PSI Documentation as Code

PSI-DAC

Version:	MS11 [1.3.0]
Date:	2025-04-23
Reference:	PSI-DAC
Total Pages:	36

Table of Contents

List of Figures

List of Tables

1 Document Meta Information

1.1 Document Signature Table

	Name	Function	Company
Author	Wolfgang Robben	Project Manager	CGI
Author	Hendrik Oppenberg	Technical Officer	CGI
Author	Christine Glaesser	Liaison Manager	CGI
Approval	Victoria McCarthy	Project Manager	SES
Checked	Pepijn Witte	Quality Assurance Manager	CGI

Table 1.1: Signature Table.

1.2 Document Change Record

1.2.1 Changes

Date	Version	author	message
2025-04-23	MS11 [1.3.0]	Christine Glaesser	Initial version

Table 1.2: DCR Table.

1.2.2 Source Control

Changes to this document are tracked electronically. No signature is required by the authors. The following information can prove the integrity of the document and reveal any change.

Repo	Date	Author	Branch	Hash

Table 1.3: GIT Changelog Table.

Figure 1.1: DCR QR-Code.

1.3 Documents

1.3.1 Reference Documents

Acronym	Reference	Title	Version
PSI-TAD	PSI-TAD	Terms, Abbreviations and Definitions	see before

Table 1.4: Reference Documents

2 Introduction

The Pooling & Sharing Interfaces Definitions (PSID) project is an ESA co-funded effort to define a common standard for the interfaces of Pooling & Sharing Systems (PSS) for Satellite Communication (SatCom) services. A PSS is a digital platform for matchmaking (Gov)SatCom users' demands (both commercial and institutional) with (Gov)SatCom providers' offers. Bringing together multiple (Gov)SatCom providers in one platform makes the market transparent, thus allowing users to get an overview of the market and to compare different offers efficiently. Additionally, a PSS assists users with little knowledge about the (Gov)SatCom domain defining their requirements on the (Gov)SatCom services. Those two aspects combined allow for fast access to the services and an efficient usage of the available capacities. To accomplish this, a PSS steps in between the usual processes of finding a provider/supplier, requesting an offer, and ordering the desired products or services, either as a service broker or by pooling products and services from different providers and offering them as an intermediary or distributor. Subsequently, the PSS can be used to monitor the services and manage multiple missions in a single application.

Eventually, a PSS can also be used as (or manage) a community hub, i.e., a number of end users or customers with similar interest that *share* their common resources and utilize a commonly obtained *pool* of (Gov)SatCom capacities. This strategy increases the efficient usage of scarce resources further.

There are already different approaches on PSSs, that might lead to an unnecessary fragmentation of the market. Therefore, a common standard for the interfaces of a PSS is required to allow the interaction between those different PSSs and reduce the effort of (Gov)SatCom providers to offer their product and services via multiple PSSs to maximize their reach.

Such a standard needs to take care of the different interfaces involved in the aforementioned processes, i.e.,

The goal of this project is to mainly define aspect 1 and to develop a software mock-up as needed to validate the various interfaces being developed.

The PSI standard derives from the existing industry-standard “Open Digital Framework” of **TM Forum** alliance[1]. The “Open Digital Framework” is a reference framework for delivering online Information, Communications and Entertainment services to the telecom world. It empowers market participants to compete and cooperate. One of PSI's goals is to make this existing standard fit for the world of satellite communication.

The consortium for this project consists of the service & technology providers SES Techcom and CGI, as well as of the (Gov)SatCom operators SES, Hellas Sat, Hispasat, Hisdesat, and LuxGovSat, and Inmarsat being both a service & technology provider and a (Gov)SatCom operator.

[1] See https://www.tmforum.org/resources/reference/gb991-tm-forums-core-concepts-and-principles-v22-0-0/

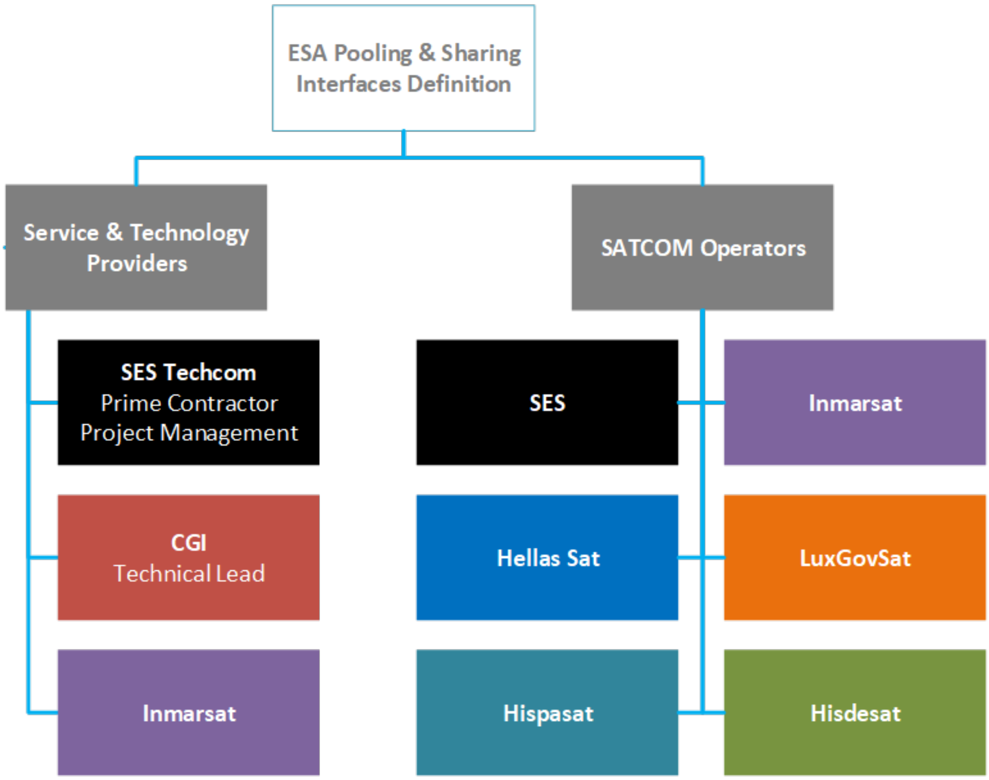


Figure 2.1: The PSI consortium.

2.1 Document Scope

This document describes the workflows and technologies around *Documentation-as-Code* pipeline and Requirements Traceability Matrix (RTM) applied within the PSI project. The document will outline the benefits of handling documentation in such a way, as well as the general outline and outlook for potential adaptation to future projects.

2.1.1 Compiled Document

NOTE: THIS IS A COMPILED DOCUMENT [2]

This document has been compiled/generated from external sources and is not being written as-is. Therefore, any changes made within this compiled version of the document will be lost upon recompilation!

To make (permanent) changes, edit the respective sources directly or contact the PSID team.

2.1.2 Signature

Changes to this document are tracked electronically. No signature is required by the authors. The information in the "Source Control" chapter can prove the integrity of the document and reveal any change.

[2]Document compiled on 2025-04-23 12:37.

2.1.3 Development State

Current document version is `1.3.0`.

3 Software Documentation

Each software product is developed following a design phase, which captures the customer's intent of using the finalised product. The documentation of the actual implementation shall explain how the software works, how it should be used and how it shall be maintained. This can include user documentation, e.g. guides and manuals for end-users, as well as technical documentation for developers. The technical documents may include the software's architecture, API documentation and details how to interact with the system in an automated manner. Also, third-party services included in the produced software and maintenance documentation like troubleshooting guides may be described in a technical documentation. Well-written documentation offers a vast amount of benefits, to name some examples:

Thus, well-written documentation helps end-users and software developers to use the software effectively and maintain it for long-term success.

Software documentation in the Space domain is especially critical as the systems and missions are complex and high-risk, also from cost perspective. Therefore, strict standards for both software development and documentation exist and are followed such as ISO 9001 or ECSS (European Cooperation for Space Standardization). These standards ensure the reliability, maintainability and security of the space software being developed, with a strong emphasis on clear traceability from requirements to final implementation and verification.

Different toolings have been established for space software related documentation. Among those are Configuration Management Systems like Git or Subversion, taking care of version control of both code and documentation, or Document Management Systems like Confluence. Model-Based Development Tools like SysML are often applied to document software behaviour, overall architecture or system interactions. For requirements tracing and verification, IBM's DOORS is a well-known application. DOORS also allows to customize settings, e.g. uploading Independent Verification and Validation (IVV) Test documents, to facilitate requirements tracing and verification.

3.1 Known issues for Software Documentation

While sound, well-written software documentation offers a vast amount of benefits as outlined above, it can be a challenge to keep the developed software and accompanying documentation on the same page. Especially in agile projects, feature development and its focus might change over the course of the project. Software documentation should be done while the feature is developed, however, as deadlines come closer, documentation is often shifted to the last minutes. Thus, software documentation is often error-prone as the actual development diverges from the initial intended feature implementation. As outlined above, different toolings exist for different goals of documentation. This represents an obstacle for developers, as they have to switch context and environment for documentation, which is emphasized once the deadline for feature development approaches. As result, documentation on software is often perceived as (annoying) add-on to existing software solutions, a nice-to-have to fulfill requirements. While updates done on software are done in an automated way via CI pipelines, are not flown down to documentation in same manner. As a result, documentation is often not in best state and very quickly outdated as new features are being developed.

To mitigate those issues, the idea to treat documentation in same manner as code was born.

4 The concept of DaC

The concept of Documentation-as-Code (DaC) was introduced in the mid-2010s. It aimed at treating documentation as integral part of the codebase, applying same principles (e.g. code review to documentation review) and tools to documentation as is done for code. Although the idea was recognised earlier, the approach has become more common as software development practices have shifted towards more agile methodologies and the increasing adoption of version control systems like Git, growing popularity of plain text such as Markdown and the adoption of Continuous Integration and Continuous Delivery (CI/CD). Since 2022, DaC is an established practice in the software development community [3], particularly in DevOps, agile and open-source communities.

4.1 Benefits

As outlined, software documentation that is not included in the codebase suffers from specific defects.

DaC can mitigate those by streamlining the workflows for both code creation and documentation generation. The environments usually used for coding, e.g. Eclipse on Linux or Visual Studio Code on Windows, allow to write files in Markdown native. Specific libraries, easily plugged in, allow the developer to stay in their well-known coding environment, while writing documentation with IntelliSense support in Markdown. This treatment renders documentation as artifact, just like implemented code.

Additionally, it is beneficial to inspect the documentation of a feature alongside the existing code, allowing to spot differences and adopt small, targeted changes to documentation. Including the resulting documentation, such as Markdown files, in code commits can address the aforementioned discrepancy between the codebase and documentation. Additionally, implementing regulations, such as checking the peer review before merging, ensures that the documentation is updated for specific features. Thus, DaC helps to mitigate the issue of a mismatch between the codebase and documentation.

Once the documentation has been transferred to a format that allows for automation, the automated formatting of the documentation to the customer's desired specifications is just one click away. This can speed up delivery time; while it is usually a significant amount of (manual) time to correctly convert documentation to another format, automating such via CI/CD pipelines like offered by GitHub and GitLab can reduce such delivery time. Thus, the delivery can be done with codebase and documentation in any format side by side.

Treating documentation as artifacts just like code allows for easy version control as well. Just like code, documents can then be tracked, reviewed, and rolled back without losing important information. If implemented in, for example, a Markdown or subsequent rendering flag, the version the documentation has been generated on can also be included in the document, e.g. in form of a Git hash code. This allows for a very convenient way of reviewing that the documentation at hand is matching the codebase, comparing e.g. the Git hash.

Traditionally, documentation is written and maintained by technical writers. This demands close collaboration between developmental team and the writing board. If not given, documentation might not match the actual feature implementation or, worse, might contain wrong information which will impact future development and maintenance activities. If however documentation is part of development and treated like any other software artifact, developers are encouraged to contribute to documentation and maintain such. The technical writers can then focus on the overall quality rather than content.

In addition, if documentation is part of the repository, each team member easily has access to the up-to-date documentation. This facilitates onboarding new team members and knowledge sharing.

Overall, DaC emphasizes collaboration, transparency and automation.

[3] See https://daniel-woste.de/posts/2024/Documentation_as_Code_in_Automotive_System_Software_Engineering.html

5 DaC application in PSID

In the context of the PSID project, we have adapted DaC to take advantage of the benefits described above. All documentation is hosted alongside the codebase, in the same repository, and pushed regularly to GitLab as artifact (cp. document PSI-SDP). The resulting documents are converted to PDFs and finally delivered to the customer.

5.1 Technical implementation

In PSID, all documentation is written in Markdown, a lightweight markup language designed for creating formatted text with a plain-text editor. The syntax is easy-to-read as well as easy-to-write, allowing to format plain text without using complex syntax mechanisms. Elements like headings, lists, bold or italic text can be created using simple text formatting like punctuation or familiar characters like asterisks or hashtags:

```
**bold**
```

will be presented as **bold**, while

```
*italic*
```

will be presented *italic*. Hashtags are used to identify chapters and, related to the number of hashtags, as subchapters

```
## Subsection Level 1
```

Primarily used for README files, Markdown was quickly established as standard for collaborative software development, e.g. as standard documentation form in GitHub repositories.

Within PSID, the repository contains the codebase next to the documentation baseline. Each aspect of documentation is written in granular Markdown files, which are organized in dedicated folders. As example, each decision record is written in a granular file, but located in a folder hosting all content for the PSI-MADR deliverable. Some sections of our documentation however are part of a common introduction or preamble. Those sections are written on granular level as well, and hosted in a *common* folder to be used by all deliverables.

While Markdown has proven very powerful for short, simple documents, Markdown is not best choice for advanced documents. As example, footnotes cannot be supported easily. Additionally, more sophisticated stylings like table captions, table of content, references to other sections or documents as well as inclusion of other documents are not natively supported by Markdown. Change records, desirable for understanding the document's evolution, is also not natively supported in Markdown.

5.1.1 RHOD

To overcome these shortcomings, a bespoke toolset called RHOD (Ruby Handles Our Documents) was adapted to the specific needs of PSID. RHOD is a Ruby-based tooling, rendering Markdown files using specific markers to LaTeX files and finally to PDF documents.

LaTeX is widely used in academic and technical fields for typesetting documents. Thus, it supports focussing on the content rather than the formatting of a document. The formatting is usually done applying a LaTeX template file, which includes styling commands like introducing hyperlinks, clickable references to other document sections, figures or tables, and commonly styled headers and footers in the resulting document. While LaTeX supports output formats like HTML, EPUB or PostScript, LaTeX files are most commonly transferred to rendered PDF files. Those files are then styled according to the settings in the template, i.e. including hyperlinks and automated references within the document. RHOD makes use of those features and, applying a defined template, renders the initial written MarkDown files via LaTeX to PDF documents.

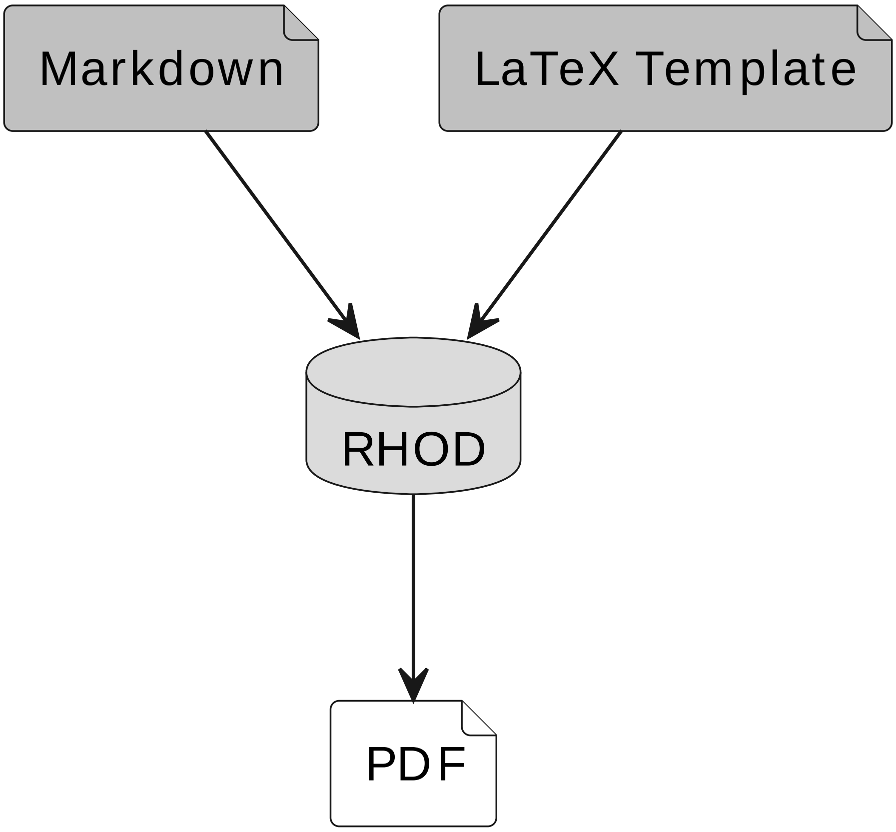


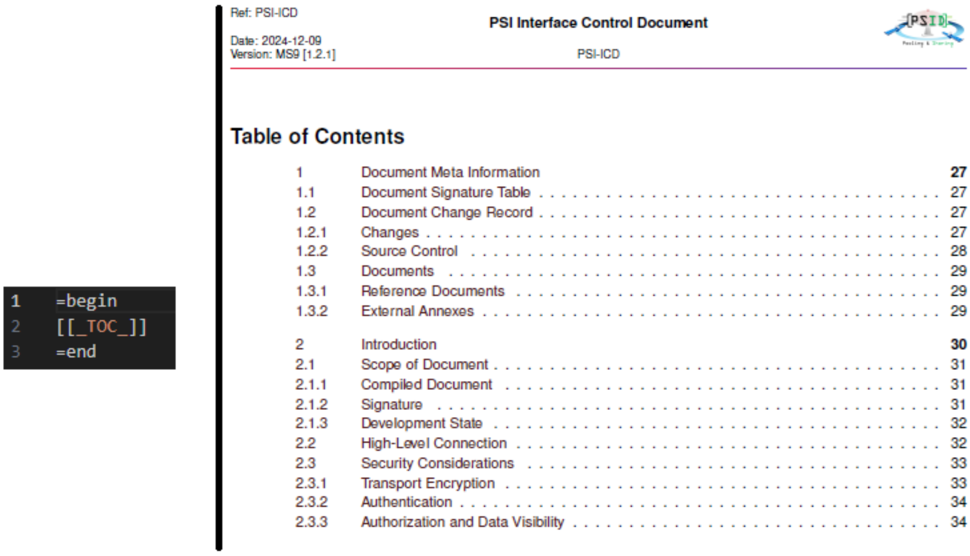
Figure 5.1: RHOD rendering pipeline, sketch

5.1.1.1 RHOD Markdown syntax

RHOD interprets specific Markdown syntax, using those as configuration items for adapting the resulting PDF according to the LaTeX templae. For a better understanding, the following images depict the Markdown code on the left, the rendered PDF output on the right. In addition, syntax examples are given for each chapter.

Document Header and Meta Information

The Markdown file includes a marker for the table of content as well as a command for rendering the document change record. The resulting PDF then includes the complete table of content and a table denoting the change record.



```
=begin
[[_TOC_]]
=end
```

Ref: PSI-ICD
Date: 2024-12-09
Version: MS9 [1.2.1]
PSI Interface Control Document
PSI-ICD

Table of Contents

Figure 5.2: Table of Content, Markdown and rendered PDF output

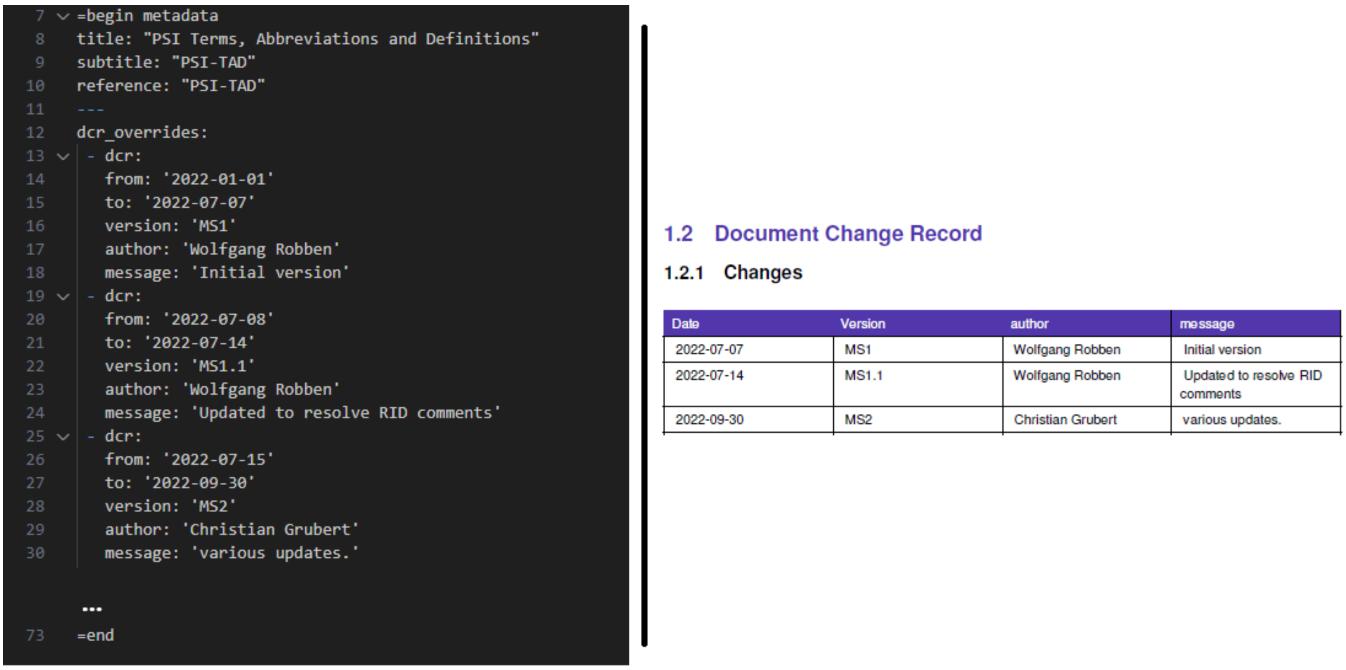
Metadata is entered into the Markdown file with syntax

`=begin metadata <text> =end`

Document Change Record

In the same command as in the metadata command, the Document Change Record can be defined via

`drc_overrides: - <...>` as shown in figure 5.3.



```
=begin metadata
title: "PSI Terms, Abbreviations and Definitions"
subtitle: "PSI-TAD"
reference: "PSI-TAD"
---
dcr_overrides:
  - dcr:
    from: '2022-01-01'
    to: '2022-07-07'
    version: 'MS1'
    author: 'Wolfgang Robben'
    message: 'Initial version'
  - dcr:
    from: '2022-07-08'
    to: '2022-07-14'
    version: 'MS1.1'
    author: 'Wolfgang Robben'
    message: 'Updated to resolve RID comments'
  - dcr:
    from: '2022-07-15'
    to: '2022-09-30'
    version: 'MS2'
    author: 'Christian Grubert'
    message: 'various updates.'

 ...
=end
```

1.2 Document Change Record

1.2.1 Changes

Date	Version	author	message
2022-07-07	MS1	Wolfgang Robben	Initial version
2022-07-14	MS1.1	Wolfgang Robben	Updated to resolve RID comments
2022-09-30	MS2	Christian Grubert	various updates.

Figure 5.3: Document Change Record, Markdown and rendered PDF output

To be compiled, the table needs to be included after definition via

`@dcr(my_dcr_table)`

in the Markdown file.

Additionally, the GIT history can be compiled using

```
@dcr(my_git_table)
```

and a QR code linking to the document history in GitLab can be included via

```
@dcr(git_qr_code){#fig:my_git_qr_code}
```

The Document Change Record is then listed as table, showing the date of change, the version, the author and the intended changes. The GIT history is compiled within Markdown to be rendered as table in the resulting PDF, including the latest change record and the corresponding hash of the commit. The QR code is shown on the compiled PDF, as can be seen in figure 5.4.

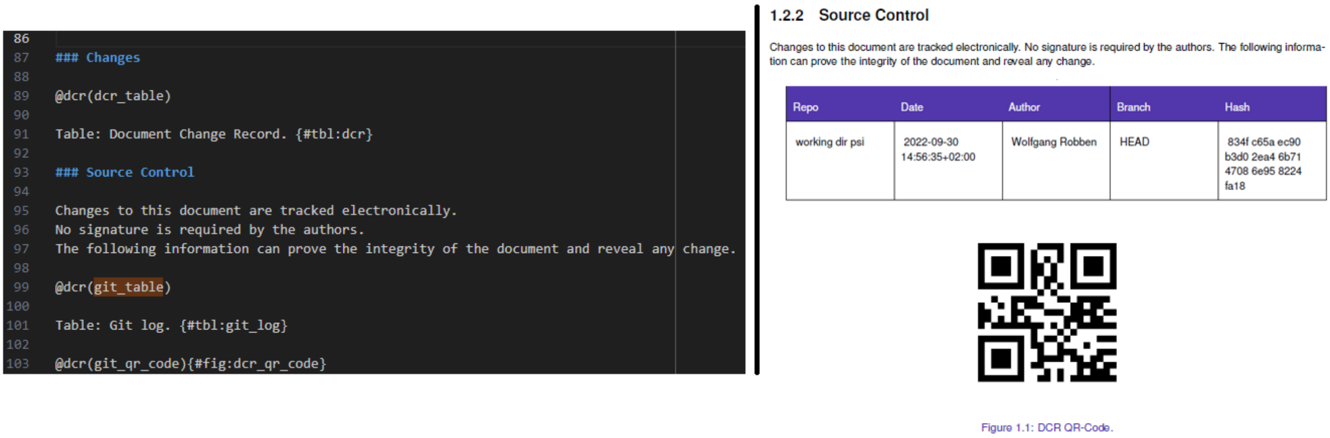


Figure 5.4: Document Change Record, Markdown and rendered PDF output

Referencing

Different figures and tables might need to be referenced in the document. As can be seen in figure 5.4, the QR code figure is marked after inclusion by

```
{#fig:myFigure}
```

Referencing in text is done using the @-sign, e.g.

```
as can be seen in figure {@fig:myFigure}
```

The same holds true for tables, substituting *fig* with *tbl*.

Including other Markdown files

As already outlined, Markdown has proven to be very powerful for short documentation, but not for large, excessive documentation. In PSI, each document consists of granular sections that are stored in dedicated Markdown files, e.g. the common introduction. This allows to re-use granular documentation sections for all documents, as for instance the common project introduction. These sections are referenced within Markdown files as

```
@ include[defined doc name](path-to-md-file)
```

as can be seen in figure 5.5.

```
# Introduction

@include [common introduction](../common/intro_description.md)

```

> Ref: PSI-ICD
> PSI Interface Control Document
> Date: 2024-12-09
> Version: MS9 [1.2.1]
> PSI-ICD
>
> ## 2 Introduction
>
> The Pooling & Sharing Interfaces Definitions (PSID) project is an ESA co-funded effort to define a common standard for the interfaces of Pooling & Sharing Systems (PSS) for Satellite Communication (SatCom) services. A PSS is a digital platform for matchmaking (Gov)SatCom users' demands (both commercial and institutional) with (Gov)SatCom providers' offers. Bringing together multiple (Gov)SatCom providers in one platform makes the market transparent, thus allowing users to get an overview of the market and to compare different offers efficiently. Additionally, a PSS assists users with little knowledge about the (Gov)SatCom domain defining their requirements on the (Gov)SatCom services. Those two aspects combined allow for fast access to the services and an efficient usage of the available capacities. To accomplish this, a PSS steps in between the usual processes of finding a provider/supplier, requesting an offer, and ordering the desired products or services, either as a service broker or by pooling products and services from different providers and offering them as an intermediary or distributor. Subsequently, the PSS can be used to monitor the services and manage multiple missions in a single application.
>
> Eventually, a PSS can also be used as (or manage) a community hub, i.e., a number of end users or customers with similar interest that *share* their common resources and utilize a commonly obtained *pool* of (Gov)SatCom capacities. This strategy increases the efficient usage of scarce resources further.
>
> There are already different approaches on PSSs, that might lead to an unnecessary fragmentation of the market. Therefore, a common standard for the interfaces of a PSS is required to allow the interaction between those different PSSs and reduce the effort of (Gov)SatCom providers to offer their product and services via multiple PSSs to maximize their reach.
>
> Such a standard needs to take care of the different interfaces involved in the aforementioned processes, i.e.,
>
> 1. an interface between PSS and resource providers (satellite operators, service providers, or other PSSs),

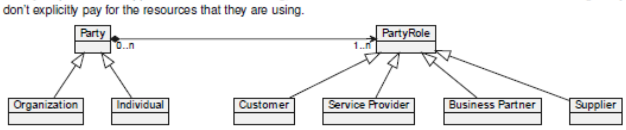
Figure 5.5: Inclusion of granular, common Markdown building block, Markdown and rendered PDF output

UML diagrams

Within our documentation, we use UML diagrams to represent workflows, dependencies or different aspects of a concept. These diagrams are annotated in PlantUML, an open-source tool using simple, text-based language. As such, the annotation of each UML diagram is seamlessly included in the overall document. Therefore, the diagrams are versioned alongside the documentation, enabling to spot differences between newly committed and previous versions of the diagram in the review. An example of a UML diagram, written in PlantUML, and the corresponding rendering is shown in figure 5.6.

````
```plantuml
@startuml
object Party
object PartyRole
' Tabs for formatting
Party "0..n" *-> "1..n" PartyRole : "\t\t\t\t\t\t\t\t\t\t"

object Organization
object Individual
Party <|-- Organization
Party <|-- Individual

object Customer
object "Service Provider" as Provider
object "Business Partner" as Partner
object "Supplier" as Supplier
PartyRole <|-- Customer
PartyRole <|-- Provider
PartyRole <|-- Partner
PartyRole <|-- Supplier

@enduml
```

![Parties and roles.](../common/pixel.png){#fig:parties_and_roles}

````

> ### 4.2 Parties
>
> Business interactions are executed between two or more parties, which are either natural persons ("individuals") or organizations. A party can have different roles in different interactions. For example a service provider can also be the customer and business partner of another provider, who is then a supplier, too.
>
> A PSS might also be used as a hub where different users share resources without cost, where access to them is merely requested and approved. For easier reading, these users are also referred as customers even though they don't explicitly pay for the resources that they are using.



> Figure 4.5: Parties and roles.

Figure 5.6: Definition of an UML diagram, Markdown and rendered PDF output

The resulting image is rendered as vector image, allowing seamless zooming at any screen size. Additionally to PlantUML, Mermaid and Vega graphs are supported by RHOD.

5.2 Automated documentation compilation

As outlined above, DaC facilitates documentation and coding in same environment and allows handling documentation as artifact within GitLab. Thus, CI/CD pipelines can be set up to automatically generate documentation alongside code sanity checks, e.g. via Linters or SonarQube. In PSID, we include RHOD in our CI/CD pipeline to be running after each commit. The documentation is then automatically rendered and delivered as downloadable artifact. Also, some documentation like our ICD document is being generated using the same pipeline; this is being described in more detail in subsequent chapters. In addition, a watermark is included on each finalized document, allowing to distinguish between pre-delivery documents or documents that can be delivered towards the customer.

6 Automation of requirements traceability

Requirements serve as foundation to define functionalities an envisioned software shall have and also the constraints the software shall express. They are guiding the overall design and development, serving as benchmarks to align user needs and expectations with the actual software implementation. They are often refined during the requirements engineering process, i.e. translating the initial (candidate) requirements as given by the user perspective to requirements that are specific to the overall software. As example, a requirement demanding secure connect can be derived by a project-specific requirement asking for implementing Keycloak to handle single-sign login.

Requirements fall into three major categories, namely:

Thus, throughout the project runtime, a close monitoring of requirements and implementation of features within the software is advised. A Requirements Traceability Matrix (RTM) often serves as such a monitoring tool. The RTM tracks requirements via linking those to design, development, testing and deliverables. The coverage of requirements and their interdependency can be inspected applying an RTM, allowing to adapt changes into the software while ensuring alignment with the overall objectives. The RTM further supports demonstration of compliance with standards through demonstrating the coverage of respective requirements within the project.

6.1 Design principles of an RTM

For implementing an RTM, the objective of the matrix is firstly defined. For instance, ensuring all requirements have been tested and implemented might be the focus of tracing requirements. Assessing the impact of changing requirements to the overall software solution might also be the primary goal. For PSID, ensurance of requirements coverage, i.e. implementation, has been the driver to install an RTM.

Secondly, the structure of the matrix has to be decided on. Usually, Excel sheets for initial collection are used, which can then be transferred to more sophisticated tools like DOORS [4]. Within PSID, we use an HTML matrix highlighting each requirement's status and reference to the requirement's associated status (cp. below).

Thirdly, the requirements are usually listed in the chosen format with a unique identifier, description, test case ID, test results and comments. For PSI candidate requirements, as given by the customer, we chose to present them accompanied by comments, assumptions for understanding their intention, their linked project-specific requirements, their justifying or descoping decision record and their status. Project-specific requirements are, in turn, presented alongside their unique ID, their title, their linked candidate requirements, associated operations and their status.

Lastly, the RTM is regularly updated to reflect changes in requirements or test results.

Maintaining an accurate RTM demands a good process. While this can be done manually on a regular basis, it is less error-prone to automate as much as possible.

This was done in PSID, as will be outlined in the following sections.

6.2 RTM automation within PSID

As mentioned above, the PSID documentation is written in Markdown and hosted in the same repository as the code. This handling allows to automate documentation generation, creating PDF files from Markdown applying a bespoke toolset called RHOD. The requirements tracing is treated likewise.

[4] See https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors/9.7.0?topic=overview-doors

6.2.1 Automated Integration Test Framework

Within PSID, we make use of an in-house toolset called AITF (Automated Integration Test Framework), a Ruby-based framework for the creation, execution and analysis of automated tests. It supports the execution of tests in multiple environments and shared tests in multiple missions. The AITF is designed to test complex system and provides the vast capabilities and flexibility of a full-blown programming language, extended by dedicated constructs required for testing. The AITF provides several means to analyse test results, including a web tool with graphical presentations of test results and access to all relevant test data. Tests are defined in Ruby source code files in a format mandated by the AITF (The AITF Test DSL). The format requires human readable descriptions which are directly linked to the source code, thus enforcing a clean and structured design of test procedures. By definition, a test procedure consists of a header with metadata like a title, a description and other optional attributes like links to requirements. The main part of the procedure is a sequence of test steps which are executed consecutively. For each test step, a description of the activities and an expected observation needs to be provided on top of the test code. AITF provides a command line interface for test execution. It is possible to execute a set of tests provided in XML format or single test procedures in a configurable environment. Our tests are thus triggered via CI/CD pipeline, running alongside our nightly builds to detect problems and regressions as early as possible (cp. 6.1).

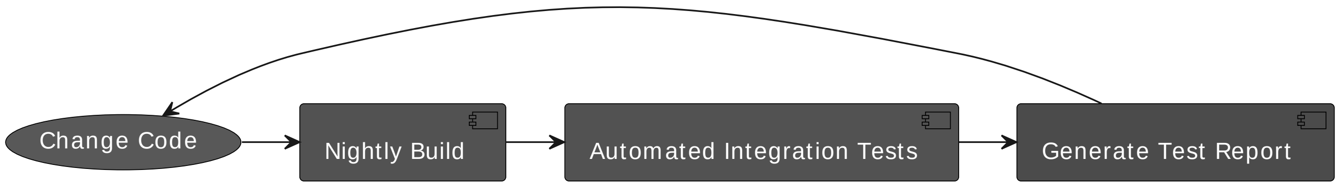


Figure 6.1: AITF nightly build testing, sketch

The results of those automated tests are, in PSID, processed as XML.

6.2.2 Requirements Tracing via repository

As aforementioned, all of our documentation is written as Markdown files alongside the code in a Git repository. The requirements baseline is not treated differently; all requirements that were given by the customer are documented in a dedicated Markdown document. The initial requirements are referred to as *candidate requirements*, requirements that are being analyzed throughout the project lifecycle, adapted to the project's needs and scope. These candidate requirements are transferred to project specific requirements, e.g. deriving from the requirement *The solution shall allow a user organisation to declare its own community of users with a maximum of three hierarchical layers.* the project specific requirement *The interface shall allow an organization to register its sub-organization(s) in hierarchical layers.* This link is well documented in granular Markdown files. Additionally, we describe in our decision records which implementations we chose and why. These records justify the implementation of a candidate requirement or descope its implementation, documenting the requirement as well in Markdown.

On the software side, requirements are linked on several levels. Firstly, tasks and operations that are implemented by the mock-up link to requirements. Tests on those implementations thus include the verification of requirements. Secondly, all tests that are being conducted include a meta data section, in which requirements tackled by the specific implementation are listed. AITF is used to run those tests in a nightly build and reports on the status of each test. As the requirements are linked to those tests, the result of the test run also shows the implementation status of affiliated requirements.

Both sources of requirements tracing, software and documentation, are used to build the RTM. A parser is used to gather information on each requirement and subsequently interpret the coverage per requirement.

On the documentation side, requirements are interpreted to be fulfilled if it is marked as *justified* in a decision record, as *partially implemented* if an aspect of the requirements is accepted while another aspect is rejected or as *analyzed* if the requirement is deemed to be descoped. On the software side, a requirement is interpreted to be *passed* if the corresponding test has passed or to be *failed* if the corresponding test fails. If a requirement is neither listed in a decision record nor in a test, the requirement is interpreted to be in *not yet analyzed* state. The results of the assessment as well as meta information on requirements - ID, title, comments, linked candidate requirement, operations, assumptions, decision record - are then compiled together with the final status in an HTML overview. The applied color code - red for *not yet analyzed* or *failed*, green for *justified* or *verified*, yellow for *partially implemented* and grey for *analyzed* - helps to get an immediate understanding of the coverage. As example, figure 6.2 depicts the report for the candidate requirements, figure 6.3 for project specific requirements linked to tasks and operations.

ID	Title	Comment	Assumptions	Linked Req.	Descoping MADR	Justifying MADR	Status
[illegible]	[illegible]		PSI_RA_001 PSI_RA_002		2023-04-24-descoped-reqs_V2.0		analyzed
[illegible]	[illegible]		PSI_RA_001 PSI_RA_002			2023-11-29-candidate-requirements-business	implemented
[illegible]	[illegible]		PSI_RA_001 PSI_RA_002		2023-02-01-beam-handling_V1.1	2023-02-01-beam-handling_V1.1	partially implemented
[illegible]	[illegible]		PSI_RA_001 PSI_RA_002			2023-11-29-candidate-requirements-business	implemented
[illegible]	[illegible]		PSI_RA_001 PSI_RA_002				not yet analyzed

Figure 6.2: Candidate requirements matrix, HTML overview.

ID	Title	Cand.Req.	Operations	Status
PSI-01-01-01-01	The interface shall allow an individual to register as a party to a PSS.	P&S_002	TOD-01-01-01	verified
PSI-01-01-01-02	The interface shall allow an organization to register as a party to a PSS.	P&S_002	TOD-01-01-01	verified
PSI-01-01-01-03	The interface shall allow an organization to register its sub-organization(s) in hierarchical layers.	P&S_138 P&S_140	TOD-01-01-01	verified
PSI-01-01-01-04	The interface shall allow an organization to register its associated community of individuals (users).	P&S_138 P&S_140 P&S_194	TOD-01-01-01	verified
PSI-01-01-01-05	The interface shall allow the governance to register a party to a PSS.	P&S_002	TOD-01-01-01	verified
PSI-01-01-01-06	The interface shall allow the store contract relevant information about a party.	P&S_024	TOD-01-01-01	verified
PSI-01-01-02-01	The interface shall allow an individual to update the existing party in a PSS.		TOD-01-01-02	verified
PSI-01-01-02-02	The interface shall allow an individual to update only their party in the PSS.		TOD-01-01-02	verified
PSI-01-01-02-03	The interface shall allow an organization to update the existing party in a PSS.		TOD-01-01-02	verified
PSI-01-01-02-04	The interface shall allow an organization to update only their party in the PSS.		TOD-01-01-02	verified

Figure 6.3: Project specific requirements matrix, HTML overview.

This overview is rendered interactive, i.e. the user can hover on the assumptions, the operations and the decision records to read their content as shown in figure 6.4, or click on it to scroll to the respective section of the RTM.

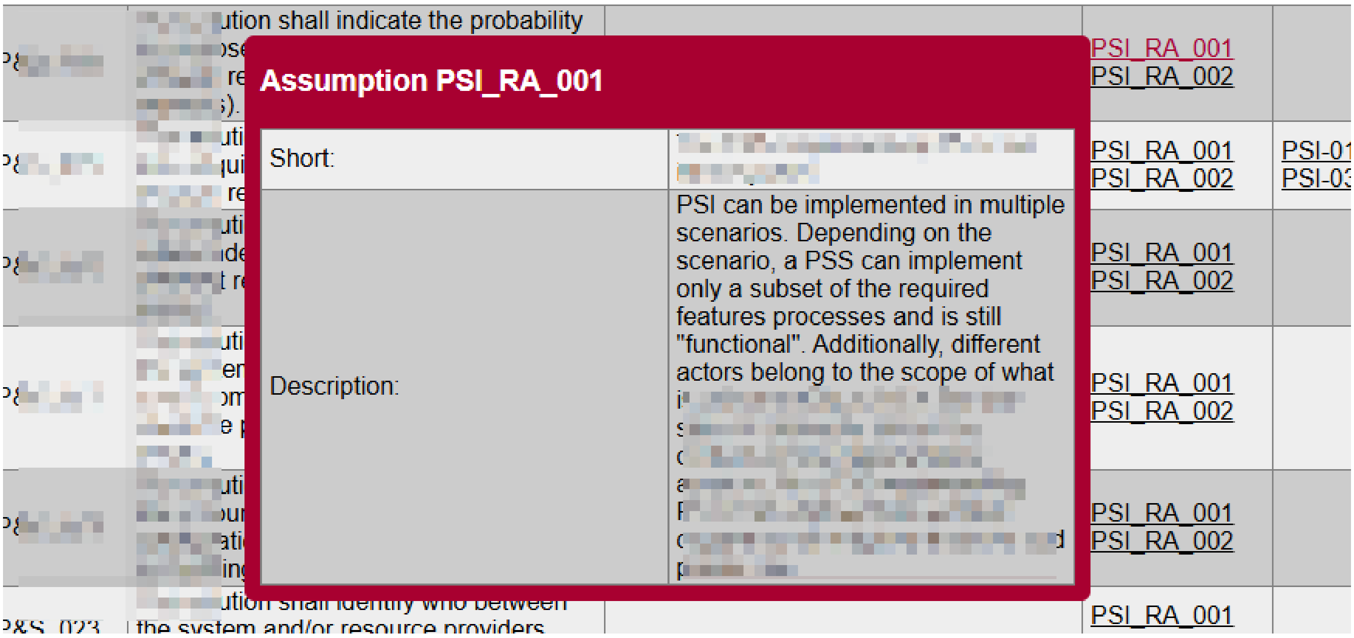


Figure 6.4: Project specific requirements matrix, HTML overview.

7 Automated Generation of OpenAPI Specification files

7.1 OpenAPI Specification files

Within the ODA framework, TM Forum hosts several API documentation files defining structure, endpoints and data models of TM Forum Open APIs. Those files - in TM Forum v4 called *swagger* files, in TM Forum v5 denoted as *OpenAPI Specification (OAS) files* - represent full definitions of APIs, rendered machine-readable. As such, developers can use these files to automatically generate libraries or test their implemented endpoints. Ensuring consistency across different implementations of TM Forum APIs, these OAS files are building the basis for TM Forum consistent API extensions or adaptions.

During the course of PSID, we migrated our APIs to match the OAS definition. As TM Forum moved forward to API version 5, the changes introduced to the previously denoted *swagger* files to *OAS* files reflect the key updates in version 5. Amongst those updates are:

OAS files are usually presented in JSON or YAML format and exist in different versions. All OAS files are available in TM Forum's GitHub repositories as well as in official documentation.

7.2 Creation of PSI OAS files

The aforementioned OAS files serve as template for PSI rendered OAS files. For this purpose, we introduce a Gradle based transformation language that is part of the public PSI repository. Gradle[5] is a well-known open source build automation tool, designed for Java, Android and Kotlin software development. As such, it supports building, testing and deploying software, compiling source code and packaging of software applications. It allows defining build tasks and configurations and offers a rich ecosystem of plugins extending its functionality to various domains. Furthermore, to improve build speed, incremental builds and parallel task execution are being supported.

The language allows the creation of transformation tasks inside the `build.gradle` configuration file. Each task is reading one input file from the TMF API definitions, applies a set of transformations and writes the result to the PSI API definition folder. The transformations vary from case to case, but are based on the following key principles:

Gradle also generates and builds a mock-up Java project including all defined end points. As Documentation as Code is strongly emphasized in PSI, the resulting API definition is rendered as Markdown documentation as well and hosted within our repository. Thus, the DaC pipeline is applied to generate the ICD based on those definitions and places the JSON schema files as annex to the ICD.

The aforementioned endpoints insert the data into a MongoDB instance, to be used in subsequent testing. MongoDB[6] is an open-source NoSQL database using a document-oriented data model. It is designed for scalability, flexibility and high performance. As opposed to relational databases storing data in tables and rows, MongoDB stores data in JSON-like documents with dynamic schemas. Thus, it supports evolving data models and, being optimized for fast reads and writes, supports high performant tasks for real-time applications.

[5]See https://gradle.org/
[6]See https://www.mongodb.com/

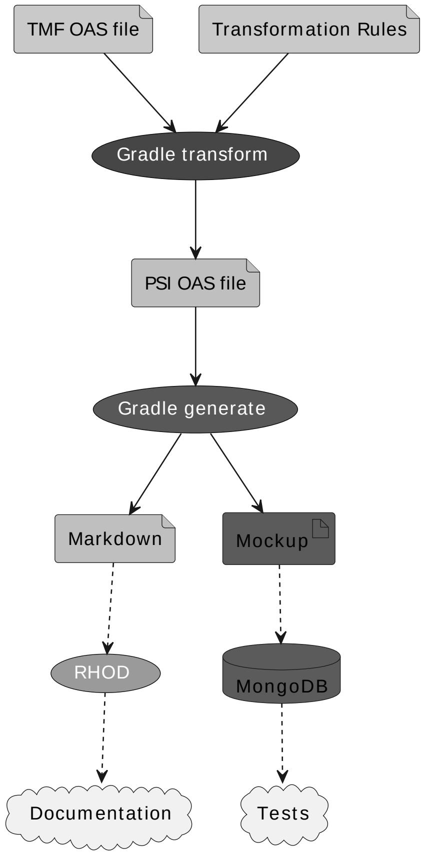


Figure 7.1: OAS Creation and Gradle workflow.

7.2.1 Validation of OAS files

AITF (cp. chapter Automated Integration Test Framework) is used to validate the end points and our APIs. It sends tests data to the Mockup, which in turn validates the request and inserts the data in our MongoDB instance. Subsequently, AITF verifies the answer being positive and matches the input data with the data now being present in the database. In a next step, AITF changes some specification and again verifies that these changes have been accepted. Finally, the AITF deletes the entry and again checks that the operation is working correctly as well.

8 Large Language Models for document generation

8.1 Introduction

Large Language Models (LLMs) represent a substantial advancement in the domain of artificial intelligence and natural language processing. These models are engineered to comprehend, generate, and interact with human language in a manner that emulates human-like comprehension and response. LLMs are trained on extensive corpora of textual data, thereby acquiring the capacity to discern intricate patterns and nuances in language. This aptitude renders them highly versatile for a plethora of applications, ranging from chatbots and virtual assistants to content creation and document generation.

The development of LLMs has been driven by breakthroughs in deep learning and the availability of large-scale computational resources. Models like BERT[7] introduced 2019 and rapid subsequent evolvements with T5[8] and the landmark research on Transformer architecture[9] have laid the foundation for modern LLMs, pushing the boundaries of what is possible in natural language understanding and generation.

It is evident that OpenAI's GPT series has been instrumental in facilitating the dissemination of LLMs to a wide audience, thereby positioning itself at the vanguard of LLM development. The advent of GPT-3 in 2020 signalled a paradigm shift in the field of LLMs, with ChatGPT establishing a new standard for language understanding and generation capabilities that had not been previously witnessed. In 2023, GPT-4 was released estimated to have around 100 trillion parameters, marking a significant leap in both size and capability. Following those developments, different LLMs have been developed and released for public usage. For instance, the Mistral Large[10] model was launched in February 2024 and positions itself as a competitor to GPT-4. It demonstrates excellent performance in reasoning and knowledge tasks, coding in multiple languages and multilingual understanding. In April 2024, Meta released the publickly available model Llama3[11], offering optimization for dialogue use cases and variants of 8B and 70B parameters. While these LLMs are publicly available and cloud-hosted, their application for company needs demanded locally hosted LLMs to ensure data safety. Thus, Ollama[12] was developed and published in 2023. Ollama is an open-source LLM that offers local deployment for the purpose of enhancing privacy and control. It is compatible with the OpenAPI API, which serves to simplify its integration. In addition, Ollama supports the integration of various LLMs, such as Mistral or Llama3.

The generation of documents represents one of the most promising applications of LLMs. The capacity to automatically generate coherent, contextually relevant, and grammatically correct text has far-reaching implications across a range of industries. Consequently, LLMs have the capacity to facilitate academic writing, assist with the drafting of research papers, generate literature reviews, and summarise complex scientific concepts. The automation of legal contract creation, briefs, and associated documentation via LLM application has the potential to enhance efficiency and reduce errors. Likewise, the generation of financial reports, market analyses and other business-related documentation has been demonstrated to enhance operational efficiency and optimise decision-making processes. LLMs have been furthermore demonstrated to possess a range of strengths, including proficiency in language trans-

[7] Devlin, Jacob; Chang, Ming-Wei; Lee, Kenton; Toutanova, Kristina (October 11, 2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". arXiv:1810.04805v2

[8] Raffel, Colin; Shazeer, Noam; Roberts, Adam; Lee, Katherine; Narang, Sharan; Matena, Michael; Zhou, Yanqi; Li, Wei; Liu, Peter J. (2020). "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". Journal of Machine Learning Research. 21 (140): 1–67. arXiv:1910.10683

[9] Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N; Kaiser, Łukasz; Polosukhin, Illia (2017). "Attention is All you Need". Advances in Neural Information Processing Systems. 30. Curran Associates, Inc.

[10] https://mistral.ai/news/mistral-large

[11] https://ai.meta.com/blog/meta-llama-3/

[12] https://ollama.com/

lation and creative content generation, question answering and chatbot implementation. As code is also interpreted as a form of language, LLMs have been shown to excel in the domains of code generation and assistance.

Within PSID, we apply automation for both code and documentation whenever needed and sensible. Thus, we analyze and investigate how applying LLMs for documentation generation can contribute to our pipelines and speed up our processes. Throughout the following chapters, we will delve into the intricacies of the aforementioned models, their development, and their performance in generating TM Forum mandated documents like the Userguide and the Conformance document for our APIs.

8.2 Ollama

Ollama is a large language model that represents a significant advancement in the field of natural language processing. Its development was driven by the need for a versatile and robust tool capable of handling a wide range of text generation tasks. The development of Ollama has been informed by advancements in deep learning, particularly the Transformer architecture, which has become the foundation for many contemporary language models. Ollama is a sophisticated piece of software that facilitates the execution of large language models (LLMs) on a local machine. This software, which is developed as an open-source project, has been designed with the objective of rendering AI models more accessible and practical for developers and researchers. Ollama facilitates the execution of AI models on personal computers, thereby reducing reliance on cloud services. The implementation of localised models is pivotal in ensuring the security and confidentiality of user data, thereby addressing the inherent privacy concerns that are often associated with cloud-based AI services. This feature is of particular significance in applications involving sensitive data, where the protection of information is of the utmost importance. Ollama facilitates the integration of numerous AI models, thereby democratising access to sophisticated AI technologies and rendering them accessible to a more extensive audience without the prerequisite of substantial computational infrastructure. This aspect is reinforced by the open-source release under the MIT license, which encourages community contributions and transparency. This openness fosters collaboration and continuous improvement, driving innovation in the field of AI.

For our purpose, we investigate how Ollama, installed locally, can be used to generate documents based on our API definition following the specified formatting of TM Forum Userguide and Conformance OAS accompanying documents.

8.2.1 Ollama local installation

Installing Ollama locally is possible for Linux, Windows 10 and later as well as macOS. The installation was tested on a Windows machine, following the steps described below.

8.2.1.1 Installing Chocolatey

Chocolatey is a package manager for Windows that has been designed to facilitate the process of installing and managing software on a system, thereby rendering it more efficient. It allows the user to install applications using simple commands in the command prompt or PowerShell, in a manner similar to the use of apt for Linux or Homebrew for macOS. Chocolatey supports both x86 and x64 architectures, thus rendering it a versatile tool for managing software dependencies on Windows machines. Chocolatey can be installed by opening a Windows PowerShell as administrator:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [
    System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).
    DownloadString('https://community.chocolatey.org/install.ps1'))
```

8.2.1.2 Installing Ollama

Staying in the Windows PowerShell as administrator, Ollama can be installed using chocolatey:

```
choco install ollama
```

8.3 LLMs: Llama 3 and Mistral

The landscape of large language models (LLMs) has witnessed significant advancements with the introduction of models such as Llama3 and Mistral. These models represent cutting-edge developments in natural language processing, with each offering unique strengths and capabilities. While they share some similarities, there are notable differences between them that are outlined in the following chapters.

8.3.1 Architectural Distinctions and Efficiency

Llama3 employs advanced self-attention mechanisms that are optimised to capture long-range dependencies in text. This architectural feature enables the model to maintain coherence over extended passages, rendering it particularly effective for tasks that require contextual understanding. Furthermore, Llama3 is designed to scale efficiently, allowing it to handle larger datasets and more complex tasks without a significant increase in computational requirements. This scalability is achieved through optimised training algorithms and efficient resource utilisation.

In contrast, **Mistral** introduces innovative attention mechanisms that focus on capturing nuanced linguistic patterns. It employs an optimized transformer architecture with innovations like sliding window attention and grouped-query attention. These mechanisms enable the model to generate highly creative and coherent text, making it suitable for applications that require a high degree of linguistic sophistication, while processing long sequences of text more efficiently and quickly. Mistral's architecture is characterised by a modular design that allows for greater flexibility and customisation.

8.3.2 Training Methodologies and Performance Benchmarks

Llama3 has been trained on a substantial and varied dataset, encompassing diverse domains such as literature, scientific papers, and news articles. This extensive training data has enabled the model to generate coherent and contextually relevant text across various genres and styles. Llama3 employs curriculum learning, a method that involves gradually increasing the complexity of the training data. This approach facilitates the model's learning process, enhancing its robustness and generalization capabilities. In addition, its proficiency across a broad spectrum of tasks is attributable to its extensive training on a substantial scale, encompassing over 15 trillion tokens of publicly available data. This extensive training has yielded state-of-the-art performance across various industry benchmarks and real-world scenarios, including enhanced reasoning abilities and code generation.[13]. Llama3 has been demonstrated to achieve high levels of accuracy in text generation tasks, producing coherent and contextually relevant outputs.

In contrast, **Mistral** has been demonstrated to excel in the generation of highly creative and sophisticated text, rendering it suitable for applications that require a high degree of linguistic nuance and innovation. Its training data has been meticulously curated to encompass specialized datasets that target specific domains and tasks, thereby ensuring the model's capacity to attain optimal levels of accuracy and relevance in designated applications. Mistral employs transfer learning to build upon the knowledge acquired by preceding models, thereby enhancing the model's

[13] https://www.hyperstack.cloud/blog/thought-leadership/all-you-need-to-know-about-llama-3

performance and efficiency, and enabling it to attain state-of-the-art results in various benchmarks. Mistral has been shown to excel in technical tasks, particularly in code generation and debugging. Its proficiency is evidenced by its high performance in benchmarks such as HumanEval and MBPP, which assess the comprehension and execution of instructions, thereby underscoring its value for precise technical tasks[14].

8.3.3 Model Sizes and Scalability

Llama3 is available in two sizes, 8 billion and 70 billion parameters, with each size having both pre-trained and instruction-tuned variants. This range allows for greater scalability in large-scale applications.

Mistral is designed to be more resource-efficient and suitable for environments with limited computational resources. However, it may not offer the same level of scalability for massive datasets or complex linguistic tasks.

8.3.4 Demonstration: decision and installation

While each model offers unique strengths and capabilities that cater to different applications and requirements, Llama3's optimized architecture and diverse training data make it a robust and efficient tool for tasks that require contextual understanding and coherence. On the other hand, Mistral's innovative attention mechanisms and specialized training data make it a versatile and adaptable tool for tasks that require linguistic sophistication and creativity.

As the field of natural language processing continues to evolve, the distinctions between Llama3 and Mistral highlight the importance of tailoring models to specific tasks and domains. By leveraging the strengths of each model, developers and researchers can achieve state-of-the-art results in a wide range of applications, driving innovation and progress in the field of AI.

For our purpose in PSI, we have chosen to analyse the possibility to auto-generate TM Forum ODA compliant documents such as the Conformance Profile document and the User Guide document based on Mistral. Based on the analysis presented above, Mistral offers better resource-efficiency for environments with limited computational resources as well as excellency in technical tasks in comparison to LLama3.

Although Mistral is suitable for resource-limited environments like e.g. a laptop, it is recommended to use GPU (Graphics Processing Unit) support. GPUs are designed with many processing cores and high memory bandwidth, making them well-suited for parallel processing tasks such as matrix operations – which are fundamental to AI computations. Offloading these tasks to the GPU allows to:

8.3.4.1 GPU support

In the PowerShell conducted as administrator, the command

```
nvidia-smi
```

allows to check for GPU support, assuming an NVIDIA GPU has been installed. If so, environment variables in Windows have to be adapted to use GPU support:

This will set Ollama to use GPU support while running LLMs.

[14] https://www.blockchain-council.org/ai/mistral-vs-llama-3/

8.3.4.2 Model installation

In a next step, the model Ollama should use is being installed. This can be done in a PowerShell without elevated rights:

```
ollama pull mistral
```

Finally, the command

```
ollama ps
```

will show if Ollama is using GPU support. The output should be similar as shown below.

```
> ollama ps
NAME                                         ID              SIZE
PROCESSOR        UNTIL

Maoyue/mistral-nemo-instruct-2407:latest    3611dcdcc6b0    9.1 GB
42%/58% CPU/GPU    9 seconds from now
```

Figure 8.1: Ollama using GPU support, command output.

8.3.4.3 Open WebUI

Open WebUI[15] is a user-friendly, open-source web interface that facilitates interaction with AI models such as Ollama. It enables the execution of text generation tasks, the fine-tuning of models, and the management of local AI configuration seamlessly through a web browser. It requires Python3.11 to be installed. Again, Python3.11 can be installed using Chocolatey and PowerShell with elevated permissions:

```
choco install python311
```

Additionally, the *uv* package should be installed. *uv* is a Python package manager that simplifies dependency management, making it easy to install, upgrade, and manage packages in your projects. It aims to provide a more user-friendly experience compared to traditional tools like pip, with features such as automatic resolution of dependencies and support for multiple Python versions:

```
choco install uv
```

To finally start Open WebUI, command in a PowerShell without elevated rights:

```
$env:DATA_DIR="C:\Users\<user.name>\Documents\open-webui\data"; uvx --python 3.11 open-webui@latest serve
     --port 8080
```

with `<user.name>` being the Windows user name for your account. Open WebUI will now be available at *localhost:8080* in your browser.

[15] https://openwebui.com/

8.4 Applying Mistral to generate TM Forum OAS mandatory documents

As previously mentioned, TM Forum hosts within the ODA framework several API documentation files, representing full definitions of APIs. Those are embedded within the Open API Project in TM Forum. This project constitutes a collaborative initiative led by TM Forum with the objective of developing standardised APIs for the telecommunications industry. The aim of the Project is to create Open APIs that enable seamless connectivity, interoperability, and portability across complex ecosystem services in the digital services value chain. All TM Forum members, including the world's largest service providers and suppliers, work together to develop these APIs. As of March 2025, TM Forum provides more than 60 Open APIs for IT transformation and end-to-end management of complex digital services. These REST-based APIs are technology agnostic and can be used in various digital service scenarios, including IoT, smart cities, and network virtualisation. The APIs are created using a crowdsourcing approach, with members encouraged to contribute extensions and enhancements. In February 2025, TM Forum launched its Gen5 Open APIs with the objective of supporting event-driven architectures, providing a simpler developer experience, and enabling intent-based automation.

To publish an API within the Open API framework, several steps and requirements must be followed[16]. Amongst those steps are the swagger file development and the creation of the API User Guide and the Conformance Profile Document. Once the API has been accepted for publication, these files will be made available for download to the TM Forum community.

For our purpose, we want to apply the LLM Mistral 7B model to understand how the LLM supported generation of the User Guide and the Conformance Document could work, if provided with the OAS file as schema.

8.4.1 Test setup

In order to enable Mistral understanding the features of the documents, we supplied the User Guide and Conformance documents from APIs TMF620 (Product Catalog Management)[17], TMF621 (Trouble Ticket API)[18] and TMF632 (Party Management)[19], all v5.0, to be analyzed by Mistral. As benchmark, we used TMF653 (Service Test Management)[20] documentation to understand in which detail and how accurate Mistral would be able to create those documents based on the according OAS schema file.

The OAS schema file as presented online in YAML cannot be natively parsed by Mistral. Thus, we reformatted the native YAML file to JSON.

Generally, the documentation from the three APIs have been given to Mistral as background information, the OAS file as content and the corresponding PDF documents for the benchmark API TMF653 as ground truth.

8.4.2 Auto-generating Conformance documentation

Mistral was asked first to analyze the provided PDF Conformance documents for their structure and similarities:

```
Prompt: "Please show me all passages of the uploaded pdf files that are the same, either in context or
    verbatim."
```

[16] https://www.tmforum.org/playbook/propose-a-new-api-or-api-suite/
[17] https://www.tmforum.org/oda/open-apis/directory/resource-inventory-management-api-TMF620
[18] https://www.tmforum.org/oda/open-apis/directory/resource-inventory-management-api-TMF621
[19] https://www.tmforum.org/oda/open-apis/directory/resource-inventory-management-api-TMF632
[20] https://www.tmforum.org/oda/open-apis/directory/resource-inventory-management-api-TMF632

As result, Mistral correctly pointed out the *Copyright Notice and Permissions* section to be identical. The *Table of Contents* was denoted as similar, but not equal, while the *Introduction* section was analyzed to be content wise similar:

```
Each document has an "Introduction - API description" section that describes the purpose and
    functionality of the API covered by the document. While the specific details differ, the structure
    and intent of this section are consistent.
```

The same observation was made for *Resource Model Conformance*, *API Operations Conformance*, *API GET/-POST/PATCH/DELETE Operation Conformance*, *Acknowledgements and Release History* and *Mandatory Attributes Tables*:

```
The sections titled "Resource Model Conformance" and "General Notes on Resource Attribute Conformance"
    are structured similarly across the documents. They explain the mandatory resources and attributes
    for the respective APIs.
Each document includes a section on "API Operations Conformance," which lists the mandatory operations
    for the API resources. The structure and format of these tables are consistent across the documents.
The documents detail the conformance for GET, POST, PATCH, and DELETE operations, including status codes
    and mandatory attributes. The format and approach to describing these operations are similar.
The "Acknowledgements" and "Release History" sections are structured similarly, although the specific
    content may vary.
The tables listing mandatory attributes for different resources (e.g., TroubleTicket, Individual,
    Organization, ProductOffering) are formatted similarly, with columns for the attribute name and
    conformance rules.
These commonalities indicate a standardized approach to documenting API conformance profiles across
    different specifications within the TM Forum.
```

As a result from this analysis, the *Copyright Notice and Permissions* could be included as text building block in any conformance document.

Secondly, Mistral was asked to generate an introduction to the API TMF653, based on the OAS schema:

```
Prompt: "Following the style in the given pdf files, formulate an introduction to the API given in the
    uploaded json."
```

Mistral (A) was able to write a very precise introduction, outlining the API features and handlings, while the bench-mark introduction was much shorter and did not provide any details (B):

```
(A) The following document is the specification of the REST API for Service Test Management. This API
    provides a structured approach to managing the testing of services, allowing for the definition,
    execution, and monitoring of test cases and test suites. The API supports operations to create,
    retrieve, update, and delete test-related entities, enabling comprehensive test management
    capabilities.
The Service Test Management API is designed to facilitate the integration of testing processes within
    service-oriented architectures, ensuring that services meet quality and performance standards. It
    includes resources and operations to handle test plans, test cases, test results, and related
    metadata, supporting automated and manual testing workflows.
This API adheres to the principles of RESTful design, utilizing standard HTTP methods and providing a
    clear and consistent interface for interacting with test management data. The specifications outlined
     in this document detail the mandatory resources, attributes, and operations necessary for
    conformance, ensuring interoperability and ease of integration with other systems and services.

(B) The following document is the conformance specification of the REST API for Any management.
```

Given this result, it can be concluded that Mistral is capable of writing a very good introduction based on the provided OAS schema file.

Next, Mistral was tasked to generate chapter *Resource Model Conformance*. The prompt had to be defined very precisely, as Mistral tended to include too many resource definitions compared to the benchmark document:

```
Prompt: "Based on the provided json file, write a section for "Resouce Model Conformance" following the
    styles in the pdf files. Include only the resources directly related to the core functionality of the
     API as mandatory."
```

The text generated by Mistral was a 100% match with the benchmark text, proving that Mistral could be applied to generate this section of the document as well.

The next chapters - *API Operations Conformance*, *API GET/POST/PATCH/DELETE Operation Conformance*, *Acknowledgements and Release History* and *Mandatory Attributes Tables*, were prompted likewise. However, Mistral was not able to generate document sections that matched the benchmark document save for subsection *Notification Model Conformance* with a 100% match. We found that several attributes were constantly missing, or even wrong assessments. Thus, PATCH and DELETE, marked as non-mandatory operations in the benchmark pdf, were listed by Mistral as mandatory and added. Remarkably though, if provided with enough detail, Mistral was able to generate a 100% match for subchapter *API GET operation conformance* using the prompt

```
Prompt: Following the schema presented in the pdf files, write the "API GET operation conformance" based
    on the provided json file.
```

Thus, Mistral seems to be able to generate those sections if the prompts are precise enough.

8.4.2.1 Overall assessment

Mistral exceedes in generating the introduction and the *Resource Model Conformance* chapter with fairly easy prompts. Also, Mistral is able to generate subchapters if the prompts are precise and written with knowledge about the API, as embedded in the OAS schema file. However, Mistral is not capable of generating documentation for the other chapters; information was found to be missing or, worse, to be wrong.

8.4.3 Auto-generating Userguide documentation

Again, Mistral was asked first to analyze the provided pdf documents for their structure and similarities:

```
Prompt: "Please show me all passages of the three uploaded pdf files that are the same, either in context
     or verbatim."
```

Mistral identified all chapters to be identical from context and also peculiarities of each chapter, e.g.

```
API Operations:
All documents describe similar API operations such as retrieving, listing, creating, updating, and
    deleting entities. The operations follow a standard RESTful approach with GET, POST, PATCH, and
    DELETE methods.
The descriptions of these operations are similar in terms of how they are executed and what they achieve.
```

As common sections, Mistral identified chapters *Notice*, *Table of Contents* and *Acknowledgements* to be

```
similar in format and content across all documents, as they are standard sections in technical
    specifications.
```

Thus, the structure of the Userguide documents was described and well analysed from content perspective. Mistral was able to identify the intention of each chapter.

Next, Mistral was asked to write the *Introduction* for the Service Test Management Userguide:

```
Prompt: Write the section "Introduction" based on the provides json file and following the style
    identified in the provided pdf files.
```

Mistral wrote an introduction that strongly differed from the benchmark document. Thus, the request was refined using the benchmark introduction as input as well to understand the difference better:

```
Prompt: Compare your answer to "The following document is the specification of the REST API for Service
    Test Management. It includes the model definition as well as all available operations. Possible
    actions are creating, updating and retrieving Service Test. The Service Test API provides a
    standardized mechanism for placing a service  test with all the necessary test parameters. The API
    consists of a simple set of operations that interact with CRM/Service Management systems in a
    consistent manner. A service test is a procedure intended to check the quality, performance, or
    reliability of a service. Service Test Management API manages service test resource and  service
    specification test resource:
 A service test specification describes the service test in terms of parameters to be
configured and measures to be taken.
 A service exists for a controlled test invocation on a service. The service test is executed
according to a schedule. and contains service test configuration parameters that are to
be applied at execution time, and service test measures that result."
```

Mistral outlined the strenghts and weeknesses of its own produced text and the benchmark text and suggested a combined introduction text.

```
My Proposed Introduction:
Strengths:
Clearly states the purpose of the API: managing and executing service tests.
Lists the main resources (ServiceTest and ServiceTestSpecification) and operations (Retrieve, Partial
    Update, Create, Delete, Notification Management).
Provides a structured overview, similar to the style in the provided PDF files.
Weaknesses:
Does not explicitly mention interaction with CRM/Service Management systems.
Lacks specific details about the nature of a service test (e.g., checking quality, performance, or
    reliability).
Does not elaborate on the execution of service tests according to a schedule.

Provided Introduction:
Strengths:
Provides a detailed explanation of what a service test is and its purpose.
Mentions interaction with CRM/Service Management systems, which gives context to the API’s usage.
Describes the resources (service test and service test specification) in more detail, including execution
     parameters and measures.
Explains the scheduling aspect of service tests.
Weaknesses:
The structure is less concise and could benefit from clearer sectioning for readability.
Does not explicitly list all operations (e.g., Partial Update, Notification Management) as my version
    does.

Combined and Improved Version:
Introduction - API description
The following document is the specification of the REST API for Service Test Management. It includes the
    model definition as well as all available operations. The Service Test Management API aims to provide
     a standardized mechanism for managing and executing service tests, ensuring quality, performance,
    and reliability assessments of provisioned services.

Resources
ServiceTest: Represents a controlled test invocation on a service, executed according to a schedule. It
    includes configuration parameters applied at execution time and measures resulting from the test.
ServiceTestSpecification: Describes the service test in terms of parameters to be configured and measures
     to be taken.
Operations
```

```
The Service Test Management API performs the following operations on the resources:

Retrieve: Allows retrieval of an entity or a collection of entities based on specified filter criteria.
Partial Update: Enables updating parts of an entity, including updating rules.
Create: Facilitates the creation of an entity, including default values and creation rules.
Delete: Allows deletion of an entity for administrative purposes.
Notification Management: Manages notifications of events related to service tests.
This API interacts with CRM/Service Management systems in a consistent manner, providing a comprehensive
    solution for service test management.
```

These observations suggest that Mistral bases the text solely on the information provided in the json schema file. As this file did not include any meta information on, e.g., the nature of the service test, Mistral could not derive the information as given in the benchmark document. Putatively, Mistral is able to write an introduction similar to the one given in the benchmark document if these meta informations are given in the provided json schema file.

Subsequently, Mistral was asked to write chapters *Managed Entity and Task Resource Models*, *Notification Resource Models* and *API Operations* with prompts like

```
Prompt: Based on the information provided in the json file, write chapter "API Operations" for resources
    directly related to the core functionality of the API.
```

However, Mistral was not able to write those chapters.

Mistral was further asked to write the chapter *Sample Use cases* based on the provided json schema file:

```
Prompt: Based on the style identified in the pdfs and the content on the provided json file, write
    chapter "Sample Use cases" giving at least one use case per resource diretly related to the core
    functionality of the API in the json file.
```

Mistral identified use cases with sufficient granularity of detail, e.g.

```
Use Case 1: Managing a Service
Scenario
A service provider wants to create, update, and manage a new service offering within their system. The
    service should include details such as name, description, status, and associated tags. The provider
    also wants to ensure that the service can be retrieved and updated efficiently.

Steps
Create a Service: Use the POST /services operation to create a new service with details such as name,
    description, status, and tags.
Retrieve a Service: Use the GET /services/{serviceId} operation to retrieve the details of the newly
    created service.
Update a Service: Use the PATCH /services/{serviceId} operation to update specific attributes of the
    service, such as changing its status or description.
Delete a Service: Use the DELETE /services/{serviceId} operation to delete the service if it is no longer
     needed.
Expected Outcome
The service provider successfully manages the lifecycle of the service, from creation to potential
    deletion, ensuring that all relevant details are accurately maintained and accessible.
```

The suggestions provided by Mistral are very good for demonstrating the API's capabilities. However, the resources identified do not match the benchmark document. As such, the POST /services resource is not presented in the benchmark document.

Lastly, Mistral was asked to write chapter *Release History* based on the json schema file.

```
Prompt: Write the Release History for the API described in the json file.
```

Mistral's answer includes several dates as provided in the json file, but does not match the benchmark document:

```
The API has undergone several iterations and updates to enhance its functionality, improve performance,
    and expand its capabilities. Below is a chronological overview of the key releases and the
    enhancements introduced in each version.

Version 1.0.0
Release Date: October 1, 2023
Features and Enhancements:
Initial Release: The first version of the API was released, providing basic functionality for managing
    services, test cases, and components.
Core Operations: Introduced core operations for creating, retrieving, updating, and deleting services and
     test cases.
Resource Models: Established the foundational resource models for services, test cases, and components,
    including their attributes and relationships.
Version 1.1.0

Release Date: November 15, 2023
Features and Enhancements:

Enhanced Test Case Execution: Improved the functionality for executing test cases, including more
    detailed reporting of test results and execution status.
Component Integration: Enhanced the integration of reusable components across different services and test
     cases, allowing for more efficient utilization of common functionalities.
Bug Fixes: Addressed several bugs reported by early users, improving the stability and reliability of the
     API.
Version 1.2.0
Release Date: December 1, 2023
Features and Enhancements:

Expanded Service Management: Introduced additional attributes and relationships for services, allowing
    for more detailed and nuanced management of service offerings.
Performance Optimizations: Implemented performance optimizations to improve the speed and efficiency of
    API operations, particularly for large datasets.
Security Enhancements: Enhanced security measures to protect user data and ensure compliance with
    industry standards.
Version 1.3.0
Release Date: January 15, 2024
Features and Enhancements:

Advanced Analytics: Introduced advanced analytics features to provide deeper insights into service
    performance and test case results.
User Interface Improvements: Enhanced the user interface for better usability and user experience,
    including improved navigation and visualization tools.
Documentation Updates: Updated API documentation to reflect the latest features and provide more
    comprehensive guidance for developers.
```

Again, this might be due to the benchmark details not being presented in the json file. The dates however vaguely match those presented in the benchmark document.

8.4.3.1 Overall assessment

While Mistral exceeded in creating content for the Conformance document, it does not for the Userguide document. Based on the above presented results, not enough detail seemed to be given by presenting the json file solely. Putatively, Mistral might be capable of writing the chapters for the Userguide document if meta information is included in the json file. Potentially, supplying additional documents like e.g. Use Case scenarios might enable Mistral to write those chapters with enough detail.

8.4.4 Conclusions

Mistral was able to write some chapters for the Conformance documents, while it struggled with content creation for the Userguide document. These results might reflect the lack of meta information to be included in the json schema file. In addition, Mistral summarised wrong information in some chapters. This might be due to misleading information in the provided json file or the misinterpretation of the LLM. Overall, it seems like the information provided in the schema file was not enough to create documents matching the detail level that was found in the benchmarking documents.

Based on these results, several options might be further analysed to apply LLMs in an automated setup:

a) Hybrid approach

An LLM - Mistral - could be applied in an automated way using the suggested prompt to write the introduction for each document, given that enough meta information is being presented in the json schema file. Sections that are the same in wording could be used as fixed building blocks. Sections that are content wise very schematical, e.g. the *API Operations* chapters, can be derived by using a parser functionality.

The pipeline could then include fixed building blocks of text (e.g., for the *Copyright Notice and Permissions* chapter of the Conformance document), LLM-generated text as for e.g. the *Introduction* and parser generated text as is already done in our ICD documentation. This approach however might not be suitable for both the Conformance and the Userguide documents, thus including option b) might be necessary.

b) Training an own LLM

Mistral showed partially very good results, but missed some sections or chapters entirely. This could be counteracted by training an own instance of Mistral, providing further documentation and thus adapt the LLM to own needs. This model could then be applied to generate each section based on the provided json scheme. Additionally, documentation that is specific for an API might be beneficial to the overall quality of generated documentation. For example, use case documentation, whether derived by the customer or by the development team, may need to be included as background information for the LLM to produce good quality documentation.

Still, some chapters might lack information and the approach using a parser to generate specific sections as in option a) might be needed.

Potentially, a combination of both options might lead to the desired results. The envisioned pipeline could then include for documentation generation building blocks for static text, a bespoke LLM triggered with specific prompts generating e.g. the introduction or use case scenarios, and a parser generating sections that build on the provided schema file(s).

9 Automation of document generation - Conclusion

We have shown how PSI uses Documentation as Code within its repository. Furthermore, we have elaborated how requirements are tracked and traced in an automated way as well. In addition, API schema files being developed within PSI are transformed to match TM Forum's OAS file definitions in an automated way. Lastly, we have assessed how LLMs, in Mistral flavor, can be used to automatically generate TM Forum's Conformance and Userguide documents based on the provided OAS file.

Thus, the integration of documentation as code, automated requirements tracing, and document generation via LLM within a pipeline can potentially engender a multitude of advantages that serve to streamline and enhance the software development process. Primarily, the quality of documentation is significantly enhanced. DaC fosters high-quality, consistent documentation that remains in alignment with the codebase, thereby reducing the manual effort required from developers and ensuring that documentation is perpetually current and accurate. Furthermore, the real-time nature of the pipeline is a significant advantage. The pipeline ensures that documentation is updated in real-time with every code change, which is particularly beneficial in fast-paced development environments, meaning that the documentation is always up-to-date and reflects the current state of the codebase. Another salient benefit is traceability. By automating the linkage of code changes to specific requirements, the pipeline ensures that every piece of code can be traced back to its origin. This is crucial for compliance, auditing and maintaining software quality, as it provides a clear and trackable history of changes. Additionally, the reduced human error resulting from automating documentation and requirements tracing is a substantial benefit, leading to more accurate and reliable documentation and traceability matrices, and enhancing the overall quality and reliability. With documentation and requirements tracing integrated into a CI/CD pipeline, developers can focus on writing code while the pipeline handles the rest. This reduction in overhead, due to the elimination of manual documentation and traceability tasks, enables a more efficient and productive workflow.

Assuming an LLM is rendered capable of handling the auto-generation of documents as suggested in the previous chapter, a significant benefit from such a workflow can be scalability. As projects grow, the LLM can continue to generate and maintain documentation without additional manual effort, ensuring that the documentation process scales with the codebase. This makes it suitable for projects of any size.

In summary, introducing a pipeline including DaC, automated requirements tracing and LLMs for generating mandatory documents for ODA publication enhances the overall quality and reliability of the software development process. On top, the inclusion of OAS automated transformation allows to easily publish TM Forum compliant APIs. Thus, this pipeline offers a comprehensive solution that improves documentation quality, enhances traceability, streamlines the development process, scales with the project, provides real-time updates, and reduces human error. This approach is a potent one, with the potential to markedly enhance the efficiency and effectiveness of the software development process.

Last Page of Document