CLI/TUI for SDC

Term Project

Department of Computer Science
OST - University of Applied Sciences
Campus Rapperswil-Jona

Version: 1.1
December 19, 2025

Autumn Term 2025

Project Team: Andreas Bachmann, Natalie Breu, Sabrina Forster
Advisor: Severin Dellsperger
Co-Advisor: Yannick Städeli
Project Partner: Nokia - Wim Hendrickx, Markus Vahlenkamp

Abstract

Automation is becoming increasingly important in network engineering. Kubernetes is a widely known platform that ensures components run in their desired state by documenting this state in YAML files. YANG is a data modeling language that defines the structure and constraints of network configuration , enabling validation and automation. Schema Driven Configuration (SDC) adopts a similar principle for managing network devices, using Kubernetes extensions called Custom Resource Definitions (CRDs). These CRDs allow users to define their own resources.

In SDC, a CRD that specifies configuration changes for a device is called an intent. An intent is structured according to YANG schemas. YANG provides standardized blueprints for configuration options and comes preinstalled on network devices. SDC retrieves these schemas and translates intents into valid configuration that devices can process, continuously checking whether the running state aligns with the desired intent. SDC will check with the target device if the running configuration aligns with the intent.
While powerful, this approach introduces challenges: writing intents manually is time-consuming, error-prone, and requires detailed knowledge of YANG, YAML and Kubernetes. As a result, adoption is slowed by the steep learning curve, and many network engineers fall back to the traditional Command Line Interface (CLI), which feels faster and more familiar. Yet SDC offers a more sustainable solution by centralizing configuration management, providing blame views of devices, and ensuring they remain in the desired state. However, schema and blame information are currently scattered across different endpoints, leaving network engineers without a single place to access all necessary configuration data.

This project addresses these usability gaps by developing a prototype Terminal User Interface (TUI) for SDC. The TUI employs a split-window design: one side provides a vendor- and version-independent command line interface (CLI), while the other visualizes the YANG schema tree and running configuration. With this solution, network engineers can explore configuration options intuitively, view the active state of devices, and configure intents in one location. Commands entered in the CLI window part are supported by static and dynamic autocompletion, reducing typing effort and mistakes. Once changes are specified, the network engineer can generate the corresponding intent, which can then be pushed to SDC.

The result is a Minimum Viable Product (MVP) that demonstrates the feasibility of simplifying intent creation through an interactive interface. Although not yet production-ready, the prototype establishes the core concept and highlights the opportunities for further improvement, including schema filtering, full intent validation and direct application or deletion of intents within SDC. This work lays the foundation for a more accessible, efficient and sustainable way of working with SDC in future network environments.

Management Summary

Initial Situation
In todays organizations, countless devices such as laptops, servers, and applications rely on the underlying network to communicate. The critical task of keeping traffic flowing between these endpoints falls to network devices like switches, routers, and firewalls. As networks grow larger and more complex, ensuring that these core devices are configured correctly becomes a major challenge. Traditionally, engineers had to access each device individually and enter configuration commands by hand. This manual approach is slow, prone to errors, and increasingly impractical when hundreds of interconnected devices must be managed simultaneously.

To make this easier, vendors began offering their own management tools. However, these tools usually only work with devices from the same vendor or even same model. In practice, most networks are built from equipment supplied by different vendors, which means engineers still face the problem of juggling multiple systems and learning different ways of configuring different devices.

Schema Driven Configuration (SDC) was developed to solve this issue. It provides a standardized way of describing how devices should be set up, regardless of the vendor. In theory, this makes network management more efficient and less dependent on proprietary solutions. In practice, however, using SDC requires engineers to write special configuration files also known as Kubernetes manifests. Many engineers are not familiar with this format, and the process can feel complicated and unintuitive compared to the command-line tools they are used to.

This situation creates a gap: while SDC offers a powerful and vendor-neutral solution, its usability is limited by the complexity of the current interaction method. A simpler, more accessible interface is needed to help engineers take advantage of SDC without requiring deep knowledge of Kubernetes or specialized file formats.

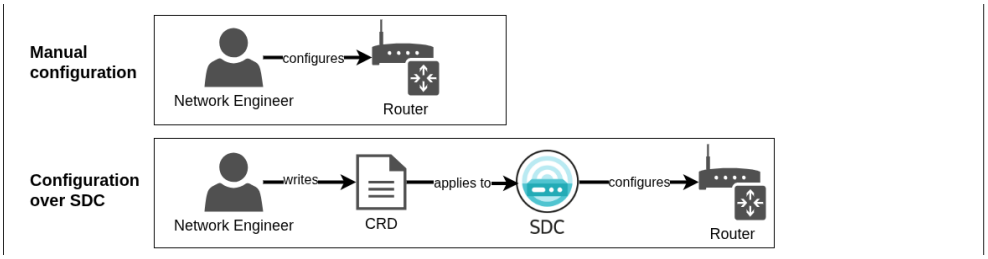

Figure 1.: Manual configuration and SDC configuration workflow.

Procedure and Technology
In order to capture all requirements in detail, the project team discussed and prioritized use cases together with the industrial partner and academic advisor. These were divided into mandatory and optional features to ensure that the most critical aspects were addressed first. In addition, non-functional requirements such as usability, responsiveness, and maintainability were defined

to guarantee that the application would meet high standards in practice.

These requirements led to an extended elaboration and prototype phase. During this phase, the architecture was carefully designed and tested in a minimal application to verify that the SDC API endpoints were reachable and that the planned features could be implemented. This early validation ensured that the design was feasible and aligned with the expectations of both the partner and the users. As a result, the elaboration phase was lengthened by one week, while the development phase was shortened accordingly.

The most significant phase was the construction and development phase, in which all prioritized features were implemented and tested. The product was developed in the programming language Golang, chosen for its performance and concurrency capabilities. For the Terminal User Interface, the Charm framework was used to design the layout and provide interactive elements. The resulting tool integrates three main components:

- CLI window for entering commands directly.
- Schema view for visualizing YANG options and available configuration parameters.
- Blame view for displaying the currently active configuration of a selected datastore.

In SDC terminology, a target is referred to as a datastore, and a CRD holding configuration for a target, it's called intent. The SDC TUIsupports this workflow by allowing users to create intents through the TUI. Once the configuration is completed in the CLI, the tool automatically translates the users input into a valid CRD file, which SDC then monitors and applies to the network devices.

To improve usability, the TUI also features autocompletion, reducing the need to type commands manually and minimizing the risk of errors. This combination of CLI efficiency, schema visualization, blame view transparency, and automated CRD generation makes the tool both powerful and user-friendly. In summary, the product is a TUI with an integrated CLI and schema/blame visualization, designed to simplify the complexity of schema-driven configuration and make it accessible to network engineers.

Results
The MVP provides a solution to make configurations for SDC simpler and more intuitive. Instead of writing technical files, users interact with a terminal app that guides them through available options and shows what is currently active.

What it does today:

- **Command input:** Users type simple commands rather than editing long configuration files.
- **Schema view:** Options are displayed in a clear, interactive tree, similar to browsing categories in a menu.
- **Blame view:** The active settings on a chosen target are shown transparently.
- **Autocompletion:** Helpful suggestions appear while typing, reducing mistakes and speeding up work.
- **Automate CRD generation:** Commands are converted into the format required by SDC.

What's working:
Core navigation: Choosing a target (datastore), browsing options, and viewing active settings is

smooth and reliable.
Speed and usability: The app responds quickly, remembers command history, and presents results clearly.

What's still limited:
CRD correctness: The generated configuration file is not fully valid in some cases.
Model coverage: The current version supports basic building blocks ("leafs" and "containers") but does not yet handle more complex structures (lists and leaf-lists), which require special treatment.
Optional features: Some convenience functions (like filtering the schema, applying or deleting configurations directly) were postponed to focus on the essentials.

In short, the MVP meets the most important goals: it reduces manual effort, makes options easy to find, and shows what's active—while highlighting where deeper model handling and output validation still need to improve.

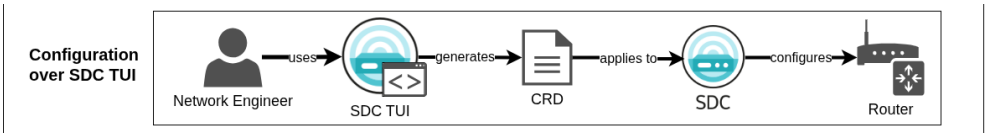

Figure 2.: SDC TUIconfiguration workflow.

Outlook
The MVP demonstrates that a combined CLI/TUI can make Schema Driven Configuration more accessible. The next steps will focus on:

- **Improving reliability:** Ensuring generated files are always valid and expanding support for advanced structures.
- **Enhancing usability:** Adding schema filtering, and smarter autocompletion.
- **Integration:** Aligning the tool with existing automation workflows, so it fits seamlessly into daily operations.

The long-term goal is to evolve from a proof of concept into a robust solution that balances simplicity for non-specialists with the reliability professionals expect.

Acknowledgements

We want to express our gratitude to the following people who have guided and supported us during the term project:

- **Severin Dellsperger**, we want to thank Severin for his support as our supervisor. His valuable feedback and insights have greatly contributed to the quality of this thesis. We also want to thank him for his YANG session.
- **Yannick Städeli**, we want to thank Yannick for his support and his valuable feedback as our testing person and documentation reviewer. His constructive feedback and suggestions have been instrumental in refining our work.
- **Wim Hendrickx**, we want to thank Wim for being able to give us insights about the industrial use of SDC and his feedback during our meetings.
- **Markus Vahlenkamp**, we want to thank Markus for his huge support in understanding intents and giving us a general understanding of SDC, the tree structures and their implementation. We are also grateful for him giving us an insight into his intent builder.
- **Urs Baumann**, we want to thank Urs for his YANG slides and taking part in the YANG session.
- **Israel Leuenberg**, participated as a reviewer for our documentation, we are thankful for his help.
- **André Schmid**, participated as a reviewer for our documentation, we are thankful for his help.
- **Pietro Lehmann**, participated as a reviewer for our documentation, we are thankful for his help.

Contents

List of Acronyms

CLI Command Line Interface. 4, 5, , 9, 15, 16, 25, 26, 30, 43, 45, 78

CRD Custom Resource Definition. 4, 5, , 9, 12, 13, 15, 27, 66–68

MVP Minimum Viable Product. 4, 5, , 9, 17, 18, 25, 45, 63

SDC Schema Driven Configuration. 3–5, , 9, 10, 12–20, 22, 23, 25, 27, 29, 39, 41–44, 62, 64, 66, 67, 74

TUI Terminal User Interface. 4, 5, , 9, 10, 12, 13, 16–20, 22, 23, 25, 27, 39, 41–46, 50–56, 61–64, 66–68, 72, 73, 78

Glossary

API Application Programming Interface. A callable endpoint to interact with other code.. 4, 9, 13, 19, 42–44

Charm Charm is collection of TUI libraries for Golang. 4, 9, 44

Data-server SDC component managing Intents and Datastores.. , 9

Datastore Defines a target device and holds its YANG schema.. , 9, 10

go-critic A linter for Golang. [**go-critic**]. , 9, 70

Golang A programming language.[5]. 4, 9

golangci-lint A linter for Golang. [**golangci-lint**]. , 9, 70

gomock gomock is a mocking framework for Golang. , 9

gRPC Google Remote Procedure Call - Framework to perform remote procedure calls.[**wiki-grpc**]. , 9, 13, 16, 18, 19, 44, 61, 66, 67

Intent Holds configuration changes.. , 9, 10

Protobuff Data defined in XML-like format, generated by protoc.[**protobuff**]. , 9, 44

revive A linter for Golang. [**revive**]. , 9, 70

Schema-server SDC component storing YANG schema information.. , 9

YAML File format like JSON used for configuration.. , 9, 15, 16, 24, 30, 44

YANG Data modeling language to model network configuration and state data.[**wiki-yang**]. 4, 5, 9, 10, 12–16, 18–20, 26, 39, 43, 55, 56

Part I.

Technical Report

1. Introduction

This chapter provides an overview of the document's structure, shows the terms and techniques to provide a basic understanding about Schema Driven Configuration (SDC) and YANG.
The final section of this chapter assesses the aims and objectives of this term project.
In summary this chapter gives an overview of what to expect from this report.

1.1. General

This report is structured into three parts:

- Technical Report, containing the most important information regarding the SDC TUI.
- Project Documentation, giving a more detailed insight of the development process and the implementation of the SDC TUI.
- Project Management, containing the teams approach on how they managed the project including the risk assessment and testing process.

1.2. Terms and Techniques

The TUI is built upon SDC, therefore this section will provide the necessary context to understand it. Since YANG plays an important role in SDC, this section also gives an introduction to this term.

1.2.1. SDC

The SDC (Schema Driven Configuration) project is a tool for managing network devices in a Kubernetes like approach. Through the use of Custom Resource Definition (CRD) the user is able to define their desired configurations. Unlike other solutions for managing network devices, SDC uses the YANG schema as a basis for the CRD structure. In its Kubernetes inspired approach, SDC ensures that the subscribed target always matches the desired state and using the YANG offers significant advantages, including vendor neutrality.

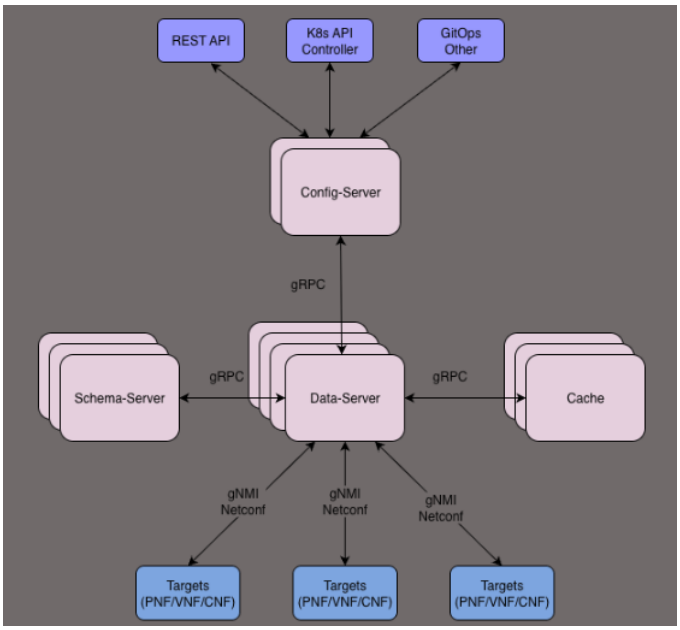

Figure 1.1.: SDC architecture, source:[11]

The SDC architecture consists of four components:

- **Schema-server**, stores the retrieved YANG schemas. This component also provides a gRPC API the SDC TUItakes advantage of, to retrieve schema information.
- **Data-server**, responsible for the datastore setup and keeping track of the running configuration of a network device. Like the schema-server it has a gRPC API. SDC TUIuses this gRPC API to retrieve information about a datastore, intent or the blame.
- **Cache**, is a persistent data storage which stores the datastores, including config, state, along with intent metadata. Also comes with a gRPC API but is for our product not necessary.
- **Config-server**, is a Kubernetes-based operator and incorporates several controllers: schema, discovery and target controller. These controllers are responsible for managing lifecycles. The config-server also consists of a config API server to manage the lifecycle of config resources.

In this report you will often be confronted with the term **datastore** or **intent**. The **datastore** holds the information about which schema, vendor and version a network device uses. An **intent** on the other hand, is the configuration pushed on to SDC. In this report we will talk about creating an intent which means nothing more than creating a CRD so it can later be applied to SDC and become an intent [11]. Figure 1.2 displays an example of how an intent could look like.

```yaml
apiVersion: config.sdcio.dev/v1alpha1
kind: Config
metadata:
  name: intent-example
  namespace: default
  labels:
    config.sdcio.dev/targetName: firewall1
    config.sdcio.dev/targetNamespace: default
spec:
  priority: 10
  config:
  - path: /
    value:
      rule:
        - name: "rule1"
          description: "block specific incoming traffic on ethernet1/1"
          active: True
          incoming:
            interfaces:
              - "ethernet1/1"
            block:
              - 1.1.1.1
              - 8.8.8.8
```

Figure 1.2.: Intent example

1.2.2. YANG

YANG is a data modeling language, structured in a hierarchical tree format. A blueprint displaying the structure of the available configuration options a network device has. This opens the gates for open source solutions and states a standard. Before YANG every vendor had their own approach. YANG differentiates between different types of tree nodes/leafs. The most important ones in context of Schema Driven Configuration (SDC) are the following: [1]

- **Container**, can hold containers, lists, leaf-lists and leafs.
- **List**, is like a container except it has a keys element. Each entry in this list is uniquely identifiable by the value of the list's keys.
- **Leaf-List**, unlike the list component, leaf-lists don't come with a keys element and can only have leafs as children.
- **Leaf**, has an identifier and can store a value. Leaf nodes have no children.

The connection to SDC:
Like mentioned in Section 1.2.1 the SDC schema-server works with YANG and an intent is built based on the YANG model. Figure 1.3 shows an example of an intent with the different YANG components described. Note that this example is illustrative only, to showcase the different node types, and does not represent a valid production configuration.

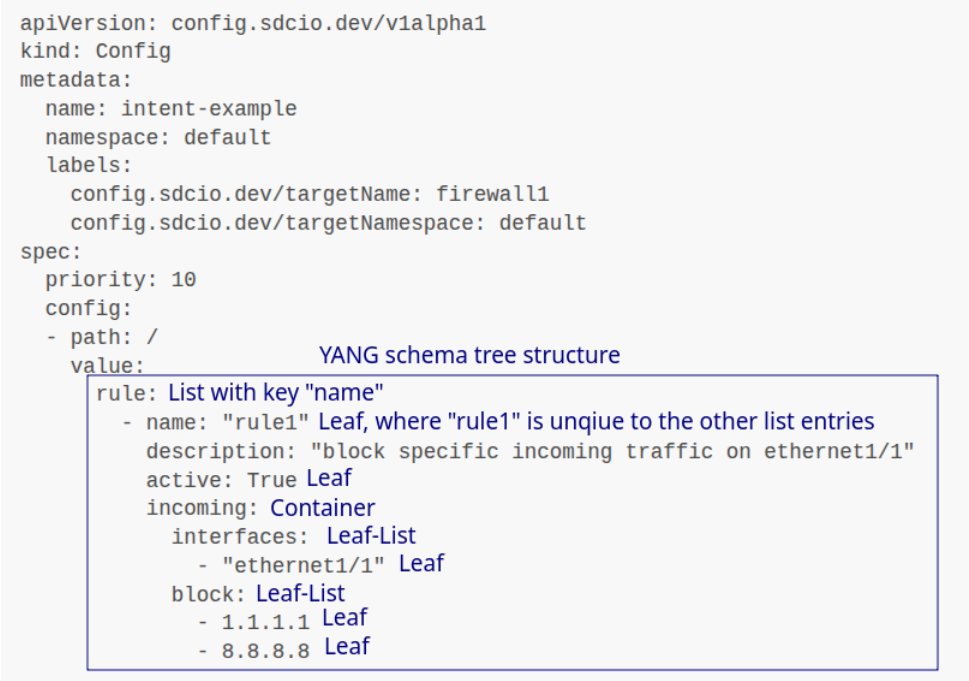

Figure 1.3.: Intent YANG example

1.3. Aims and Objectives

Section Aims and Objectives introduces the problem addressed by the project, followed by its objectives and proposed solution.

1.3.1. Problem

Before network automation was introduced, network devices such as switches and firewalls were configured manually by connecting directly to each device and applying configurations via CLI. As networks grew to hundreds or even thousands of components, this approach became inefficient and error-prone. Vendors responded by introducing centralized solutions for managing their own devices, but these solutions were limited to homogeneous environments where all devices were compatible with the vendors software.

In practice, most networks are heterogeneous, consisting of devices from multiple vendors. This created a new challenge: engineers had to maintain complex configuration management systems, often requiring deep knowledge of each devices protocols and file structures. As a result, workflows became increasingly complicated and difficult to scale.

Schema Driven Configuration (SDC) was introduced to address these challenges. By leveraging standardized SDC schemas, SDC eliminates vendor lock-in and provides a unified way to describe configurations across different devices. However, despite these advantages, the current interaction with SDC presents its own difficulties.

At present, network engineers must create Kubernetes manifests — specifically Custom Resource Definition CRD — to configure intents in SDC. This process requires familiarity with Kubernetes concepts and YAML syntax, which many network engineers do not possess. For engineers accus-

tomed to working with CLI-based tools, the manual creation of YAML manifests feels unintuitive and cumbersome. Furthermore, the complexity of YANG models adds another barrier, particularly for beginners who may struggle to understand the hierarchical structures and node types.

Thus, while SDC solves the vendor neutrality problem, it introduces a new usability challenge: the steep learning curve associated with Kubernetes manifests and YANG schemas. This gap between powerful functionality and practical usability highlights the need for a more accessible solution.

1.3.2. Objective

Knowing about the problems described in subsection 1.3.1. The current interaction with SDC requires the manual creation of Kubernetes manifests, a process that many network engineers find unfamiliar and unintuitive. To address this challenge, the objective of this project is to design and implement a CLI and TUI that simplify interaction with SDC and make it more accessible to network operators.

This objective is pursued through the following structured approach:

- **Requirements Analysis** – Collaborate with the industry partner to gather and analyze requirements, ensuring the CLI/TUI meets the practical needs of network engineers. A more detailed insight into the agreed requirements with the industrial partner, can be found in the requirements' chapter 1
- **Design and Prototyping** – Define the architecture and user interface of the CLI/TUI, and create prototypes to validate design decisions with stakeholders.
- **Implementation** – Develop the CLI/TUI using Go (Golang) and suitable frameworks for building command-line and terminal interfaces.
- **Functional and Usability Testing** – Conduct thorough testing to verify that the CLI/TUI fulfills its intended functionality, offers a user-friendly experience, and performs reliably.
- **Documentation** – Provide comprehensive documentation covering design, implementation, and usage, supporting both end users and developers.

By following this approach, the project directly tackles the problem of complex and unfamiliar Kubernetes manifest creation, offering a more intuitive and efficient way for engineers to interact with SDC. The original assignment can be found in the Appendix **??**.

The final product should feature:

1. **Dynamic Autocompletion via gRPC:**
   Implement a mechanism to fetch available configuration options in real-time using gRPC to guide users.
2. **Intuitive YANG Model Exploration:**
   The CLI/TUI should provide a user experience similar to popular web-based YANG exploration tools like Nokia's YANG documentation[10] or CiscoDevNet's Yangsuite[4] , allowing users to navigate and understand the underlying YANG models directly within the terminal environment. This will empower network engineers to discover and utilize the available configuration parameters efficiently.
3. **Intelligent Validation:**
   Ensure that entered values are correct and that the configuration is ready for deployment on network devices.

1.3.3. Solution

The outcome of this project is a MVP, which will later be extended into a bachelor's thesis.
The MVP has implemented the following features:

- **Static Autocompletion** for the hard coded basic commands.
- **Dynamic Autocompletion** for the datastores, intents and partial schema data.
- **Schema View** which enables an intuitive YANG model exploration.
- **Blame View** to display the active configuration on a datastore.

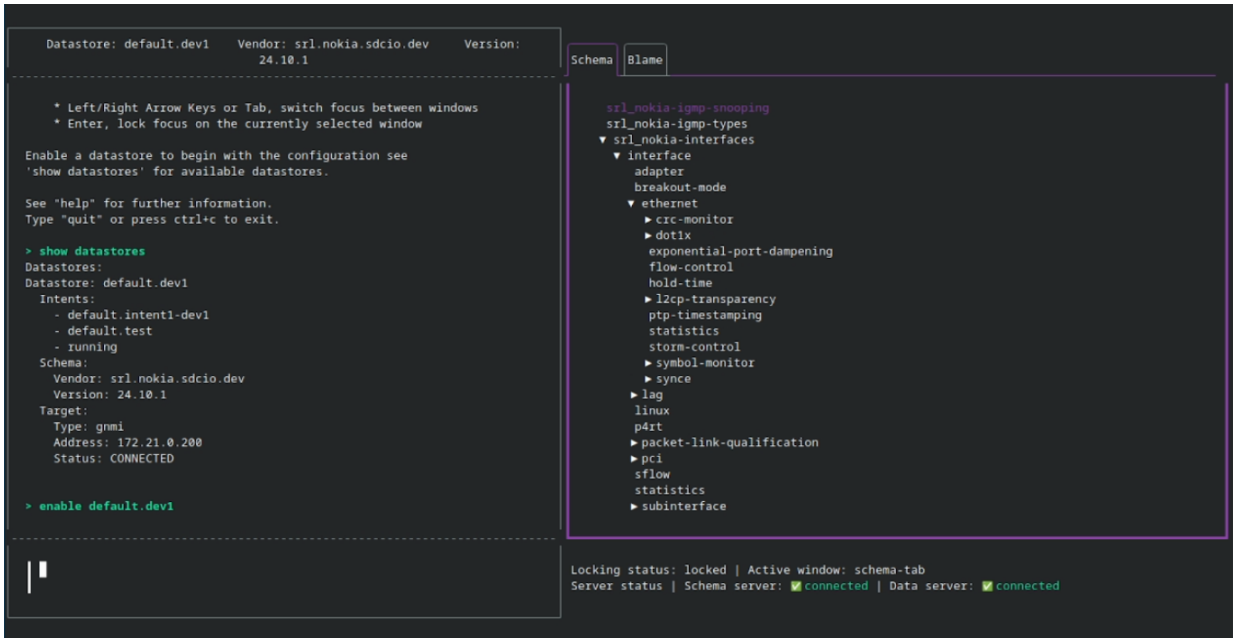

Figure 1.4.: SDC TUI MVP

2. Results

This chapter explores the results of this term project in detail.

2.1. Distinction

In the Achievements section we will reference the schema and data-server of SDC. There were no changes made to the implementation of the data-server, schema-server and their corresponding gRPC APIs remain unchanged.
The TUI has been designed to be executed inside a container within a Kubernetes cluster to have access to the schema-server and data-server.

2.2. Achievements

This section examines the solutions and the features implemented during the term project through the development of a TUI for SDC. The TUI allows the user to explore YANG schemas, view the running configuration in blame format and configure new intents or work with existing intents. During the project all required use cases as described in chapter Use Cases 1.1.2 for the MVP were achieved.

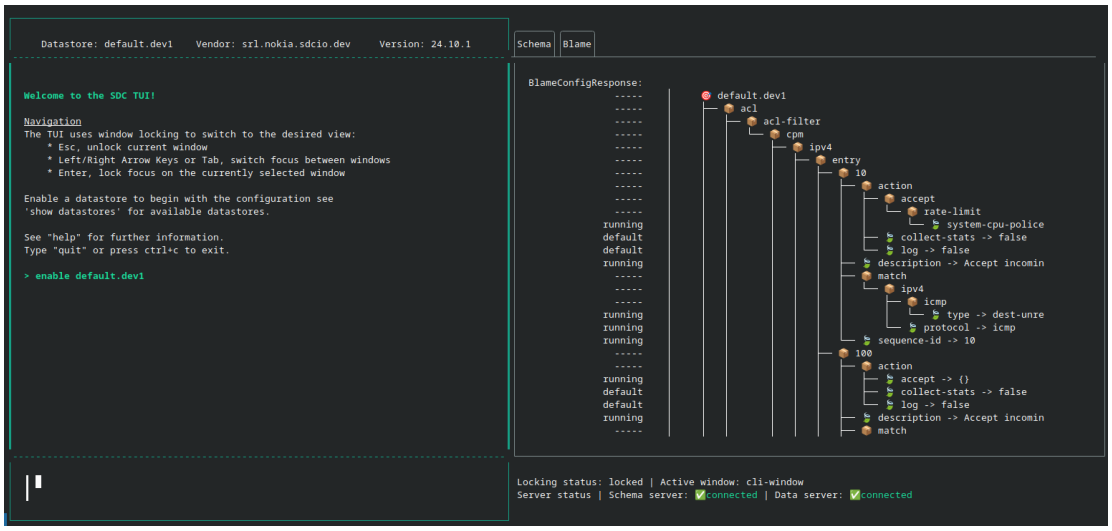

Figure 2.1.: SDC TUI MVP

2.2.1. Autocompletion

The "Autocomplete" function is implemented to help users enter input faster while also indicating the available input options.

The autocompletion is split into two parts: a static and a dynamic part which depends on the

information given from the schema and data-server. Not all information is included in the auto-completion, some is skipped or just returned as it was typed if no match has been found.

Static
The static autocompletion consist of a list including basic commands like `show datastores` or `get` that do not change between targets.
The list of static commands is the basis for all autocompletion elements. Figure 2.2 shows all available static commands.
The dynamic elements are children to the static commands.

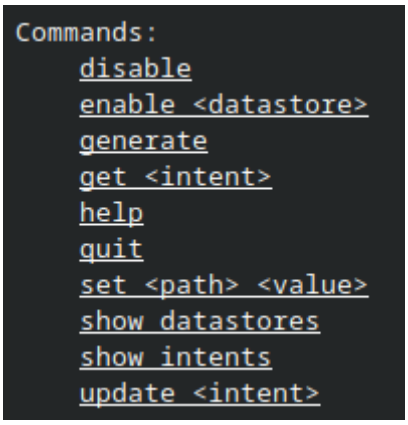

Figure 2.2.: List of all static commands

Dynamic
The dynamic autocomplete uses information gathered from the schema- and data-server to make autocomplete the user input.
The dynamic autocomplete uses the containers, leafs and leaf-lists from the YANG schema. The YANG schema information is retrieved from the schema-server.
From the data-server, the autocompletion retrieves available target names and their supported intents after a target has been selected.

2.2.2. Gather Running Information

The TUI implements processes to get information from the SDC data-server using the gRPC API. The information retrieved from the data-server is displayed to the user, used for dynamic auto-completion or used get further data from data- or schema-server.

Figure 2.3 shows the interaction between the SDC TUIand the data-server. The user is able to display the **available datastores**, **available intents** and the **content of an intent**, all information which was retrieved from the data-server.

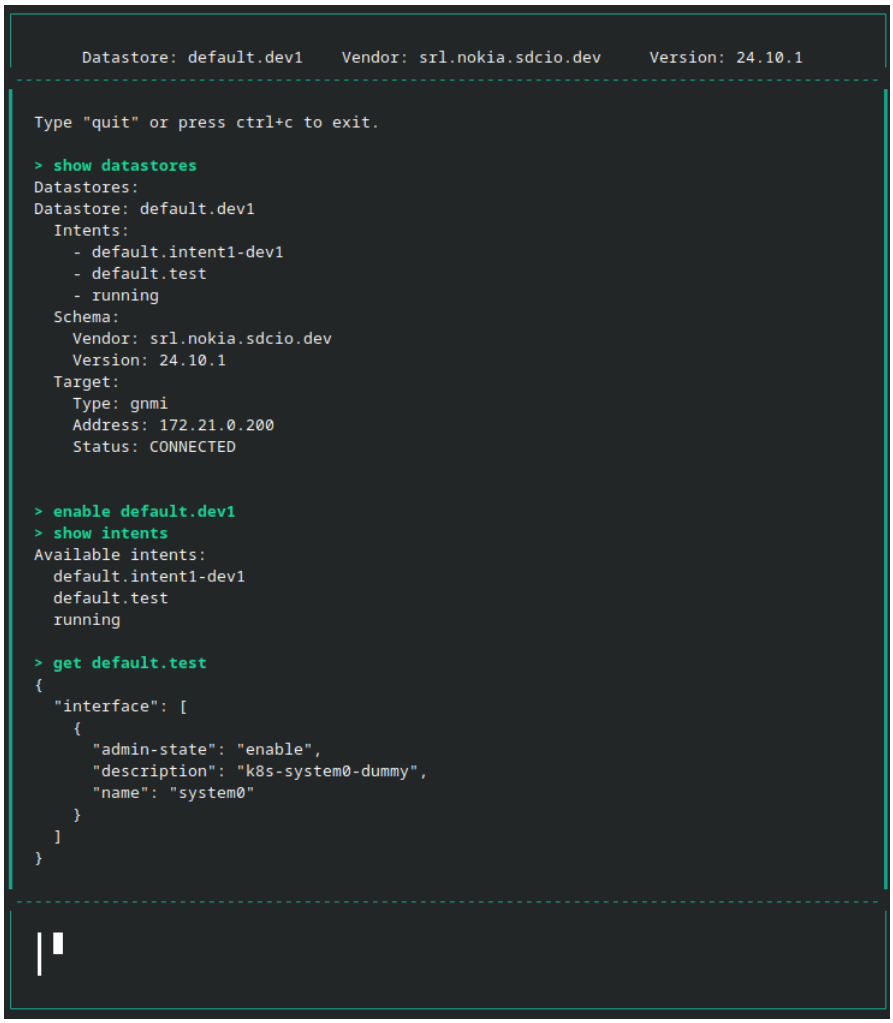

Figure 2.3.: Screenshot of the SDC TUIdisplaying the information retrieved from the data-server

2.2.3. Gather Schema Information

Once the user has chosen the target the first levels of the YANG schema information tree is loaded from the schema-server. Further schema information is loaded when the parent object is viewed or used for dynamic autocompletion. This is for performance reasons, if the schema would be loaded in one batch the TUI could lock up for multiple seconds.

2.2.4. Visualize Schema and Data Information

The fetched information needs to be displayed to the user, during the configuration process, for it to be of use.

YANG schema exploration
The schema tab allows a user to explore the YANG schema. The user can expand the elements of interest to see the possible configuration options.

Figure 2.4.: Schema Visualization

Blame Visualization
The blame tab allows a user to explore the running configuration on the selected target and see which intent is active for a specific datastore.

Figure 2.5.: Blame Visualization

2.2.5. Intent Configuration

To configure intents there are two possible ways:
The user can generate a completely new intent from scratch or add updates to an existing intent.

Configure new intent
Using the `set <path> <value>` command a new intent can be configured.
If the user uses the same `set` command and path but with a different value, then the previously set value gets overwritten.

```
> enable default.dev1
> set interface ethernet-1/1 description test
Intent set successfully
```

Figure 2.6.: Screenshot of the SDC TUIafter the `set` command was executed

Update existing intent
To update an existing intent the user needs to select the desired intent using the `update <intent>` command.
Afterwards a user can use the `set <path> <value>` command to modify or add more configuration. Figure 2.7 displays an example of an updating workflow.

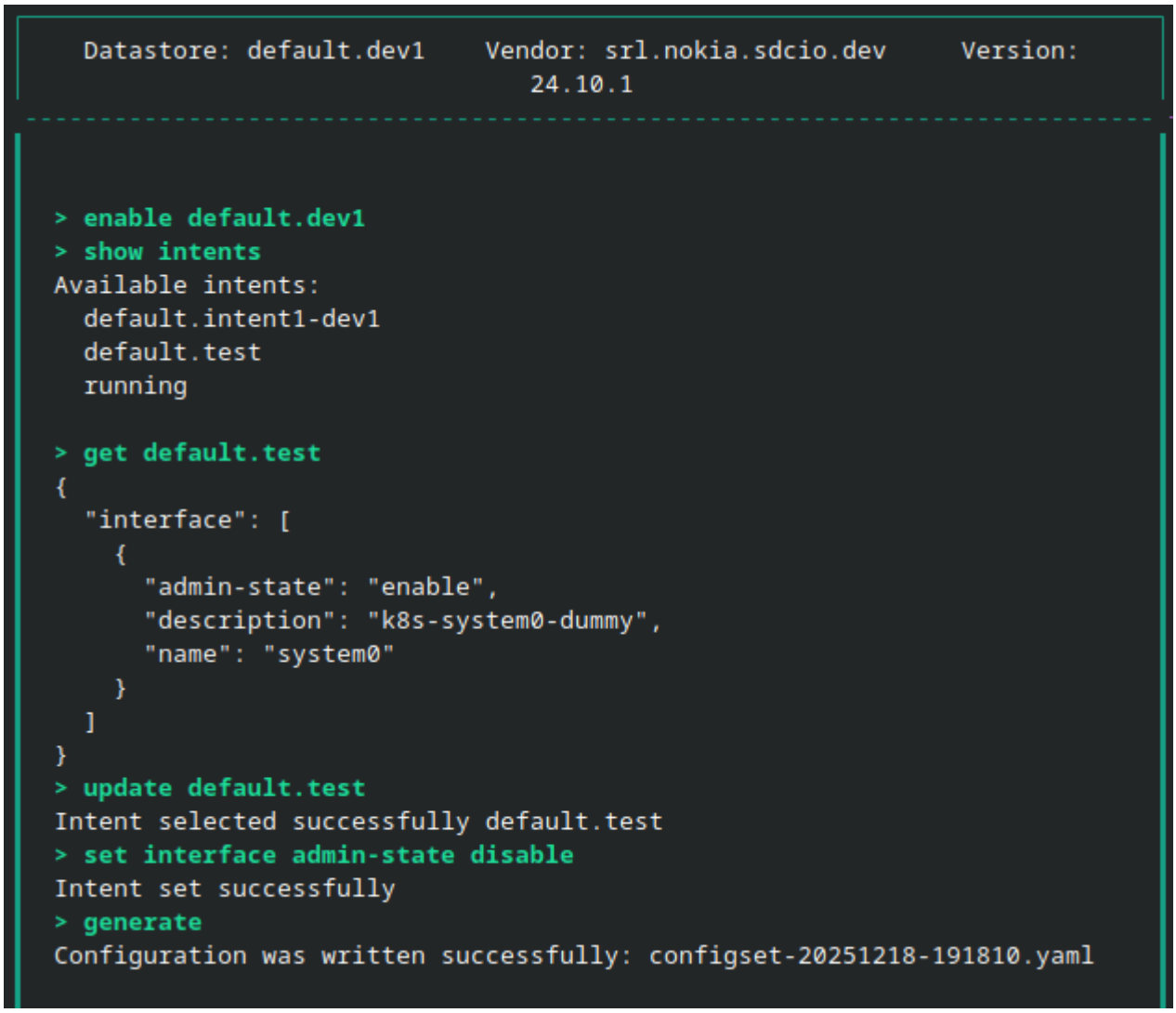

Figure 2.7.: Screenshot of the SDC TUIfor updating an existing intent

2.2.6. Connection Status

The status banner indicates to the user if the connection to the schema- and or data-server has been successfully initiated, has failed or is still loading.

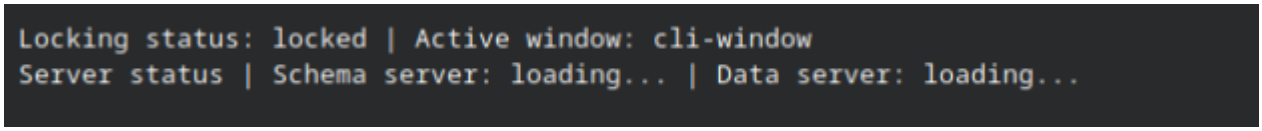

Figure 2.8.: Connection Status in the Beginning

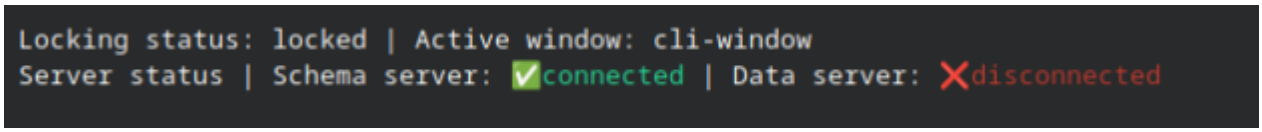

Figure 2.9.: Connection Status after Connection was successful or failed

2.2.7. Export Intent Configurations

With the command `generate` the intent is written to a YAML file, so it can be applied using the `kubectl apply -f <intent.yaml>` command. The file has the name *configset-20251218-190448.yaml* the first part "configset" a static name followed by a timestamp, to distinguish the generated intents.

```
alpine-frontend:/app/sdcio-cli/sdcio-cli/src# cat configset-20251218-190448.yaml
apiVersion: config.sdcio.dev/v1alpha1
kind: ConfigSet
metadata:
    name: intent1
    namespace: default
spec:
    target:
        targetSelector:
            matchLabels:
                sdcio.dev/region: us-east
    priority: 10
    config:
        - path: /
          value:
            interface:
                ethernet-1/1:
                    description: test
```

Figure 2.10.: Output of a new generated intent

3. Conclusion and Prospect

This chapter reflects the conclusion as well as the possible outlook for the future.

3.1. Conclusion

The project covered all the mandatory use cases, although some were only partially implemented. Optional use cases could not be realized due to the tight scheduling constraints and the fact that not all mandatory use cases were implemented flawlessly. Certain functional aspects still require improvements. In particular, *UC9: Add/Update intent* and *UC10: Generate CRD* remain incomplete, as the current configuration output is invalid. This limitation arises because the setting and generating concept currently supports only leafs and containers, without handling special cases such as leaf-lists or lists.

Importantly, all non-functional requirements were implemented successfully, the only exception being NFR9: TUI setup, which still has space for optimization. This SDC TUIis only a MVP and still needs further improvements.

Despite these limitations, the project establishes a solid foundation for further development. It demonstrates the feasibility of a CLI/TUI for SDC and provides the groundwork for extending functionality in future work, such as a bachelor thesis. With continued refinement, the tool has the potential to significantly improve usability and adoption of schema-driven networking.

3.1.1. Outcome Analysis

This section gives a review about the status of the individual use cases and non-functional requirements.

Table 3.1.: Outcome Analysis of Use Cases

Title	Status	Reason / Discussion
UC1: Show datastores	Implemented	Fully working
UC2: Select a datastore	Implemented	Fully working
UC3: Switch to schema view	Implemented	No issues
UC4: Read current node	Partially implemented	Not all fields render correctly
UC5: Navigate schema tree	Implemented	FFully working
UC6: Switch to blame view	Implemented	Fully working
UC7: Read blame	Implemented	Fully working
UC8: List intents	Implemented	Fully working
UC9: Add/Update intent	Partially implemented	Issues with layers and merging
UC10: Generate CRD	Partially implemented	Needs schema cleanup or renderer fix
UC11: Autocompletion	Partially implemented	Missing datastore completions
UC12 (optional): Commit changes	Not implemented	Feature skipped due to prioritization

UC13 (optional): Get apply kubectl command	Not implemented	Feature skipped due to prioritization
UC14 (optional): Get delete kubectl command	Not implemented	Feature skipped due to prioritization
UC15 (optional): Filter schema	Not implemented	Feature skipped due to prioritization
UC16 (optional): Create ConfigSet CRD	Not implemented	Out of project scope in MVP
UC17 (optional): CRD Validation	Not implemented	Feature skipped due to prioritization

Table 3.2.: Outcome analysis of non-functional requirements.

Title	Status	Reason / Discussion
NFR1: Interoperability	Implemented	Works reliably
NFR2: Maintainability	Implemented	Passed linting
NFR3: Result presentation	Implemented	Works for schema, blame, YAML
NFR4: Tab switching	Implemented	Completed with clear active tab markers
NFR5: gRPC loading time	Implemented	Passed consistently
NFR6: gRPC warnings	Implemented	Added only in last version
NFR7: Startup workflow	Implemented	ImplementedWorks but added after MVP1
NFR8: Schema response time	Implemented	Meets timing requirement
NFR9: TUI setup	Partially implemented	Setup explained but SDC prerequisites too heavy
NFR10: Command history	Implemented	Works reliably

3.2. Outlook

The following improvements address partially implemented use cases and non-functional requirements.

3.2.1. Improvements

As mentioned in the Conclusion section, some use cases and one non-functional requirement were only partially implemented.

UC4: Read current node

The schema view should clearly distinguish node types (container, list, leaf-list, leaf). This can be achieved by introducing visual markers, such as emojis, similar to the blame view. Inspiration may be drawn from Nokia's YANG model explorer.[10]

UC9: Add/Update intent

Current implementation supports only leafs and containers. A sustainable workflow must be designed to handle lists and leaf-lists. Additionally, the CLI should allow engineers to set intent metadata (e.g., name, namespace).

UC10: Generate CRD

Closely tied to UC9, this functionality must translate user-defined trees into valid CRDs, accounting for all node types.

UC11: Autocompletion

The current implementation is missing the autocompletion for possible datastores.

NFR9: TUI setup

Installation should be simplified through automation. While complexity partly stems from SDC's own setup, reducing friction at the TUI level remains essential.

3.2.2. Prospect

This section covers additional improvements the TUI could implement and aspects that could be addressed next.

Usability Enhancements – UC12 (optional): Commit changes

Would allow engineers to review configurations before deployment. UC13 (optional): Get apply kubectl command and UC14 (optional): Get delete kubectl command would streamline workflows by generating ready-to-use commands.

Efficiency in Navigation – UC15 (optional): Filter schema

Would reduce time spent navigating large schemas, improving productivity.

Scalability – UC16 (optional): Create ConfigSet CRD

Would enable configurations across multiple devices, supporting larger environments.

Validation and Safety – UC17 (optional): CRD Validation

Would leverage SDC's built-in validation to prevent faulty intents.

User Experience

TUI styling should be refined to improve readability and overall usability.

Part II.

Project Documentation

1. Requirements

This chapter covers the product requirements. The use cases are covered in chapter 1.1 and the non-functional requirements in chapter 1.2.

1.1. Use Cases

1.1.1. Actors

For our use cases we decided to only have the actor, which in the end is a **Network Engineer**.

- **Network Engineer**: A **network engineer** prefers working with CLIs, to configure their infrastructure, over writing files by hand.

1.1.2. Use Cases

In total, we defined 17 use cases, where five of them are optional. The priority of the use cases is set by their order.

The following diagram shows the relationships between the use cases.
The blue use cases are mandatory, while the orange ones are optional. Additionally, the mandatory use cases have a solid border, while the optional ones have a dashed border.
The solid arrows represent the relationships of the mandatory and the dashed arrows the optional use cases.

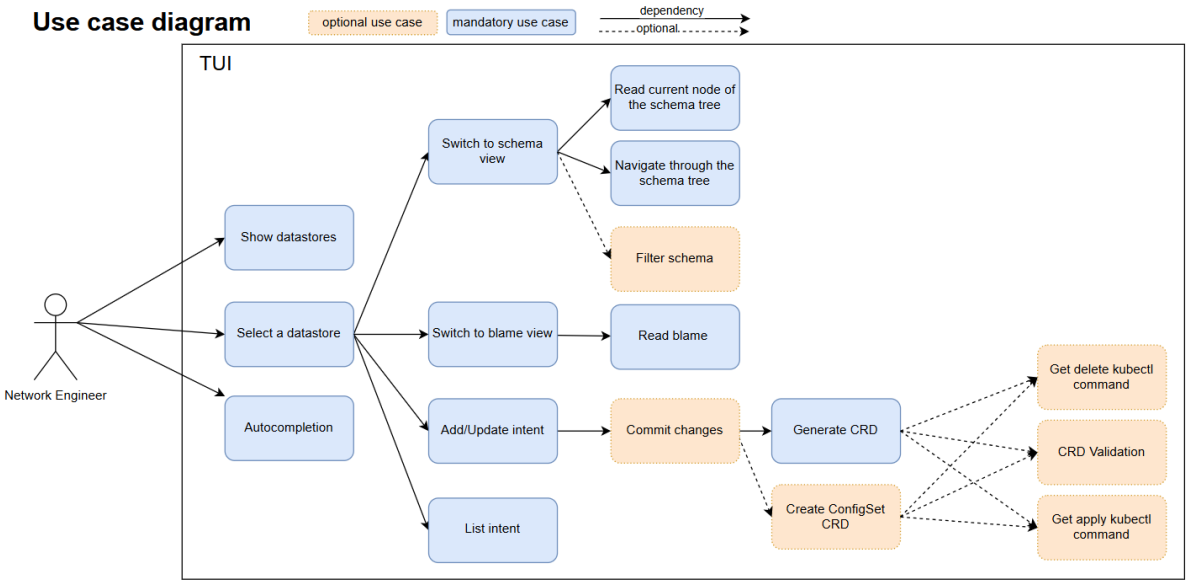

Figure 1.1.: Use case diagram.

UC1: Show datastores	
Actor	Network Engineer
Goal	The available datastores are shown with their schema information (vendor and version).
Precondition	Connection to SDC schema and data server.
Use Case	The network engineer uses the TUI to display all datastores and their schema information (vendor and version).
Acceptance Criteria	When the network engineer uses the TUI command to list datastores, all available datastores with their schema information are displayed.

Table 1.1.: UC1: Show datastores

UC2: Select a datastore	
Actor	Network Engineer
Goal	The network engineer can select a datastore to work with and the schema information (name, vendor and version) will be displayed while working with it.
Precondition	Connection to SDC schema and data server.
Use Case	The network engineer uses the TUI to choose a datastore to work with.
Acceptance Criteria	When the network engineer selects a datastore, the TUI displays the selected datastore's schema information (name, vendor and version).

Table 1.2.: UC2: Select a datastore

UC3: Switch to schema view	
Actor	Network Engineer
Goal	The network engineer is able to switch to the schema window.
Precondition	*UC2: Select a datastore*, see table 1.1.2.
Use Case	The network engineer can use the TUI to switch to the schema view via keystrokes.
Acceptance Criteria	The network engineer is able to switch to the schema view.

Table 1.3.: UC3: Switch to schema view

UC4: Read current node of the schema tree	
Actor	Network Engineer
Goal	Read the current node of the schema tree and display its content.
Precondition	*UC3: Switch to schema view*, see table 1.1.2.
Use Case	The network engineer can read a current node of the schema tree, to see possible configuration options.
Acceptance Criteria	The network engineer is able to read the content of the current node of the schema tree.

Table 1.4.: UC4: Read current node of the schema tree

UC5: Navigate through the schema tree	
Actor	Network Engineer
Goal	Navigate through the different nodes of the schema tree.
Precondition	*UC3: Switch to schema view*, see table 1.1.2.
Use Case	The network engineer can use the TUI to navigate through the different nodes of the schema tree, to see possible configuration options.
Acceptance Criteria	The network engineer is able to switch to the schema view, focuses that window and navigates through the schema tree using keystrokes.

Table 1.5.: UC5: Navigate through the schema tree

UC6: Switch to blame view	
Actor	Network Engineer
Goal	The network engineer is able to switch to the blame window. The blame view shows the running configuration in blame format.
Precondition	*UC2: Select a datastore*, see table 1.1.2.
Use Case	The network engineer can use the TUI to navigate to the blame view via keystrokes.
Acceptance Criteria	The network engineer is able to switches to the blame view.

Table 1.6.: UC6: Switch to blame view

UC7: Read blame	
Actor	Network Engineer
Goal	Get the running configuration with config blame.
Precondition	*UC6: Switch to blame view*, see table 1.1.2.
Use Case	The network engineer uses the TUI and gets the running configuration displayed in blame format.
Acceptance Criteria	The network engineer is able to switch to the blame view, focuses that window and uses keystrockes to navigate and read the running configuration in blame format.

Table 1.7.: UC7: Read blame

UC8: List intent	
Actor	Network Engineer
Goal	Display the available intents for configuration.
Precondition	*UC2: Select a datastore*, see table 1.1.2.
Use Case	The network engineer uses the TUI to get a list of available intents.
Acceptance Criteria	The network engineer used the TUI command to list intents and all available intents are displayed.

Table 1.8.: UC8: List intent

UC9: Add/Update intent	
Actor	Network Engineer
Goal	Add or update an intent.
Precondition	*UC2: Select a datastore*, see table 1.1.2.
Use Case	The network engineer uses the TUI to update or add an intent, so it can later be deployed.
Acceptance Criteria	The network engineer uses the TUI command to add or update an intent

Table 1.9.: UC9: Add/Update intent

UC12: (optional) Commit changes	
Actor	Network Engineer
Goal	Commit changes to have them stored or discarded.
Precondition	*UC9: Add/Update intent*, see table 1.1.2.
Use Case	The network engineer can use the commit command to display their changes to then save or discard them.
Acceptance Criteria	The network engineer has made changes to the configuration and now can use the commit command to either save or discard them.

Table 1.10.: UC12: (optional) Commit changes

UC10: Generate CRD	
Actor	Network Engineer
Goal	The network engineer can generate a CRD out of the commited changes.
Precondition	*UC2: Select a datastore*, see table 1.1.2. Or if implemented *UC12: (optional) Commit changes*, see table 1.1.2.
Use Case	The network engineer can generate a CRD out of their commited changes, by using an apply/g.
Acceptance Criteria	The network engineer has finished their configuration and now can generate a CRD out of it.

Table 1.11.: UC10: Generate CRD

UC11: Autocompletion	
Actor	Network Engineer
Goal	Get suggestions for completing commands or parameters.
Precondition	Access to SDC schema and data server information.
Use Case	The network engineer uses the TUI and starts typing a command or parameter. The TUI provides suggestions for completing the input.
Acceptance Criteria	The network engineer starts typing a command or parameter, clicks tab and the command gets autocompleted.

Table 1.12.: UC11: Autocompletion

UC13: (optional) Get apply kubectl command	
Actor	Network Engineer
Goal	Get the correct kubectl command to apply the configuration.
Precondition	*UC10: Generate CRD*, see table 1.1.2.
Use Case	The network engineer gets a kubectl command from the TUI with which the network engineer can apply the configuration.
Acceptance Criteria	The network engineer has finished their configuration and now can run an apply/g to get the kubectl command.

Table 1.13.: UC13: (optional) Get apply kubectl command

UC14: (optional) Get delete kubectl command	
Actor	Network Engineer
Goal	Get the correct kubectl command to delete the configuration.
Precondition	*UC10: Generate CRD*, see table 1.1.2.
Use Case	The network engineer gets a kubectl command from the TUI with which the network engineer can delete the configuration.
Acceptance Criteria	The network engineer has finished their configuration and now can run a delete/g to get the kubectl command.

Table 1.14.: UC14: (optional) Get delete kubectl command

UC15: (optional) Filter schema	
Actor	Network Engineer
Goal	The network engineer can filter the output of the schema tree, to effitiently retrieve their desired information.
Precondition	*UC3: Switch to schema view*, see table 1.1.2.
Use Case	The network engineer can run commands like include or grep, in the schema view, to search for desired schema information.
Acceptance Criteria	The network engineer switches to the schema view, focuses that window and uses keystrockes to filter the schema tree output.

Table 1.15.: UC15: (optional) Filter schema

UC16: (optional) Create ConfigSet CRD	
Actor	Network Engineer
Goal	The network engineer can generate a ConfigSet with configurations for multiple devices.
Precondition	All mandatory use cases must have been achieved.
Use Case	The network engineer can use the TUI to generate a ConfigSet CRD which stores configuratoin for multiple devices.
Acceptance Criteria	The network engineer has finished their configuration for multiple devices and now can generate a ConfigSet CRD.

Table 1.16.: UC16: (optional) Create ConfigSet CRD

UC17: (optional) CRD Validation	
Actor	Network Engineer
Goal	After completing the configuration, the network engineer can validate their configuration.
Precondition	*UC10: Generate CRD*, see table 1.1.2.
Use Case	The network engineer has finished their configuration and now can run a validation to check for errors.
Acceptance Criteria	The network engineer uses the TUI command to validate the generated CRD and receives feedback about possible errors.

Table 1.17.: UC17: (optional) CRD Validation

1.2. Non-Functional Requirements

Listed below are the non-functional requirements for the TUI. The current status of the requirements and the testing can be found on YouTrack under the linked workitems. The requirements are not ordered.

NFR1: Interoperability	
Requirement	The TUI must be able to communicate with SDC endpoints, using gRPC for the schema server and the data server.
Verification	The app will be tested for its ability to communicate with the endpoints.
Time of Verification	Verification takes during development.
Acceptance Criteria	To be accepted, the TUI must successfully exchange data with SDC without errors.

Table 1.18.: NFR1: Interoperability

NFR2: Maintainability	
Requirement	The TUI should be easy to maintain, with clear code quality and naming conventions.
Verification	Takes place using a linter for Golang.
Time of Verification	Verification takes place before every pull request into the development branch.
Acceptance Criteria	To be accepted, the code must pass the linting pipeline.

Table 1.19.: NFR2: Maintainability

NFR3: TUI result presentation	
Requirement	If a schema, running configuration or YAML file is too big to be displayed in the TUI, then a scrollbar or arrow keys can be used to expose the overflowing content.
Verification	Print a schema that is too large to be displayed and check if it is scrollable, repeat the same for the running configuration and YAML file display. Checked by project team.
Time of Verification	After reaching each milestone.
Acceptance Criteria	Must be able to scroll through the content using arrow keys or a scrol wheel.

Table 1.20.: NFR3: TUI result presentation

NFR4: Tab switching and window locking	
Requirement	The network engineer should be able to switch tabs in another and understand on which window the network engineer is currently working on.
Verification	Manually test the switching of the window. Have the industrial partner and the advisor test it.
Time of Verification	When the first MVP is released.
Acceptance Criteria	A tab must be switchable using 3 keystrokes. The active tab must be thicker and an other color.

Table 1.21.: NFR4: Tab switching and window locking

NFR5: gRPC loading time	
Requirement	At the startup of the TUI, gRPC should not take longer than 5 seconds to connect with the SDC data and schema server.
Verification	During the usability tests the testees will start the TUI and measure the loading time.
Time of Verification	After the first MVP is released.
Acceptance Criteria	If it takes longer than 5 seconds during the startup, it fails.

Table 1.22.: NFR5: gRPC loading time

NFR6: gRPC warnings and errors	
Requirement	If the gRPC endpoints are not reachable, the TUI should display an error message or warning, to inform the network engineer about the problem. For usability these messages get displayed with an Emoji.
Verification	Make the TUI call the gRPC endpoints over a wrong endpoint.
Time of Verification	After the first MVP is released.
Acceptance Criteria	The error message or warning must be displayed if the gRPC server is not reachable and listing which component is not reachable.

Table 1.23.: NFR6: gRPC warnings and errors

NFR7: TUI workflow at startup	
Requirement	At the first startup the network engineer should only be able to work with schema and running configuration tabs after a datastore is selected, otherwise they should be locked.
Verification	Test the TUI at startup if the tabs are locked until a datastore is selected. The team members will test this.
Time of Verification	After the first MVP is released.
Acceptance Criteria	Using a checklist with yes or no if the screen is locked. A yes from all testees is needed.

Table 1.24.: NFR7: TUI workflow at startup

NFR8: Schema query response time	
Requirement	When sending a schema query, the TUI should receive a response within 2 seconds.
Verification	During usability tests, the testees will send schema queries and measure the response time.
Time of Verification	After the first MVP is released.
Acceptance Criteria	If the query takes longer than 2 seconds to respond, it fails.

Table 1.25.: NFR8: Schema query response time

NFR9: TUI setup	
Requirement	The TUI setup should be self explanatory without needing much knowledge of the technology behind it.
Verification	Have the industrial partners and advisor setup the TUI.
Time of Verification	After the first MVP is released.
Acceptance Criteria	The setup should not take more than 3 terminal commands.

Table 1.26.: NFR9: TUI setup

NFR10: Command history	
Requirement	The TUI should display the output and input of the last 5 commands for usability.
Verification	Each team member will enter test commands and check if the past input and outputs are stored.
Time of Verification	After reaching a milestone
Acceptance Criteria	The TUI must display the last 5 commands (input and output).

Table 1.27.: NFR10: Command history

2. Domain Analysis

This chapter presents the domain analysis of the project, providing an overview of the key entities, and relationships within the problem space. It begins with a domain model that illustrates the main components relevant to the system and their interactions, helping to establish a common understanding of the terminology and structure used throughout the project. The chapter then reviews the related work, highlighting how it builds upon or differs from prior approaches.

2.1. Domain Model

Figure 2.1 provides an overview of the problem domain and the main components involved in SDC-based configuration management. The domain model illustrates the relationships between the SDC data-server, schema-server, underlying communication mechanisms, and the technologies used to represent and manage configuration data.

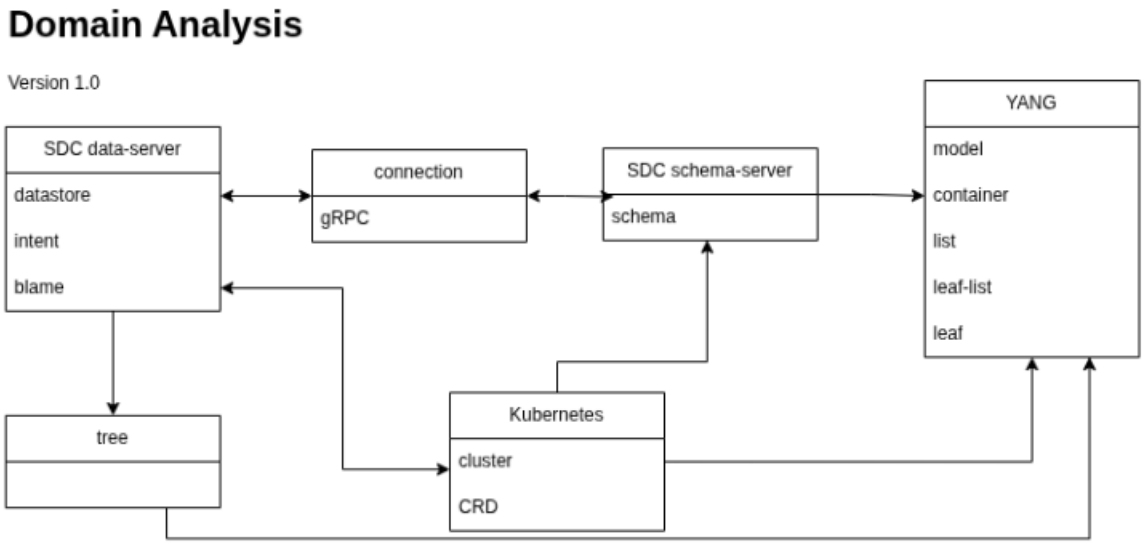

Figure 2.1.: Overview of the problem domain

2.2. Domain Model

At the core of the domain is the **SDC data-server**, which is responsible for managing configuration data and intents. It contains several key elements, including *datastore*, *intent*, and *blame*. A datastore represents a target device and stores information such as the available intents, the schema vendor, and the schema version associated with that device. Intents describe the desired configuration state to be applied to the target devices. The blame component provides traceability by indicating which intent or configuration element is responsible for a specific configuration on a device. This information is represented in a hierarchical tree structure, which explains the

one-way relationship from the data-server to the tree component.

The **tree** component itself does not contain domain-specific elements but serves as a structural representation for hierarchical data. It is primarily used to visualize blame information in a tree format. Additionally, the tree has a one-way relationship to YANG, as YANG models are inherently hierarchical and structured as trees.

Communication between the different components is handled through connections based on gRPC. gRPC is a high-performance, open-source remote procedure call (gRPC) framework that enables efficient and strongly typed communication between distributed services. The connection component has a bidirectional relationship with both the data-server and the schema-server and is also used to communicate with external endpoints, such as network devices or APIs exposed by the data-server and schema-server.

The **SDC schema-server** is responsible for managing and providing access to schema information. Its primary element is the schema, which contains YANG-based data models. The schema-server retrieves YANG models from network devices and makes them available for validation and configuration purposes. It has a bidirectional relationship with the connection component and a one-way relationship to YANG, reflecting its role as a consumer and processor of YANG models.

YANG is a data modeling language used to model configuration and state data for network devices. YANG models define the structure and constraints of configuration data in a hierarchical manner. The domain model includes the main YANG building blocks: *model* represents the top-level definition of a YANG module *container* groups related configuration nodes *list* defines a collection of entries identified by keys *leaf-list* represents an unordered list of simple values. *leaf* defines a single configuration value. These elements form the basis for representing and validating configuration data throughout the system.

Finally, **Kubernetes** plays a central role in the deployment and integration of SDC components. It consists of a cluster, in which both the data-server and schema-server are deployed, explaining their bidirectional relationship with Kubernetes. CRDs (Custom Resource Definitions) are used by SDC to define datastores and create intents. These CRDs are structured according to YANG models, as they describe configuration data in a declarative and hierarchical manner. For this reason, Kubernetes has a one-way relationship to both the schema-server and YANG, highlighting its role in hosting and representing configuration structures rather than consuming them directly.

2.3. Related Work

This section provides an overview of previous work related to this term project. Since SDC is a relatively new technology and at the time still under developement, only a limited number of tools and approaches exist. Therefore, this section focuses exclusively on the SDC intent builder. The related work serves primarily as a source of inspiration and reference for the design and implementation of the MVP developed in this term project.

2.3.1. SDC intent builder

The SDC intent builder [**intentbuilder**] is proof of concept to interactively build configurations for SDC. This tool is written in Golang by Markus Vahlenkamp from the Nokia SDC Team. It provides dynamic autocompletion with information from schema- and data-server to help the user with

the configuration. It doesn't provide the ability to explore the schema or the running configuration separately from the autocompletion. This term project takes inspiration of the intent builder.

Figure 2.2.: SDC intent builder [**intentbuilder**]

3. Architecture and Design Specifications

This chapter provides a detailed overview of the architecture and design specifications of the TUI developed for interacting with the SDC. It presents the conceptual foundation, the structural decomposition, and the sequence of operations that underpin the system.

3.1. Architecture

The architecture section covers the different levels of the SDC TUIusing the C4 model approach, combining the component and code level together. Since the SDC TUIis not deeply nested.

3.1.1. System Context Level

The basic concept is a TUI that runs in the context of a Kubernetes cluster, within the same namespace, in which also the SDC is deployed, so it can communicate with the schema- and data-server from SDC. This TUI should be able to assist the user in the process of writing configurations for SDC by providing information and autocompletion.

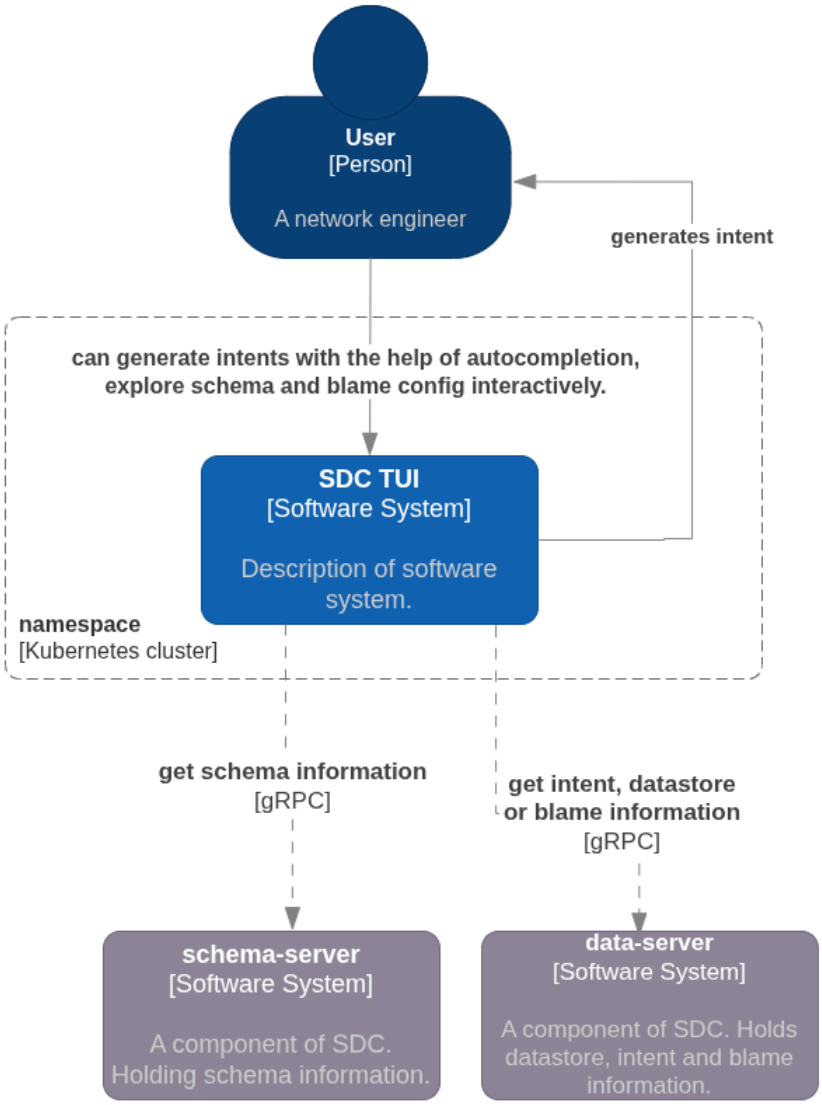

Figure 3.1.: System Context diagram for the SDC TUI system

3.1.2. Components Level

The components level diagram shows the structure of the TUI, more specific the different Golang packages: *main, tui, grpc_client, data, schema, autocomplete and intent*.

- *main* starts the *tui* package.
- *tui* contains the frontend and calls the functions in the other parts.
- *data* is the part for getting information about targets, intents and running configuration.
- *schema* contains functions for to getting the YANG schema information.
- *intent* has the functions to build the output and write it to the output file.
- *autocomplete* contains all functions for the autocompletion.

- *grpc_client* initiates connection to the data- and schema-servers.

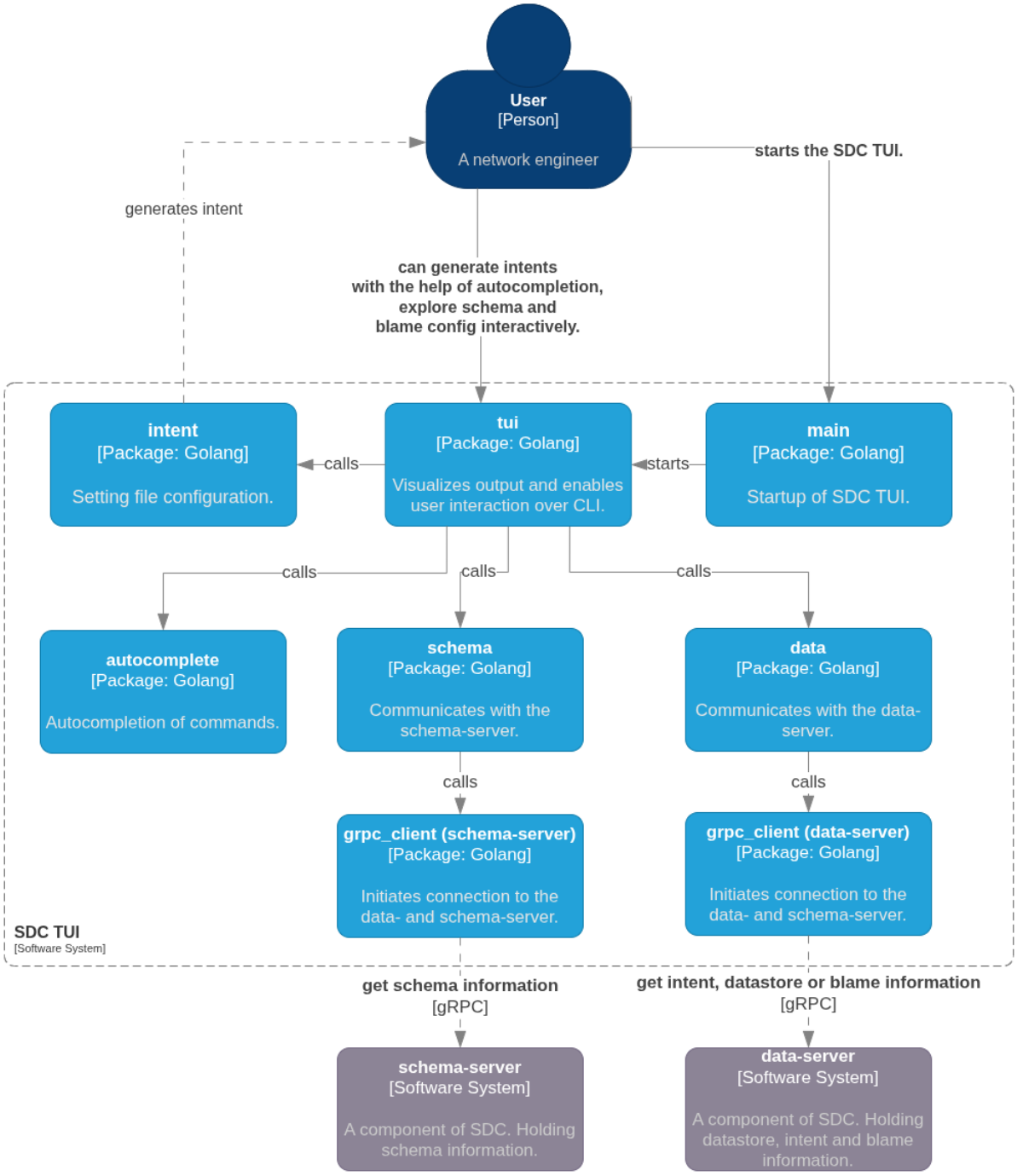

Figure 3.2.: Components level diagram, showcasing the different Golang packages

3.1.3. Component + Code Level

The packages: *main, tui and grpc_client* are excluded, since they don't provide add additional value considering the initial starting point.

Data
At startup the Data section provides the information about possible targets for the user to configure. When a target is selected it provides the information about the running configuration and the intents it is derived from.

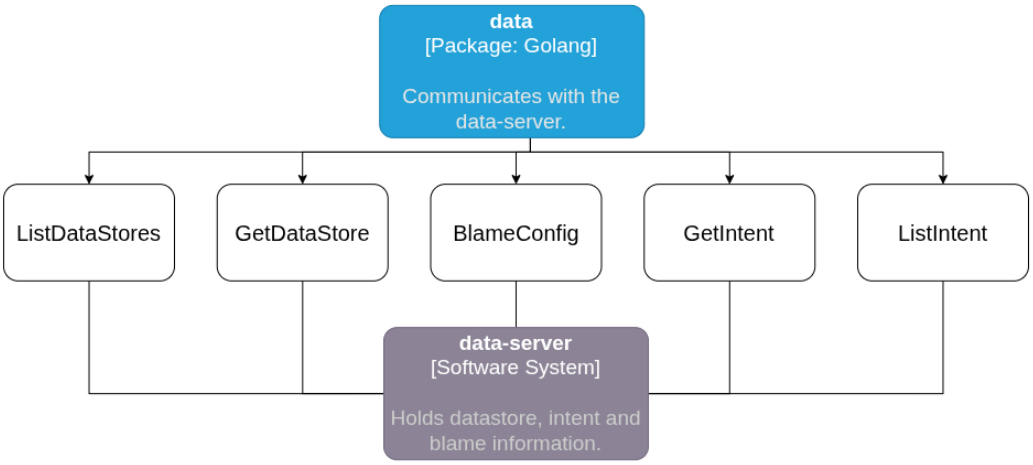

Figure 3.3.: Diagram of the data package

Schema
The Schema part gets the information from schema-server to display it to the user, so the user can explore possible configuration paths in the schema tab. For performance reasons at target selection only the first level is fetched the further levels are loaded once the user explores the schema.

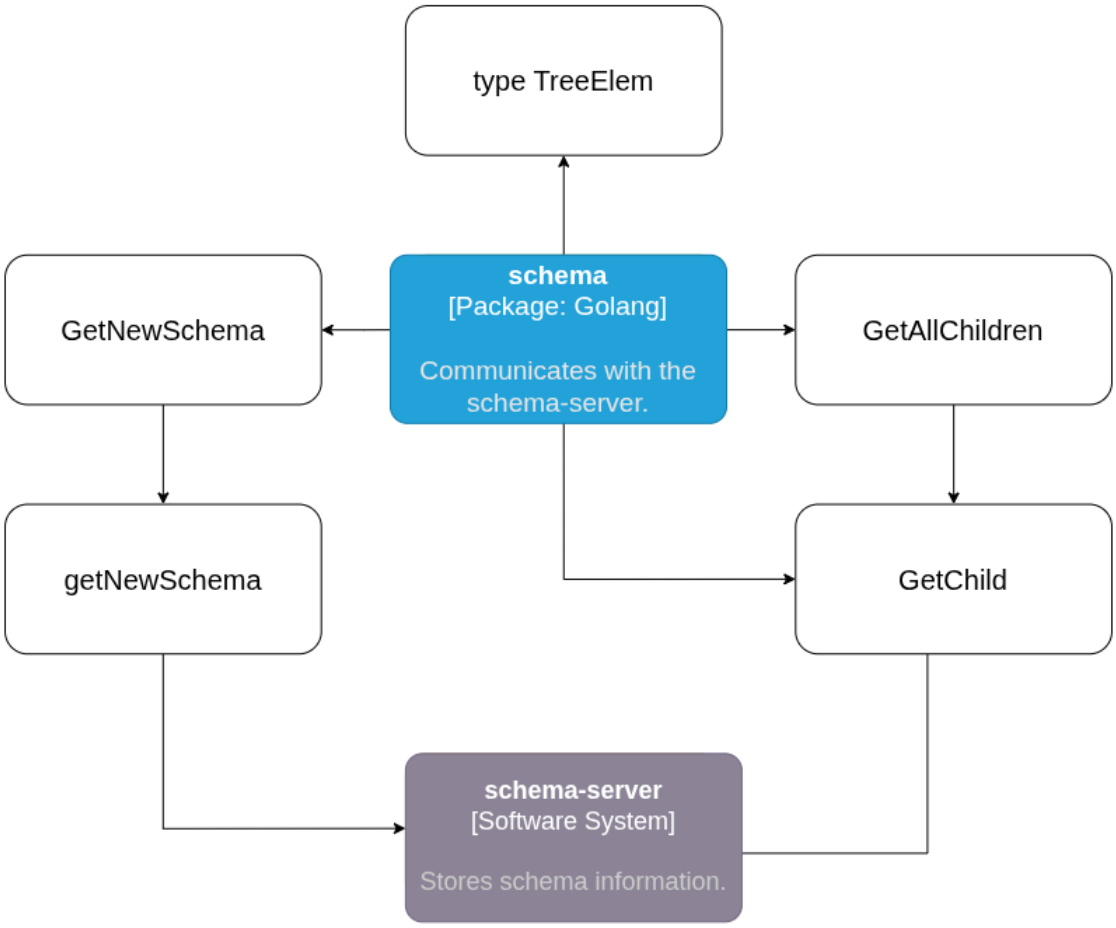

Figure 3.4.: Diagram of the schema package

Autocomplete
For the autocompletion we built a tree of possible commands. At startup a basic set of static commands and dynamic information about possible targets is used to build a basic command tree. Once a configuration target is selected further parts, of the dynamic autocompletion, is loaded. This includes the intents and the first level of schema information. Further levels are loaded on demand.

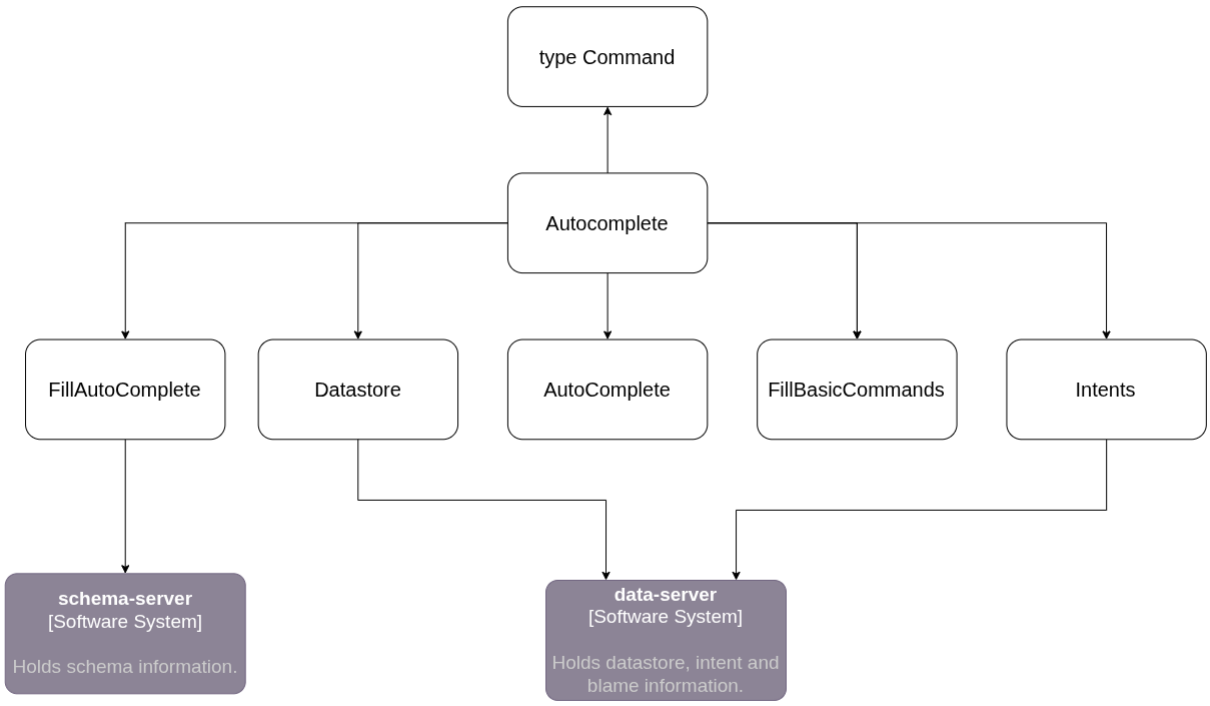

Figure 3.5.: Diagram of the autocomplete package

Intent
The intent section handles the output of the TUI by building a tree of the given configuration and recursively writing the tree nodes to the config. To achieve this, it uses a modified version of the "TreeElem" type from the "Schema" part, it replaces the "leaf" type from SDC used in the schema part with a custom "Leaf" struct. It also is responsible for loading an existing intent into the TUI, so it can be modified.
For this it retrieves the intent as a JSON-object from the data-server and then converts the response into a tree.

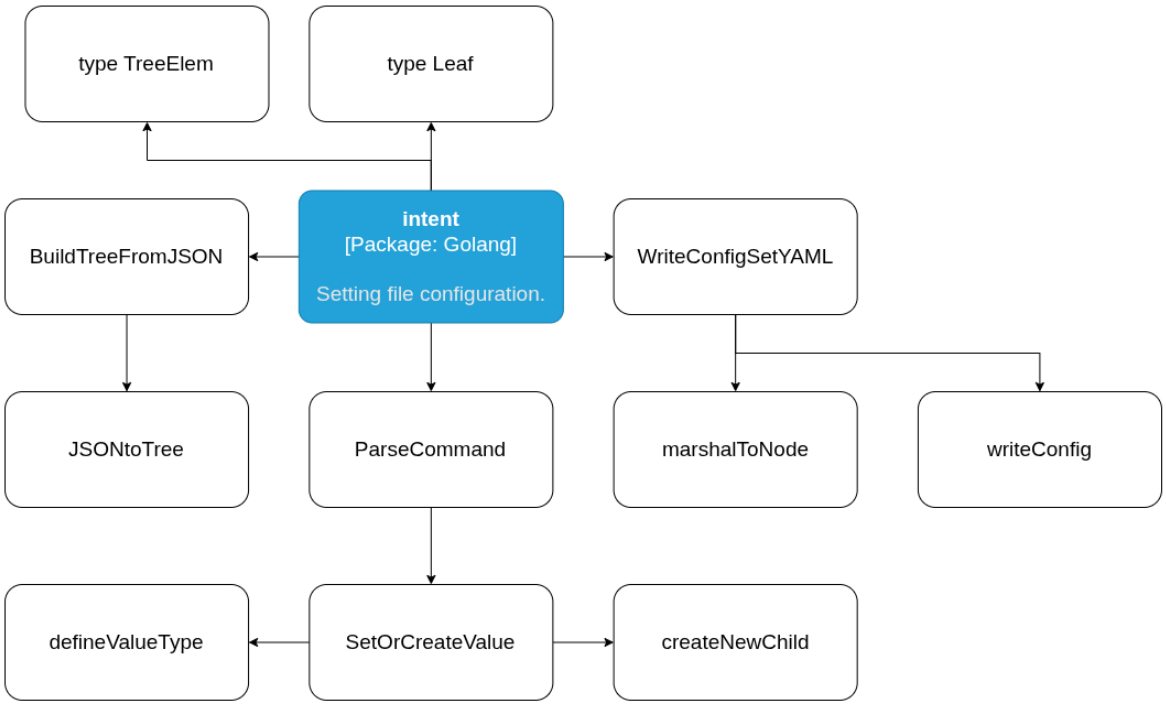

Figure 3.6.: Diagram of the intent package

3.2. Sequence Diagram

This chapter showcases a sequence diagram of the SDC TUIstartup process, when the milestone M3: Basic read functionality implemented was reached Due to the cost and maintenance of such a complex diagram, this section only covers the start sequence. That sequence diagram no longer being relevant to the MVP, but it adds a deeper understanding of the complexity and how tightly coupled the components are.

3.2.1. Start sequence

During startup the TUI fetches possible datastores from the SDC data-server and displays it as an option for the user to select. After the user selected a target the first two levels of schema information and config blame data are queried from the respective SDC endpoints. This Information is then displayed to the user.

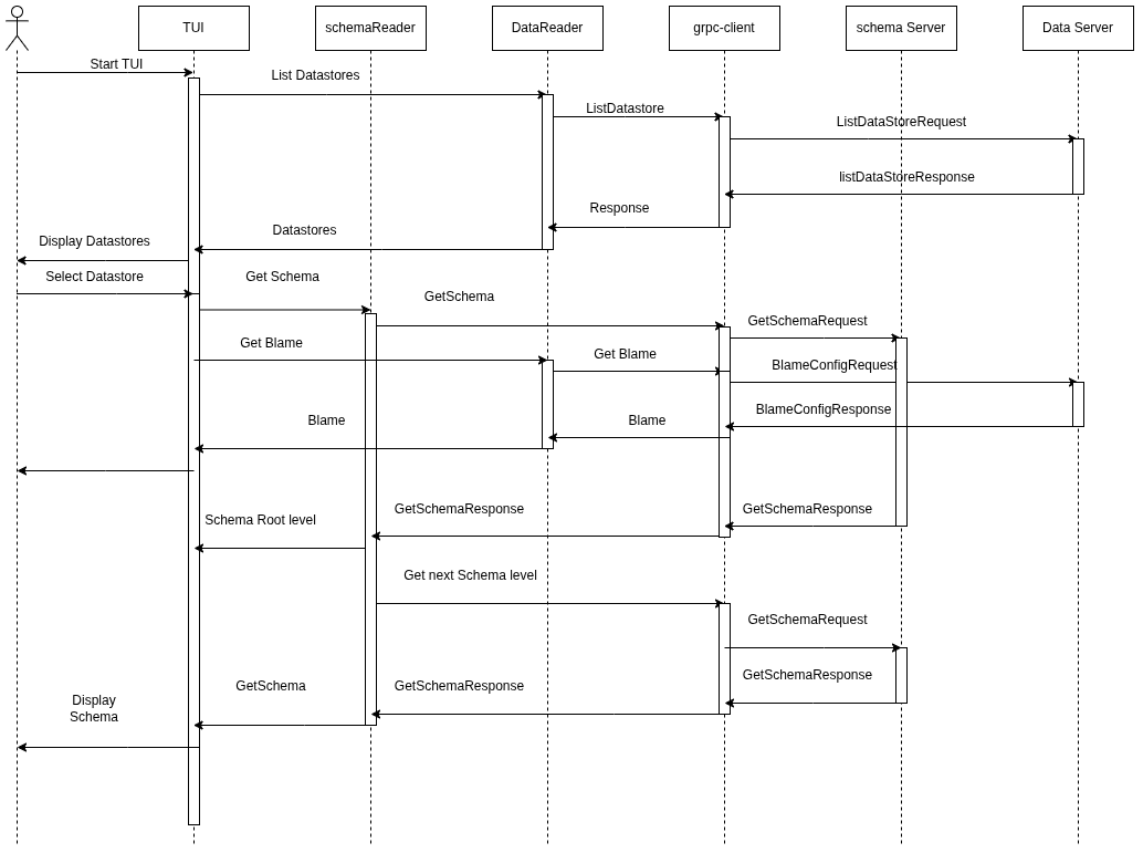

Figure 3.7.: Start up sequence diagram from when M3: Basic read functionality implemented was reached

3.3. Design Decisions

This section covers the languages, frameworks, libraries and tools which are used in this project as well as why they were chosen. Aside from that, general design decisions are also examined.
General Tools:

General:		
Language	**Usage**	**Reasoning**
Latex[7]	Language used for the documentation.	We already worked with it in a previous project.
Golang[5]	The programming language both frontend and the backend share.	SDC is written in Golang therefore it enhances the compatibility, unification and maintainability after the bachelor's thesis.

Table 3.1.: Design decisions for the general tools.

TUI instead of a CLI:
The first decision we had to confront ourselves with, was which format our product should adapt to.

We have the CLI which has the advantage to be simple and more comfortable to users preferring to work in a simple terminal only.
And on the other hand we have a TUI which is graphically more pleasant to look at and is more interactive than a CLI.

Since the product is not only there for configuring intents but also to bring the concept of YANG closer to the user, we decided to go for a TUI. But in general our product is two in one. There is a CLI inside the TUI, giving the user the best of both worlds.
We benefit from the TUI layout which enables different views, with one section being reserved for the CLI, keeping the familiar terminal feeling on the left side of the TUI. The right side shows the schema tree view of the chosen datastore and the blame view. More about the layout of the product can be found in the section 4.5.

Repository Structure:
We decided to keep documentation and code separate. The code being stored in GitHub, where the documentation lays in the GitLab.
GitHub was chosen for the code in lookout for the future to publish our product as an open source solution.
GitLab was chosen because it is the internal standard of the university.

Development Process:
In the first stages of our prototype we realized, that not all needed information was exposed to the outside of the cluster. Parts of the data-server were exposed, where the schema information was only accessible through the SDC API endpoint. And since these endpoints were not exposed to the outside, we had to think of a solution for our product. We had the following options:

- Option Ingress: Add an ingress to the cluster, which makes the SDC endpoints accessible and exposed to the outside.
- Option Pod: Create a Kubernetes pod inside the cluster, where the TUI can run.

Our decision fell on the Pod option, since it is more secure and less complex to set up. The Ingress option feels more casual to work with in our testing environment since it is in our local environment and not inside a cluster.

But setting up the ingress would be time-consuming and also lead to a more difficult installation for end users, if they would have to create an ingress themselves. This would not be productive. Another argument is the security aspect, since the Ingress option exposes the endpoints to the outside. Not every user wants to expose this information to the outside and their working environment could be inside the cluster.

These arguments led to our decision to have the product run inside the same cluster as SDC is running in.

3.3.1. Frontend

For the frontend we decided to use multiple frameworks from the Charm repository, because it is commonly used for building TUIs in Golang and is well documented.
A more detailed insight into the TUI design can be found in section 4.5 UI Design.

Frontend Frameworks and Libraries:	
Bubbletea [3]	TUI framework for Golang.
Bubbles [2]	Used for the paginator to display scrolling process.
Lip Gloss [9]	For the TUI layout and coloring.

Table 3.2.: Frontend Frameworks and Libraries.

3.3.2. Backend

A repeating format in our concept was the tree model.
We needed a tree for the schema view, configuration and generation component.
Since these components all needed a parent-child like relationship model.

Backend Frameworks and Libraries:	
gRPC	To initiate a connection to the SDC API endpoints.
Protobuff	SDC already uses Protobuff for their structured data. To access the different entities we used the provided protocol buffers from SDC.
YAML	Needed so that the configuration can be translated into a YAML file.

Table 3.3.: Backend Frameworks and Libraries.

4. Implementation

All source code for this project is available in the sdcio-cli GitHub repository, with the implementation located in the `src` directory. This chapter provides an overview of the implementation of the SDC TUI. It begins by describing the general file structure and the sec:application-deployment process. The chapter then details the frontend and backend implementations. Finally, section sec:uidesign discusses the design decisions made for the TUI.

4.1. General

The following sections will cover the frontend and backend implementation. The sdcio-cli GitHub repository is not structured into separate frontend and backend folders, as shown in figure 4.1.

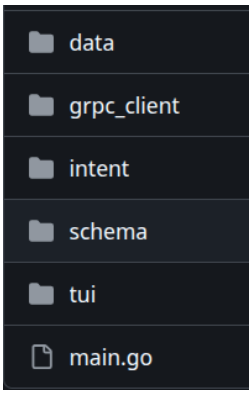

Figure 4.1.: Screenshot of the folder hierarchy of the https://github.com/andu3214/sdcio-cli/tree/main repository

To make adaptation easier for frontend or backend engineers, the folder hierarchy can be understood in terms of the following components: **Backend**

- `data`: retrieves information from the data server.
- `grpc_client`: responsible for creating a connection to the SDC servers.
- `intent`: contains files related to intent creation, update, and generation.
- `schema`: retrieves information from the schema-server.

Frontend

- `tui`: holds all frontend components.

4.2. Application Deployment

The installation process of the SDC TUIis described in the README.md file in the sdcio-cli GitHub.

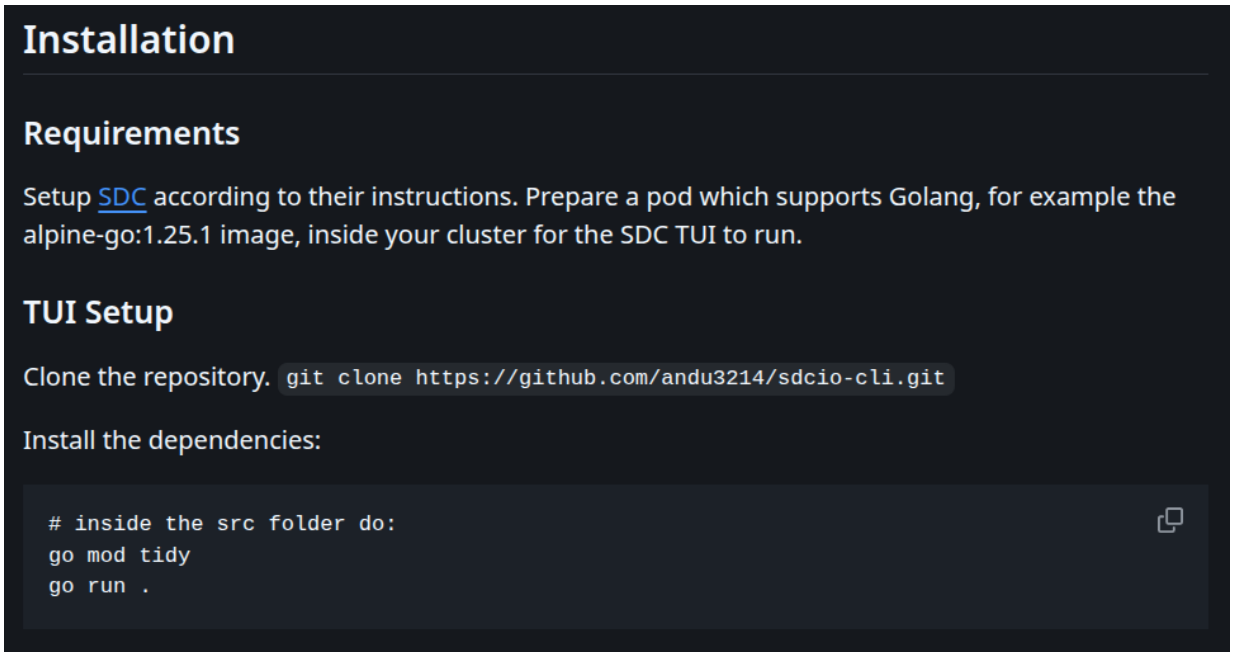

Figure 4.2.: Screenshot of the installation instructions from the README.md file in sdcio-cli GitHub

4.3. Backend

This section covers the most important backend components implementation. Giving a short overview of each folders content and going more into detail if a component is complex.

4.3.1. Data

The data section only consists of the file `data.go` which holds all function needed to retrieve information from the data-server. These functions are pretty much self-explanatory calling the appropriate function and return the response or error.

4.3.2. Grpc_client

This folder contains two files `grpc_client.go` and `grpc_status.go`.
The first file holds the constants `SCHEMA_SERVER` and `DATA_SERVER`. In the MVP these constants are hardcoded (`"data-server:56000"`) with the link to the server location. There are two functions one to open a connection to the data-server and the other does the same for the schema-server.

```
1 func GetgrpcSchemaClient() (sdcpb.SchemaServerClient, *grpc.ClientConn) {
2   schemaServerAddr := SCHEMA\_SERVER
3   conn, err := grpc.NewClient(schemaServerAddr,
      grpc.WithTransportCredentials(insecure.NewCredentials()))
4   if err != nil {
5     panic(err)
6   }
7
8   schemaClient := sdcpb.NewSchemaServerClient(conn)
9   return schemaClient, conn
```

```
}
```

The `grpc_status.go` is responsible for the status located in the SDC TUIfooter. Similar to the `grpc_client.go` it also consists of two functions to check the data- and schema-server, if they are still reachable. The `TIMEOUT` constants set the limit for how many seconds it should wait for a response.

```
const TIMEOUT = 5

// CheckSchemaStatus pings the schema server and returns whether it's reachable.
func CheckSchemaStatus() error {
  client, conn := GetgrpcSchemaClient()
  defer conn.Close()

  // if server does not respond within TIMEOUT seconds it will give an error
  ctx, cancel := context.WithTimeout(context.Background(), TIMEOUT*time.Second)
  defer cancel()

  _, err := client.ListSchema(ctx, &sdcpb.ListSchemaRequest{})
  return err
}
```

As visible in the code snippet above. The current timeout limit is set to 5 seconds to fulfill the tab:nfrgrpcload.

4.3.3. Intent

The intent folder holds four files:

- **generate_intent.go** writes set configuration into a YAML file.
- **set_intent.go** receives the input command from the frontend and then modifies the tre holding the configuration, adjusting it according to the received command.
- **tree.go** holds the tree structure.
- **update_intent.go** responsible to translate an received intent, which comes in JSON format and translate it into a tree format and sets it as the current active tree.

4.3.3.1. YAML intent generation

The YAML generation is implemented in the file `generate_intent.go`.
The function WriteConfigSetYAML constructs a complete ConfigSet YAML document using `yaml.Node` in order to enforce a deterministic key order.
Using `yaml.Node` allows explicit control over key ordering, which is not guaranteed when marshaling Golang maps directly.
The root YAML fields (apiVersion, kind, metadata, and spec) are currently hardcoded. Inside spec, the fields target, priority, and config are added in a fixed order.

```
spec := map[string]interface{}{
    "target": map[string]interface{}{
      "targetSelector": map[string]interface{}{
        "matchLabels": map[string]string{
          "sdcio.dev/region": "us-east",
        },
```

```
      },
    },
    "priority": 10,
    "config": []map[string]interface{}{
      {
        "path":  "/",
        "value": writeConfig(treelement),
      },
    },
  }
```

The config field contains a single entry with a hardcoded path (`"/"`) and a dynamically generated value. This value is created by the recursive function `writeConfig`, which traverses the configuration tree.

For each tree element, all child nodes are added recursively as nested maps. Leaf values are added while preserving their Golang types (string, int, bool) so that they are correctly represented in YAML. Unsupported types are converted to strings.
LeafLists are added as YAML sequences and are currently populated with placeholder values, as user-defined list input is not yet supported.
The helper function `marshalToNode` converts Go values into `yaml.Node` structures by marshaling and unmarshaling them, allowing integration into the manually constructed YAML.
The logic is triggered by entering the `generate` command in the TUI. A timestamp is appended to the filename, after which the `WriteConfigSetYAML` function is executed.

File permissions (0644) The third argument of the `os.WriteFile` function specifies the file permission mode that is applied when the file is created. The value `0644` is an octal representation of Unix file permissions.
Each digit defines the access rights for a specific user group:

- **Owner**: read and write permissions (`6 = 4 + 2`)
- **Group**: read-only permission (`4`)
- **Others**: read-only permission (`4`)

In symbolic notation, this corresponds to `rw-r-r-`. The leading zero indicates that the value is interpreted as an octal number.
This permission setting allows the file owner to modify the file while permitting read access for all other users, which is appropriate for configuration files such as YAML documents.

4.3.3.2. Set Intent

The workflow of how the functions in the `set_intent.go` are called, are the following. First ParseCommad() receives the full `set` command from the frontend. Then it strips the `set` string from the rest of the input. What is left now is a tree traversal order and the last word is the value, which are returned, as `path` and `value`, back to the frontend.

The next function to be called it `SetOrCreateValue` which is responsible to place the value at the defined location. During this process it can call the helper functions `defineValueType` and `createNewChild`:

- `defineValueType` returns the value according to its type.
- `createNewChild` creates and returns a new tree element.

4.3.3.3. Tree

```
The TreeElem is the structure of the generated tree part of the file:
type TreeElem struct {
  Schema      *sdcpb.Schema
  Parent      *TreeElem
  Name        string
  Path        *sdcpb.Path
  ChildNames  []string
  Children    []*TreeElem
  Leafs       []Leaf
  LeafLists   []*sdcpb.LeafListSchema
  Description string
  Namespace   string
  Type        string
}
```

The difference to the TreeElem from the schema treeElement is the additional attribute `Leafs`, which is an Array of Leaf.

```

type Leaf struct {
  Name  string
  Value interface{} // Use interface{} to allow multiple data types
}
```

This different Leaf struct is needed, such that the actual Value of the Leaf can be saved. This is an example how the Leaf struct is used:

```

Leafs: []Leaf{
      {Name: "type", Value: "bridged"},
}
```

4.3.3.4. Update Intent

This part contains two functions:

- **BuildTreeFromJSON** is called by the frontend after the command `update <intent>` was called. This function receives the JSON response from the data-server of an intent. The function creates a new empty tree and calls the `JSONTOTree` function and in the end returns the new tree holding the configuration from a received intent.
- **JSONToTree** this function does the whole work, translating the received JSON response into a tree and returns it afterwards.

4.3.4. Schema

In this section the implementation of the Schema part is detailed, which only consists of the file `schema.go`. But consists of two parts: general schema-server calls and methods for the TreeElem implementation. The most important components will be discussed in the following sections.

4.3.4.1. TreeElem

The type **TreeElem** is the backbone of the schema implementation it contains all the information that are needed for the tree exploration.

```
type TreeElem struct {
    Schema      *sdcpb.Schema
    Parent      *TreeElem
    Name        string
    Path        *sdcpb.Path
    ChildNames  []string
    Children    []*TreeElem
    Leafs       []*sdcpb.LeafSchema
    LeafLists   []*sdcpb.LeafListSchema
    Description string
    Namespace   string
    Type        string
}
```

The TreeElem type represents the information we need from the YANG container. THe Children elements are YANG containers the list is filled during run time when the level is reached, to check if there are Children the Childnames list can be used. While schema, path, Leafs and LeafLists get there schema type from the sdc protobuf definitions. The Path element is the YANG path to the current with the child name we can build the path of the next container.

4.3.4.2. get Schema

The schema tree starts with the __root__ element, it is fetched as soon the datastore is selected, while sub-container are fetched later when needed. The needed parts for the schema tree are written in the TreeElem type.

4.3.4.3. GetChild and GetAllChildren

To complete the tree there are two methods GetChild gets the specific requested container and adds it to the list of children. GetAllChildren gets all children from the ChildNames list to get each container GetAllChildren uses GetChild. There were experiments with fetching the complete tree recursively, but that lead to freezing the Terminal User Interface (TUI).

4.4. Frontend

- **autocomplete** folder, holding the implementation of the autocompletion.
- **data_items.go** provides the connection between `input.go` and the backend `data/data.go` components. It also formats and displays the output returned from the backend.
- **help.go** stores the text for the help command output.
- **input.go** contains the `cliModel` struct, manages the input textarea, autocompletion, and execution of commands.
- **messages.go** defines all message types used to trigger events and update the TUI state.
- **schema_view.go** implements the interactive schema tree, allowing expansion, collapse, and navigation of schema nodes.

- **server_status.go** handles the periodic checking and display of backend server status.
- **start.go** holds the welcome text that gets displayed when starting the TUI.
- **styles.go** defines the visual styling of windows, headers, tabs, command lines, and outputs using `lipgloss`.
- **tui.go** is the core component that coordinates the layout, manages window focus, tabs, viewports, and delegates events to sub-models (CLI, schema, blame).

4.4.1. Autocomplete

This Section describes the implementation of the autocomplete function. Autocomplete is called once the user has locked the left window and presses the tab key.

4.4.1.1. Type Command

The command struct is used to save the names of possible input in a tree structure.

```
type Command struct {
    name         string
    childsPulled bool
    schemaElem   *schemaClient.TreeElem
    flags        []string
    subcommands  []*Command
    commandType  string
}
```

The autocompletion is based on a tree structure were a list of the base commands is the root level each command has a list of subcommands. The autocompletion also uses dynamic information from the YANG schema so it is not possible to fetch it all in one go. To indicate the state of the subcommands there is the variable childsPulled of type Bool that is false until the subcommands are filled.

4.4.1.2. Autocompletion

The Autocompletion method takes an input string array and a tree of commands as a parameter. If the command is not the last element of the array, the input word is searched in the tree. If found the command is added to the output. If a word is not found in the commands tree, then it is skipped and the next word from the input is searched on the same level. The last element of the input list is then expected to be incomplete, and it is searched for commands in this level of the tree that contain the part in the input.

4.4.1.3. FillBasicCommands

The method FillBasicCommands returns a list of static commands.

```
show
  |- datastores
  |- intents
blame
disable
enable
set
```

```
help
update
list
get
```

4.4.1.4. Datastores

The Datastores method fetches the available datastores from the data-server. This happens during the startup of the TUI. The names of the datastores are filled under the enable command.

```
show
  |- datastores
  |- intents
blame
disable
enable
  |- <datastore names>
set
help
update
list
get
```

4.4.1.5. Intents

The Intents method fetches the available intents from the data-server once a datastore is selected. The intent names are added to the tree under update and get commands.

```
show
  |- datastores
  |- intents
blame
disable
enable
  |- <datastore names>
set
  |- <sub commands>
help
update
  |- <intents>
list
get
  |- <intents>
```

4.4.1.6. FillCommands

Once the datastore is selected the TUI is able to load the YANG schema. For performance reasons the level __root__ + 2 is loaded, this is the first level that is relevant for the configuration. Further levels are loaded once they are reached by the function.

```
show
  |- datastores
  |- intents
blame
```

```
disable
enable
  |- <datastore names>
set
  |- <sub commands from schema>
help
update
list
get
```

4.4.2. Data Items

The implementation is pretty self-explanatory. What's special is the type `dataStoreStruct`:

```
type dataStoreStruct struct {
  name          string
  schemaVendor  string
  schemaVersion string
}
```

The response from the backend gets translated into its own object, so the variables are of type string and now data-server specific.

4.4.3. Help

Simply has the following content:

```
var helpText = `
` + InputStyle.Render("SDC TUI - Help") + `
This TUI allows you to configure intents,
display the blame configuration and navigate over the schema.

Navigation: The TUI uses a window locking mechanism to switch between the windows:
    Esc                          Unlock the current window
    Left/Right Arrow Keys or Tab   Move between windows
    Enter                        Lock focus on the currently selected windo

Commands:
    ` + UnderlineStyle.Render("disable") + `               Deactivate the currently
    active datastore.
    ` + UnderlineStyle.Render("enable <datastore>") + `    Activate the specified
    datastore for configuration.
    ` + UnderlineStyle.Render("generate") + `              Write the current
    configuration to a CRD file.
    ` + UnderlineStyle.Render("get <intent>") + `          Display the full contents
    of the specified intent.
    ` + UnderlineStyle.Render("help") + `                  Show this help message.
    ` + UnderlineStyle.Render("quit") + `                  Exit the TUI.
    ` + UnderlineStyle.Render("set <path> <value>") + `    Modify configuration at the
    given path, or create a new intent.
    ` + UnderlineStyle.Render("show datastores") + `       List all available
    datastores.
    ` + UnderlineStyle.Render("show intents") + `          List all available intents.
    ` + UnderlineStyle.Render("update <intent>") + `       Select an existing intent
    as basis for editing with 'set' command.
`
```

```
23
24 func TuiHelpResponse() string {
25   return helpText
26 }
```

Which is shown in the SDC TUIas the following figure:

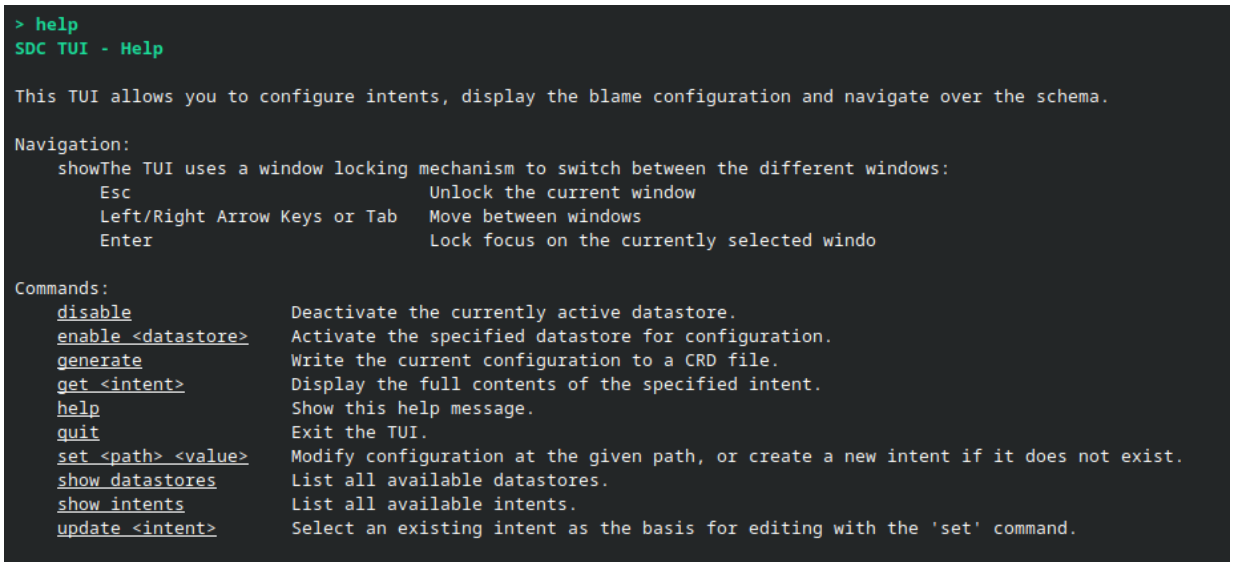

Figure 4.3.: Screenshot of the help command output in the SDC TUI

4.4.4. CLI Model

The `cliModel` struct represents the core state of the TUI. It contains the input textarea, the command tree, the current intent, and the autocompletion commands.

```
1 type cliModel struct {
2   input                textarea.Model
3   tree                 *writer.TreeElem
4   intent               *writer.TreeElem
5   autoCompleteCommands []*autocomplete.Command
6 }
```

Fields:

- `input`: manages the textarea input component from the TUI library.
- `tree`: holds the current command tree used for intent handling.
- `intent`: points to the currently selected intent in the tree.
- `autoCompleteCommands`: stores all available commands for autocompletion.

4.4.4.1. newCLIModel

The `newCLIModel` function initializes a new instance of the CLI model. It sets up the textarea input, focuses it, sets the width and height, and loads the list of autocompletion commands.

```
func newCLIModel(leftWidth int) cliModel {
  ta := textarea.New()
  ta.ShowLineNumbers = false
  ta.SetHeight(2)
  ta.SetWidth(leftWidth)
  ta.Focus()

  return cliModel{
    input:                ta,
    autoCompleteCommands: autocomplete.FillBasicCommands(),
    tree:                 writer.EmptyTree,
  }
}
```

4.4.4.2. Init

The `Init` method returns the initial command for the TUI application, which displays a welcome message.

```
func (c cliModel) Init() tea.Cmd {
  return startMessage()
}
```

4.4.4.3. Update

The CLI model includes several helper functions to handle command execution and output. Most return a `tea.Cmd` that produces an `OutputMsg`, optionally including errors or formatted results. Key examples:
invalidCommand: returns a message indicating that the entered command is not valid.

```
func invalidCommand(cmd string) tea.Cmd {
  return func() tea.Msg {
    command := InputStyle.Render("> " + cmd)
    invalid := InvalidCommandStyle.Render("x " + cmd + " is not a valid command")
    return OutputMsg{
      output: command + "\n" + invalid,
      err:    nil,
    }
  }
}
```

helpCommand: returns a command displaying the help response for a given input.

```
func helpCommand(help string) tea.Cmd {
  return func() tea.Msg {
    command := InputStyle.Render("> " + help)
    helpResponse := TuiHelpResponse()
    return OutputMsg{
      output: command + helpResponse,
      err:    nil,
    }
  }
}
```

Other helpers such as `runCommand`, `CommandOutput`, and `startMessage` exist to render input commands, display results or errors, and show the initial welcome message, but are not shown here for brevity.

4.4.5. Intent

This subsection covers the implementation of intent-related commands in the TUI. The TUI supports creating, updating, and generating intents through interactions with the backend and the command tree.

The main responsibilities include:

- Parsing input commands and updating the internal command tree.
- Writing configuration sets to YAML files with timestamped filenames.
- Retrieving and loading intents from the backend, handling errors appropriately.
- Returning formatted messages for successful or failed operations.

The implementation leverages helper functions to handle output formatting and error reporting, keeping the command logic concise and modular.

4.4.6. Messages

The TUI uses a set of *message types* to handle events and communicate between components in the application model. Each message type represents a specific event, such as user input, command output, or system status updates.

Concept: These message types are used in the TUI model to trigger specific behaviors when events occur, following the Bubble Tea [3] framework's message-driven architecture. For example:

- `InputSubmittedMsg` is sent when the user submits a command.
- `ClearInputMsg` triggers the input field to be reset.
- `AutocompleteMsg` carries the autocompleted input back to the TUI.
- `OutputMsg` contains the result of a command and optionally an error.
- `ErrorMsg` represents an error event, including a message and the associated error object.

These messages allow the TUI to react in a decoupled way: the model updates its state and triggers side effects based on the received message, while the view renders the changes.

```
type OutputMsg struct {
  output           string
  err              error
  activeDataStore  dataStoreStruct
  currentDataStore string
}

type ErrorMsg struct {
  err    error
  output string
}

type ClearInputMsg struct{}
```

Remarks: This design keeps the TUI reactive and modular: each message corresponds to a specific action or state update, making it easy to extend or modify behavior without tightly coupling components.

4.4.7. Schema View

The `Schema View` provides an interactive interface for traversing the schema tree in the TUI. It allows users to expand and collapse nodes, scroll through the tree, and visualize hierarchical data in a readable way.

4.4.7.1. schemaNode

Represents a single node in the local schema tree. Each node stores the underlying schema element, its children, depth in the tree, and whether it is currently expanded.

```
type schemaNode struct {
  Elem     *schema.TreeElem
  Expanded bool
  Children []*schemaNode
  Depth    int
}
```

Usage:

- `Elem` contains the actual schema data for this node.
- `Children` holds dynamically loaded child nodes when a node is expanded.
- `Expanded` tracks whether children are visible.
- `Depth` is used for indentation and rendering.

4.4.7.2. schemaView

Represents the interactive schema tree view in the TUI. It manages the tree of nodes, cursor position, the flattened list of visible nodes, the viewport, and the backend schema client.

```
type schemaView struct {
  tree         *schemaNode
  cursor       int
  flattened    []*schemaNode // used for rendering visible nodes
  schemaClient schema_server.SchemaServerClient
  viewport     viewport.Model
}
```

Usage:

- `tree` is the root of the schema tree.
- `cursor` tracks the currently selected node for navigation.
- `flattened` stores only the visible nodes for rendering efficiency.
- `viewport` handles scrolling and display within the TUI.
- `schemaClient` is used to fetch child nodes from the backend when expanding.

Concept: Together, these structs enable an interactive, scrollable, and expandable view of the schema tree. The TUI listens for keyboard input to move the cursor, expand or collapse nodes, and automatically updates the viewport to keep the selected node visible. This design maintains a reactive and modular structure while allowing efficient navigation of hierarchical data.

4.4.8. Start

The TUI displays a welcome message when the application starts, providing basic instructions for navigation and usage.

```
var welcomeText = `` + InputStyle.Render("Welcome to the SDC TUI!") + `

` + UnderlineStyle.Render("Navigation") + `
The TUI uses window locking to switch to the desired view:
* Esc, unlock current window
* Left/Right Arrow Keys or Tab, switch focus between windows
* Enter, lock focus on the currently selected window

Enable a datastore to begin with the configuration see
'show datastores' for available datastores.

See "help" for further information.
Type "quit" or press ctrl+c to exit.
`

func TuiWelcomeResponse() string {
  return welcomeText
}
```

Concept: The `TuiWelcomeResponse` function returns a formatted string that is shown to the user upon starting the TUI. The message includes:

- A greeting rendered using `InputStyle`.
- Navigation instructions for switching and locking windows.
- Guidance for enabling datastores and accessing help.
- Exit instructions for quitting the application.

This design ensures that users see a clear and informative introduction, helping them understand how to interact with the TUI immediately.

4.4.9. Styles

The TUI leverages the library to style and format the terminal interface. Styles are used to visually distinguish active/inactive windows, headers, command lines, output, and other UI elements. They define colors, borders, padding, alignment, and text formatting. The code is sorted according to the SDC TUIinterface order.
Concept: Rather than styling each element individually in the code, pre-defined style variables are used consistently throughout the TUI. This approach allows easy modification of colors or layout globally and keeps the rendering logic clean.
Key Examples:

- `InputStyle`: Styles the command line input with bright green text and bold formatting.

- `InvalidCommandStyle`: Styles invalid commands in red to indicate errors.
- `OutputWindowStyle / OutputWindowActiveStyle`: Define the borders, padding, and foreground color of the main output window. Active windows are highlighted with a brighter border color.
- `HeaderActiveStyle / HeaderInactiveStyle`: Used for the header of a window, with different border colors indicating active or inactive state.
- `RightWindowStyle / RightWindowActiveStyle`: Styles the right-hand side window (e.g., schema or status panel), similar to the main output window but with a distinct color scheme.
- `Tabs (Tab / ActiveTab / TabGap)`: Define the appearance of tabs including borders and active/inactive states.
- `FooterStyle / StatusStyle`: Styles informational text at the bottom of the screen such as status messages or statistics.

Behavior: These styles are combined with TUI components to render content dynamically:

- Borders visually group components (windows, tabs, and panels).
- Colors highlight active elements, errors, or status.
- Padding and alignment ensure consistent spacing and layout.

Example: The active output window highlights the border in bright blue:

```
OutputWindowActiveStyle = lipgloss.NewStyle().
    Border(OutputBorderActive).
    BorderForeground(ActiveColorBlue).
    Padding(1, 2)
```

Overall, the style system makes the TUI visually structured and enhances user experience by clearly distinguishing interactive areas, outputs, and status information.

4.4.10. TUI

The `model` struct is the central part of the TUI. It is instantiated in `main.go` and is responsible for managing the overall layout, window focus, tabs, and delegating events to the sub-models (CLI, schema, and blame views).

```
type model struct {
  cli              cliModel
  output           []byte
  width            int
  height           int
  ready            bool
  focusedContainer container
  activeRightTab   int
  rightViewport    viewport.Model
  leftViewport     viewport.Model
  previousContent  string
  currentDataStore string
  headerInfo       string
  blameOutput      string
  activeDataStore  dataStoreStruct
```

```
  windowLocked     bool
  tabWindowLocked  bool
  activeTab        int
  schemaView       schemaView
  schemaClient     schema_server.SchemaServerClient
  schemaServerStatus string
  dataServerStatus   string
  statusCmd          tea.Cmd
}
```

Concept: The core model handles the following responsibilities:

- Manages the left and right containers including CLI input, output, schema tab, and blame tab.
- Tracks the active tab, window focus, and locking status to control user interaction.
- Applies active and inactive styles dynamically based on focus and locked/unlocked state.
- Delegates events to sub-models:
  - `cliModel` handles command input, execution, and autocompletion.
  - `schemaView` renders and navigates the schema tree.
  - Right viewport displays blame/log information.
- Manages viewports for scrolling content and updating visible text.
- Updates status information for connected servers and renders the layout using `lipgloss`.

This design keeps the TUTUII modular, reactive, and interactive, allowing the main model to act as a central coordinator while delegating specialized tasks to dedicated sub-models.

4.5. UI Design

This section describes the design decisions that were made while implementing the TUI frontend. We will explain the styling decisions, accessibility features, focus mechanism, scrollability, color scheme, responsive design and show the different mockups we created during the design process.

4.5.1. Styling Decisions

The TUI was split into two main windows:

- Left window, containing the CLI where the user can enter their input and the output is displayed.
- Right window, storing the schema and blame view.

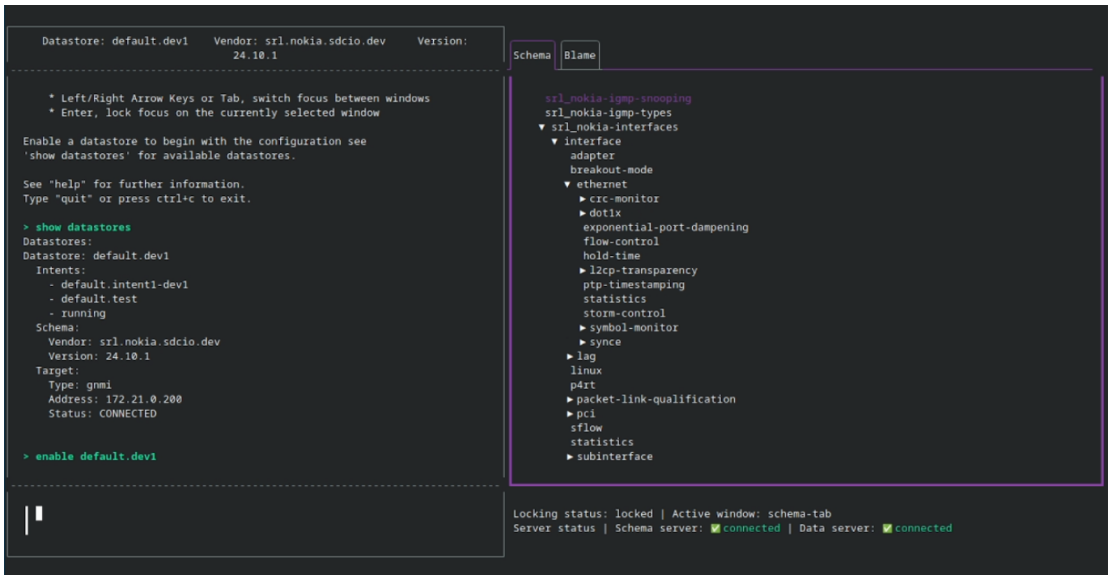

Figure 4.4.: Final version of the MVP

With the layout of the two halves the user has all needed information on one screen without loosing visibility of the CLI window, which could disrupt the configuration workflow.
For the outputs in the CLI window, the schema and blame view, a viewport component from framework was used. A viewport is a scrollable container that can hold content larger than its visible area.
The viewport component was chosen, because it supports scrolling, vertical paging progress, keybinding and mouse wheel interaction.

The top left-hand corner of the TUI contains a header, where the user can see which datastore is currently selected. To be able to create intents a datastore must be selected. Therefore, having this information displayed permanently gives the user clearance about which datastore they are working on.

The bottom right-hand corner, of the TUI, gives insights about the connection status of the data- and schema-server. This way the user can always see if the TUI is connected to the servers or not. As well as get information about which window they have currently focused.

4.5.2. Accessibility

To make the TUI more accessible, the commands entered by the user are displayed bold green in the output window, and if an error message occurs bold red accompanied by an ❌ emoji to enhance understanding for people with color blindness.
For the same reasons, emojis are also included in the connection status displayed in the footer. When the server connection was successful, we display an ✅ emoji and green bold text.
If the server connection has failed to be initiated, then an ❌ emoji and red bold text gets displayed.

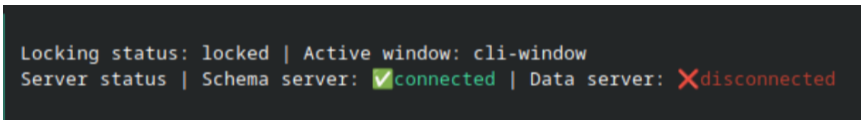

Figure 4.5.: Connection status in the footer

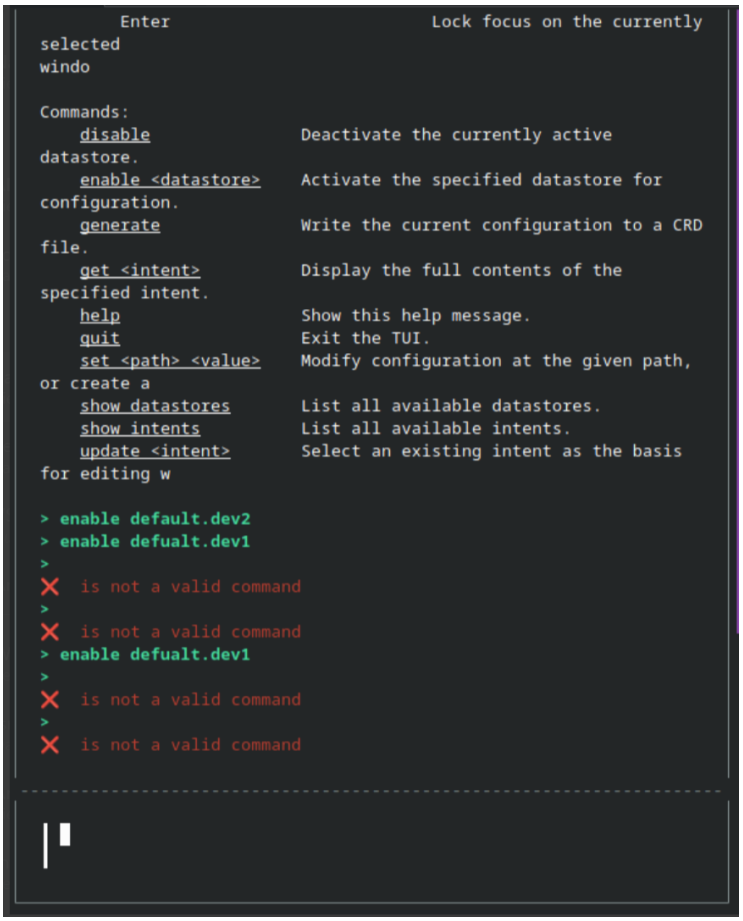

Figure 4.6.: Error message in the output window

For switching between the different windows, we added a bold white border to show on which window the user currently has focused. The following images illustrate the border.

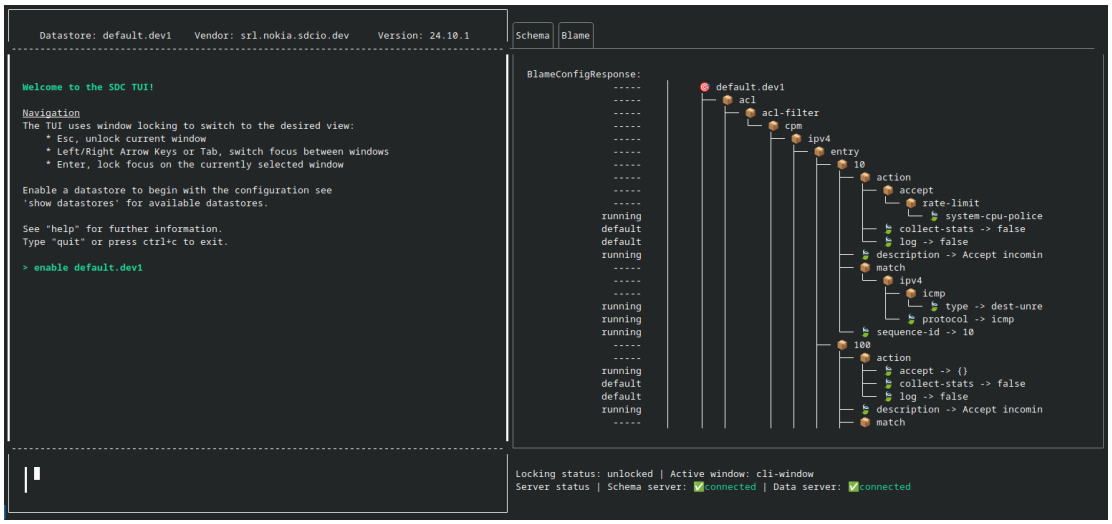

Figure 4.7.: Bold border on the output window

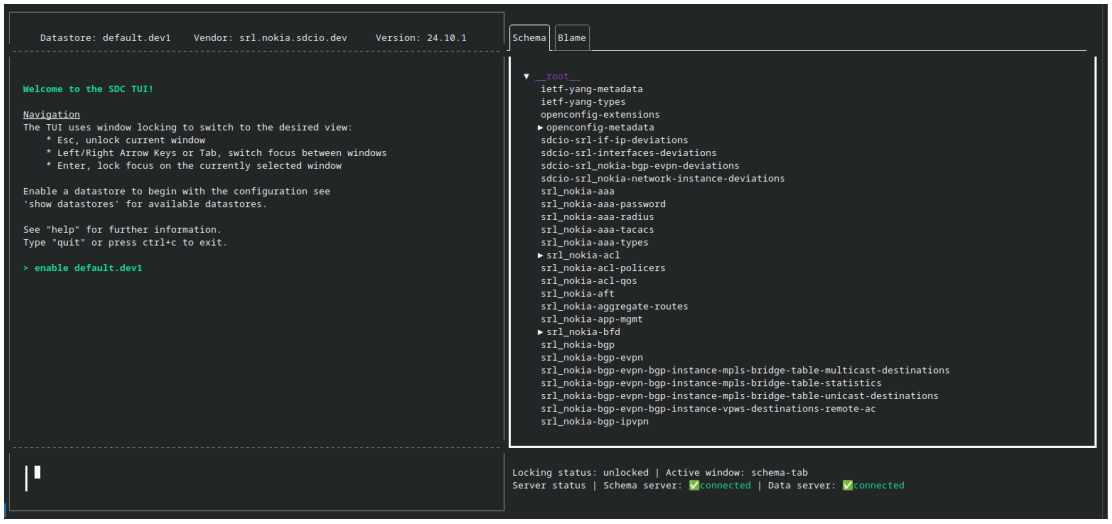

Figure 4.8.: Bold border on the schema window

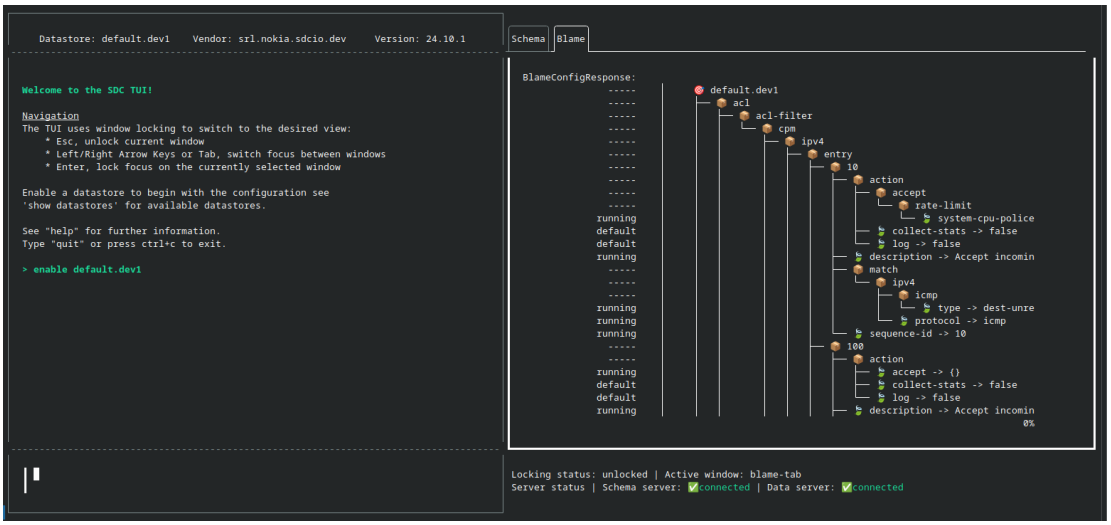

Figure 4.9.: Bold border on the blame window

A colorful border around the window, indicates to the user, which window they have currently focused. The inactive windows or views are displayed in gray. A more detailed description of the focus mechanism can be found in section 4.5.3.

4.5.3. Focus Mechanism

The user can switch between the different windows by pressing tab or the `left/right` arrow keys. The active window is highlighted with a thicker border.
Because the same keys have different meanings depending on the context - for example `tab` can be used for autocompletion but also to switch the focused window. To resolve this problem we implemented a focus mechanism, that adds different context to a key, depending on the window, that is active.

The unlocking mechanism works by pressing `esc` and to lock a window `esc` can be pressed again or `enter` is pressed. Once the window is unlocked, the user can switch to the desired view. Here an illustration of the focus mechanism:

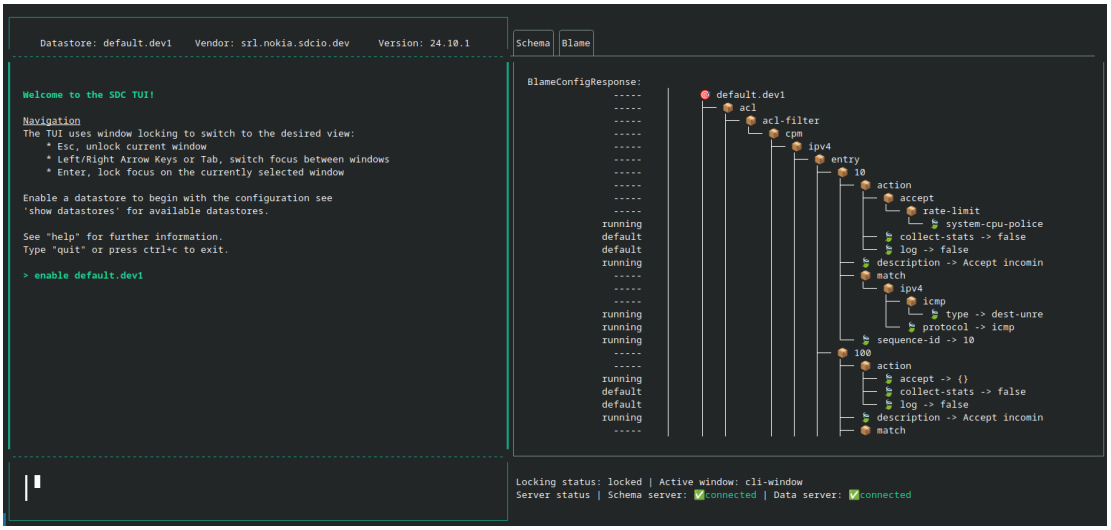

Figure 4.10.: Focus on the output window

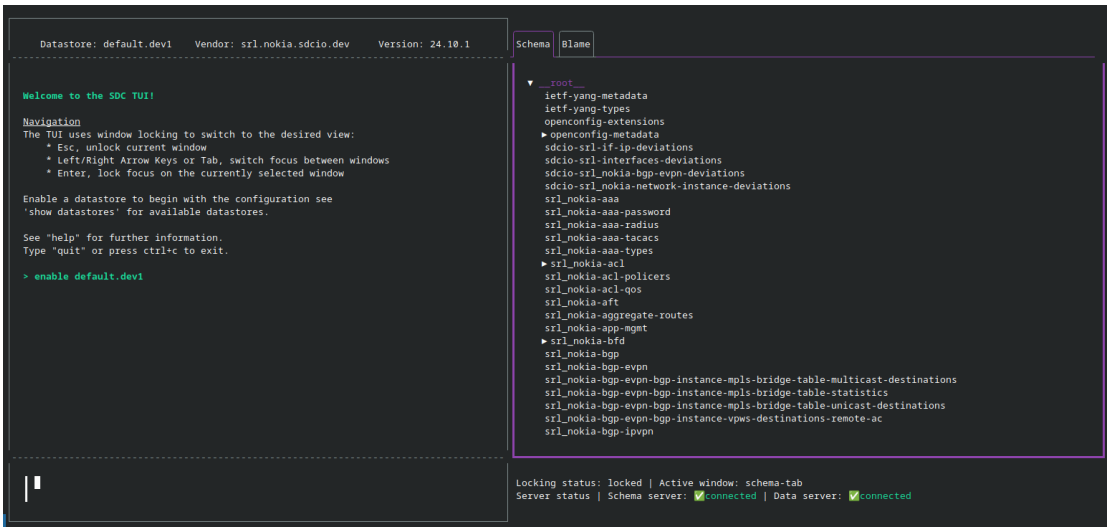

Figure 4.11.: Focus on the schema window

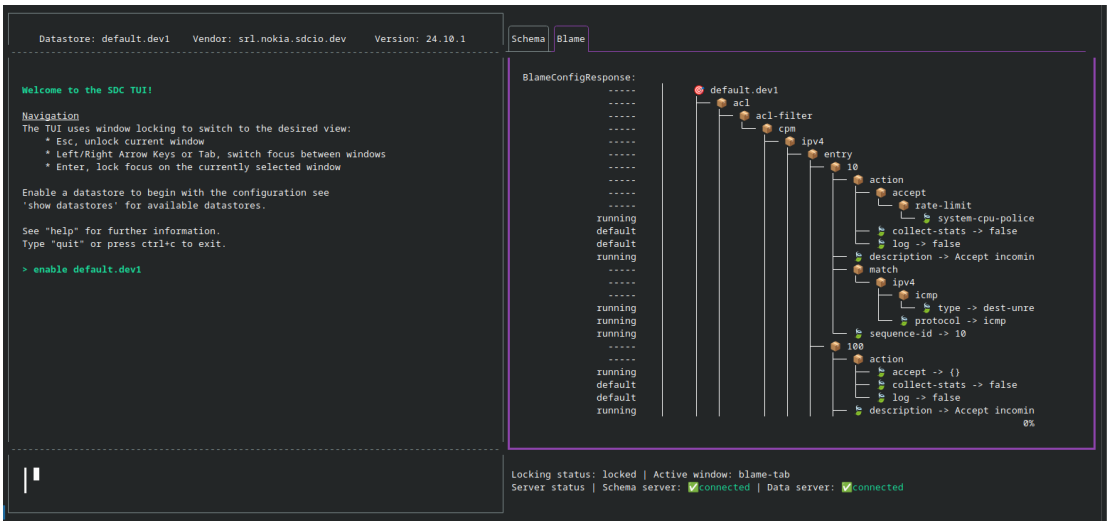

Figure 4.12.: Focus on the blame window

4.5.4. Scrollability

To make the TUI more user-friendly we added scrollability to the output, schema and blame view. Since the blame can be really big, there is an additional percentage indicator on the right side, that shows the user at which point of the file they are.
The output, schema and blame view support vertical scrolling and in addition the blame view also supports horizontal scrolling.
For now we only support horizontal scrolling in the blame view, because otherwise the blame output would make unfitting line breaks, which would lead to a confusing output. The content of the schema view in its current state is readable without the need for horizontal scrolling. Since there is a possibility that a schema node could extend the window width, we will support horizontal scrolling there in a future versions.

Figure 4.13.: Scrollability in the blame view with percentage indicator

Figure 4.14.: Horizontal scrolling start position in the blame view

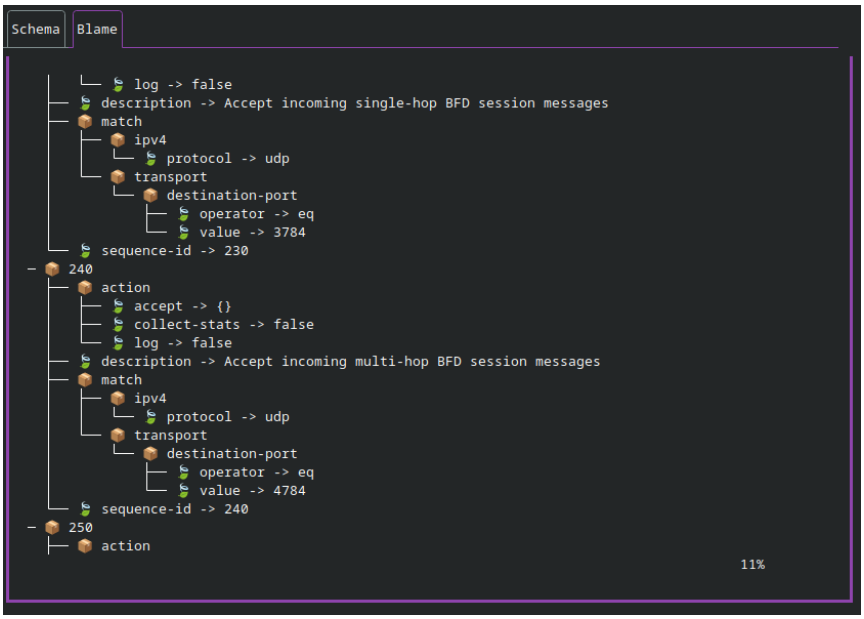

Figure 4.15.: Horizontal scrolling after scrolling in the blame view

4.5.5. Colors

First we had the colors blue and white set as our main colors, because they match the Nokia brand colors. But after testing the TUI across different shells, we realized that the colors vary depending on the terminal.

So we decided to use light-green, purple and gray, since these colors are more consistent across different terminal emulators and settings. Green is for the output window and purple for the schema and blame view. So the user can easily distinguish between the different windows.

4.5.6. Responsive Design

To make the TUI responsive we used percentage values for the width and height of the different windows. This way the TUI can adapt to different terminal sizes and the user has a good experience on any device. But because there are many small components in the TUI, we had some challenges to make it fully responsive.
In future versions we want to improve the responsiveness of our product even more.

4.5.7. Startpage and Help

During the user testing phase, the help for some guidance was wished for. Therefore a startpage was added that shows the user some basic information about the TUI and how to get help. The help page can be accessed by typing `help` or `?` in the input field.

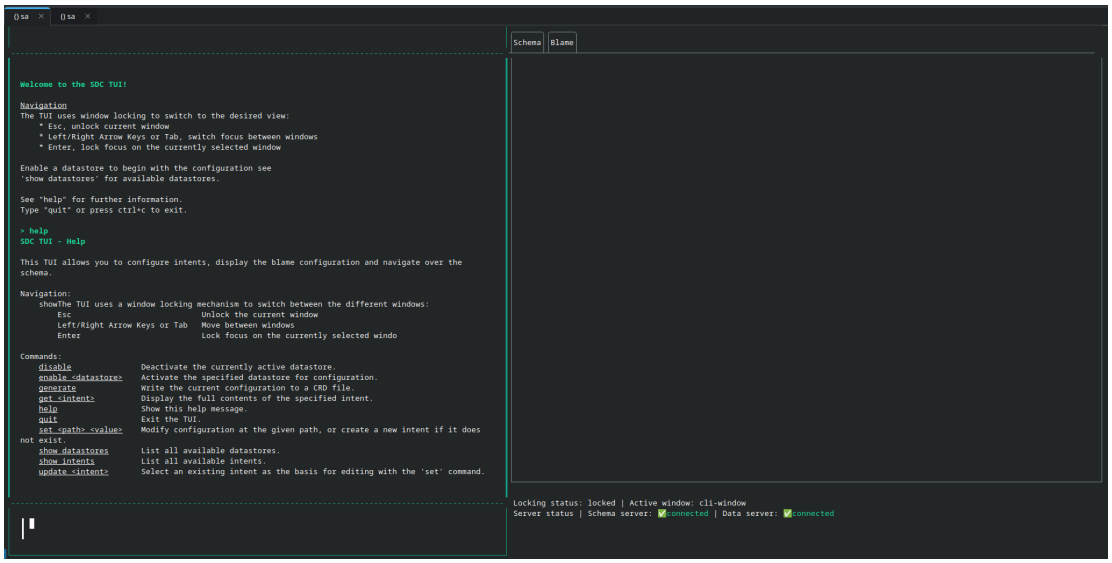

Figure 4.16.: Start- and help-page of the TUI

4.5.8. Improvement

During the implementation of an increasing number of features, difficulties were encountered with correctly rendering the window heights in the TUI. When the content of a view became too wide, the window borders were shifted upwards. After the addition of each new feature, this issue was addressed, but with the introduction of the latest feature, the same problem occurred again and is not yet fixed. This aspect needs to be improved in the next version in order to achieve more stable window heights.

4.5.9. Frontend Improvement

First the TUI frontend looked like shown in Figure 4.17. With the basic layout of the two main windows, input and output on the left and schema and blame view on the right and the blue and white color scheme. An illustration of the mockups and user journey can be found in the appendix.

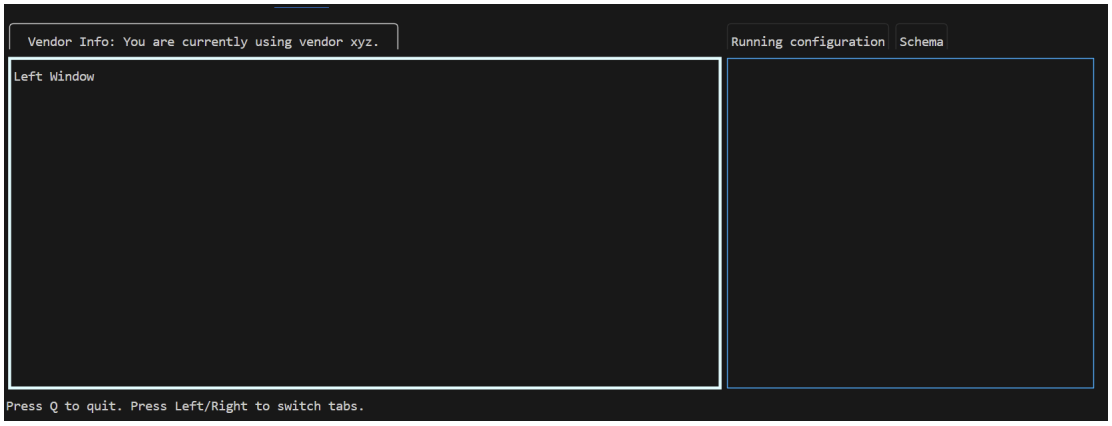

Figure 4.17.: First version of the TUI frontend

Later the input field and error output section were added to improve the user experience, as shown

in Figure 4.18.

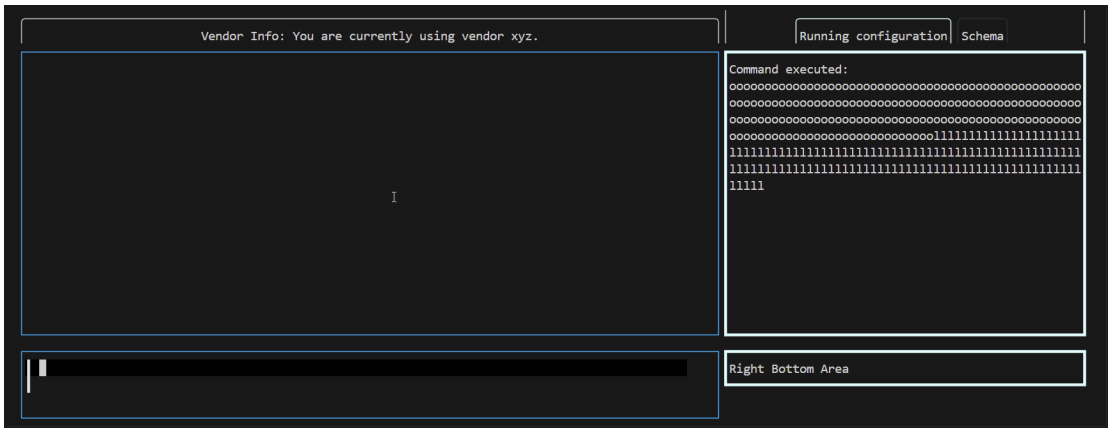

Figure 4.18.: Added input field and error output section

Later the progress bar was added as status indicator of the connection, as shown in Figure 4.19.

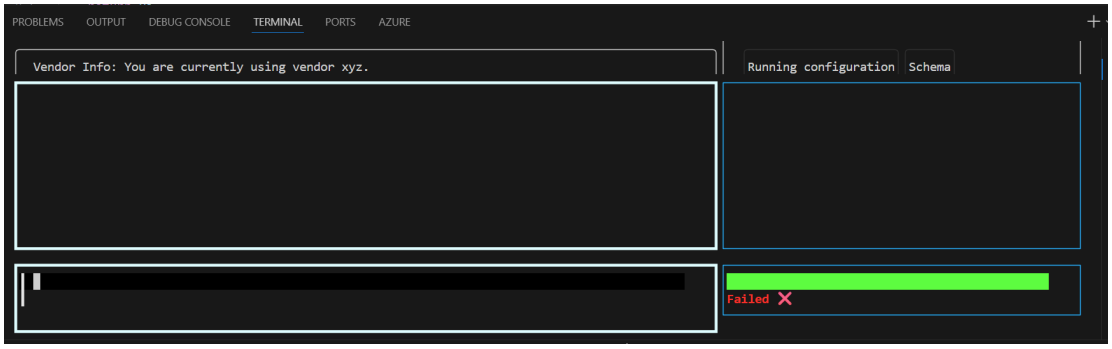

Figure 4.19.: Added progress bar as status indicator

Finally, the borders and colors were improved to make the TUI more accessible, as shown in Figure 4.20.

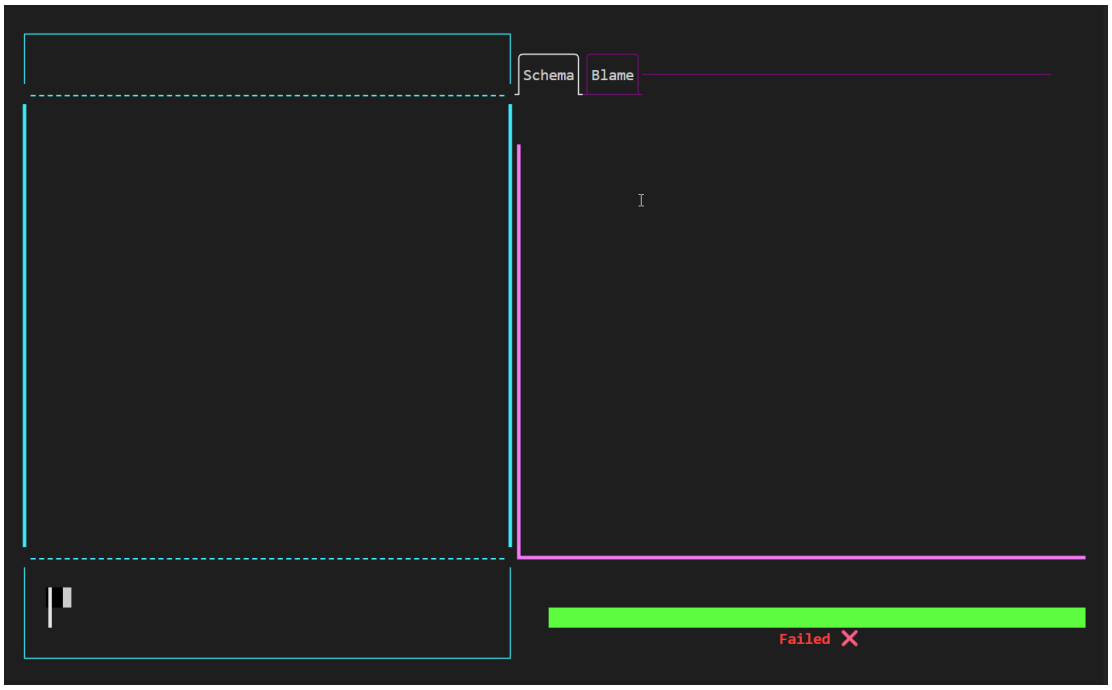

Figure 4.20.: Improved borders and colors of the TUI frontend

The last picture shows the final version of the TUI frontend, as shown in Figure 4.21. With corrected borders and output colors for better accessibility and displaying the inactive window with a gray border.

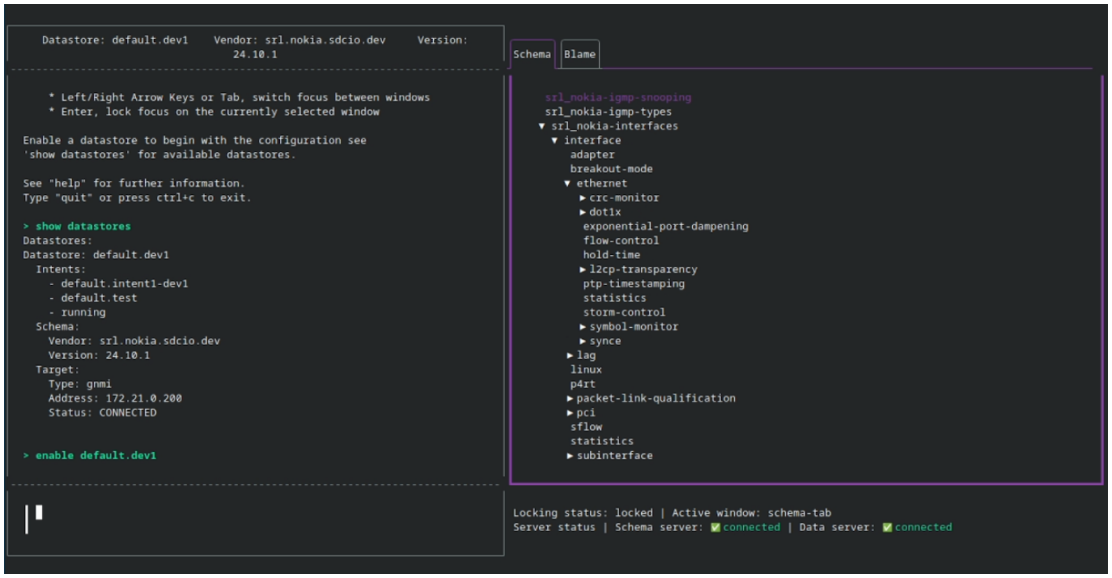

Figure 4.21.: Final version of the MVP

Bibliography

[1] Martin Björklund. *The YANG 1.1 Data Modeling Language*. RFC 7950. Aug. 2016. DOI: 10.17487/RFC7950. URL: https://www.rfc-editor.org/info/rfc7950 (visited on 12/11/2025).

[2] *Bubbles*. Repository of the framework Bubbles. URL: https://github.com/charmbracelet/bubbles (visited on 12/11/2025).

[3] *Bubbletea*. Repository of the framework Bubbletea. URL: https://github.com/charmbracelet/bubbletea (visited on 12/11/2025).

[4] *CiscoDevNet's Yangsuite*. Website of Cisco YANG Suite. URL: https://developer.cisco.com/yangsuite/ (visited on 12/16/2025).

[5] *Golang*. Wikipedia website for Golang. URL: https://en.wikipedia.org/wiki/Go_(programming_language) (visited on 12/11/2025).

[6] *gomock*. Documentation website for gomock. URL: https://pkg.go.dev/github.com/golang/mock/gomock (visited on 12/18/2025).

[7] *Latex*. Wikipedia website for Latex. URL: https://en.wikipedia.org/wiki/LaTeX (visited on 12/11/2025).

[8] *Latex - Grammar/Spell Checker*. Website of the linter Latex - Grammer/Spell Checker for Visual Studio Code. URL: https://valentjn.github.io/ltex/ (visited on 12/17/2025).

[9] *Lip Gloss*. Repository of the framework Lip Gloss. URL: https://github.com/charmbracelet/lipgloss (visited on 12/11/2025).

[10] *Nokia's YANG model explorer*. Nokia's SR Linux YANG model explorer. URL: https://yang.srlinux.dev/ (visited on 12/16/2025).

[11] sdc team. *SDC (Schema Driven Configuration)*. Official SDC website. URL: https://docs.sdcio.dev/ (visited on 12/09/2025).