 100% accuracy won't be practical with the current setup.
- Run the program with a much better LLM (perhaps Anthropic or OpenAI API) once some of the small bugs are ironed out.
- Make the program agentic: incorporate the agentic loop which was part of my initial plan but I was relying on an external open source agent (which was not quite ready then). But I think I can create the agentic loop myself in the future with the experience I have gained in the past few months.
- Purchase a GPU that supports CUDA so that I can run vLLM to run the inferencing. This is because vLLM has paged attention that can help parallelizing the program. Currently it runs sequentially for one requirement at a time as LM Studio doesn't directly support multiple requests at the same time. In theory I can run multiple versions of the same LLM but it will be too compute heavy.
- Finetune local model with latest:
    - Programming language
    - Codebase
- Modularize the code. Have different components responsible for segregated work to ensure separation of concerns.

Self Evaluation

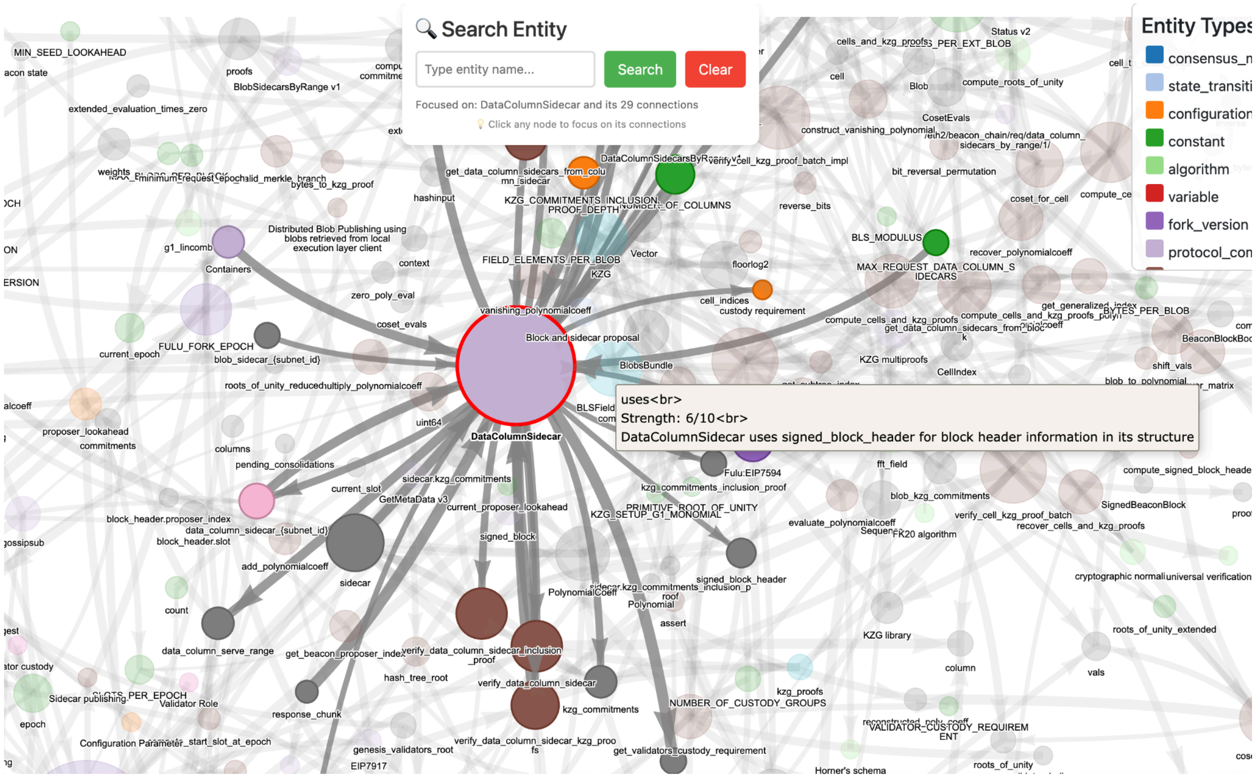
It has been an excellent journey as as protocol fellow and I have thoroughly enjoyed it. I thank Ethereum Foundation, Mario, Josh, my mentor Fredrik for all the support. I also thank all the other fellows for their wonderful time, work and experience. From EthCC to DevConnect, I got to meet and learn from several key people in Ethereum ecosystem which wouldn't have been possible without this fellowship program.

My initial approach was too reliant on a third party solution which ended up not working well and I changed my approach eventually. This is something of a learning: start simple and small but while keeping full control over things so that you minimise the unknowns.

I plan to continue working on my project and make it production ready so that in future we will get automatic reports on status of eips or specs for client teams.

Miscellaneous

Although not directly part of the epf work, I created a knowledge graph visualizer for Fulu specs which can be seen here:
https://githubgitlabuser.github.io/specs-knowledge-graph/fulu_real_knowledge_graph.html



Focused on: DataColumnSidecar and its 29 connections

💡 Click any node to focus on its connections

uses
Strength: 6/10
DataColumnSidecar uses signed_block_header for block header information in its structure

I am planning to maintain it and make such knowledge graphs for other hardforks as well as lean ethereum specs in the future.

The key aspect here is the search feature. If you are reading specs and missing some big picture, you can search for that particular entity of the spec in this visualizer and immediately know what other entities or functions or constants etc relate to this entity along with some descriptions. This will help in having the bigger picture in mind easily. This has somewhat helped me while reading fulu specs.

Links

https://github.com/githubgitlabuser/specprover

https://github.com/githubgitlabuser/specs-knowledge-graph

https://github.com/ethereum/EIPs/tree/master/EIPS

https://github.com/ethereum/consensus-specs/tree/master/specs