Documentation as Code in Automotive System/Software Engineering

Momcilo V. Krunic
Department of Computer Engineering and Communications, Faculty of Technical Sciences,
University of Novi Sad,
Trg Dositeja Obradovica 6, Novi Sad 106314, Serbia
momcilo.krunic@uns.ac.rs

***Abstract*—Documentation as Code (DaC) is an approach that applies the principles of software development to the production of technical documentation. Using modern tools, DaC enables software engineers to treat documentation as a first-class citizen in the development process, alongside code and tests. In this paper, we discuss the advantages of DaC in system and software engineering, including improved accuracy, traceability, and maintainability. In the automotive industry, DaC has been used to document various aspects of vehicle development, such as requirements, design, testing, and compliance. This paper provides an overview of the state-of-the-art in DaC in the automotive industry and discusses the potential benefits and challenges of using this approach. In addition, case studies and examples of how DaC has been used in the automotive industry to improve the quality and maintainability of documentation are presented. This research has been conducted with more than 150 engineers actively contributing to DaC on the project for more than a year within a company, so the scalability of the presented solution has been tested. Finally, a set of guidelines is provided for teams to follow when adopting DaC to ensure successful implementation.**

***Index Terms*—Automotive engineering; Documentation; Software engineering; Software systems.**

I. INTRODUCTION

The automotive industry is under increasing pressure to improve the quality and efficiency of vehicle software development. One approach that has been gaining popularity in recent years is Documentation as Code (DaC), which treats documentation as a first-class citizen in the development process, alongside code and tests. The main idea behind DaC is to make documentation more accessible, maintainable, and up-to-date by storing it in the same repository as the code and using the same tools for version control, collaboration, and continuous delivery.

DaC has been applied in various domains, such as application development, IT, and web development. However, its application in the automotive industry is still in its infancy. The automotive industry has unique requirements and constraints, such as safety, cybersecurity, and compliance with standards such as ASPICE [1], [2], which stands for Automotive SPICE (Software Process Improvement and Capability dEtermination), which make it challenging to apply DaC. Furthermore, the automotive industry has a long product lifecycle and requires maintaining documentation for a longer period.

This paper provides an overview of the state-of-the-art in DaC in the automotive industry and discusses the potential benefits and challenges of using this approach. The paper will also present case studies and examples of how DaC has been used in the automotive industry to improve the quality and maintainability of documentation. This paper will be of interest to researchers, practitioners, and professionals in the automotive industry who are looking for ways to improve the quality and efficiency of vehicle software development.

Also, this paper discusses the use of DaC in compliance with the automotive standard ASPICE and the V-model. The ASPICE standard is a widely used framework for evaluating and improving the quality of automotive software development processes. The V-model, on the other hand, is a widely used software development model that describes the various phases of a project lifecycle and the relationships between them.

By integrating DaC practises into automotive system/software engineering, we can ensure that the documentation produced during the development process is accurate, consistent, and up-to-date. This can be achieved by using version control systems, such as Git, to manage the documentation and by using automated tools to check the documentation for errors and inconsistencies. Additionally, by using the V-model, we can ensure that the documentation is produced in the appropriate phase of the project and is aligned with the requirements and design of the system.

The paper concludes that by using DaC practises in conjunction with the ASPICE standard and the V-model, we can improve the quality of automotive system/software engineering and ensure that the documentation produced is accurate, consistent, and up-to-date, as well as accessible and easy to use.

The rest of the paper is organised as follows. The next section provides a brief overview of DaC and its benefits. Then, the paper will present the processes and tools used in the case study, the research that inspired the writing of this paper, and examples of how DaC has been used in the automotive industry. Finally, the paper will conclude with a discussion of the potential benefits and challenges of using DaC in the automotive industry and future research directions.

Manuscript received 5 February, 2023; accepted 14 April, 2023.

II. OVERVIEW

Documentation as Code (DaC) is a holistic approach to documenting technical information that can be applied to the development of technical documentation in automotive engineering. By applying DaC, this research explores its ability to improve accuracy, traceability, maintainability, accessibility, and utilisation.

The benefits of using DaC are considerable; it allows automotive engineers to author technical documentation faster with more precision and less overhead cost. Automated processes, such as automated builds or continuous integration pipelines, can be used to create documentation from source files and to send changes to production systems quickly and reliably. Moreover, incorporating version control to track document changes helps automotive engineers identify and address problems more quickly. Automated testing can also be used to validate the accuracy of documentation before it is released to production.

The art of documenting computer programmes has evolved significantly in the past few years. Today, many different tools and techniques are used to produce high-quality technical documentation. Some key trends and practises currently considered state-of-the-art in DaC include the following.

– *Use of Markdown and other lightweight markup languages*: DaC often relies on storing documentation in plain text files that can be version-controlled, reviewed, and rendered as HTML, PDF, or other formats. Markdown is a popular format for this purpose, as it is easy to read and write and can be converted to other formats using a variety of tools. The research presented in this paper used Markedly Structured Text (MyST) [3] Markdown flavor as the language of choice for documenting technical information.

– *Automated documentation generation*: DaC often uses tools and scripts to automatically generate documentation from code, comments, tests, and other sources, such as models or artifacts of design. This helps to ensure that documentation is accurate and up-to-date with the code, and can reduce the effort required to maintain it. This case study utilised the Sphinx [4] framework for automated code documentation generation as part of the continuous delivery [5] process.

– *Use of version control systems*: DaC relies on version control systems to manage and track changes to documentation, just like code. This allows collaboration, review, and rollback of changes, and enables the traceability of documentation to specific versions of code. This is one of the key enablers of the DaC, since it ensures that all software development artifacts are stored and released together. This greatly simplifies forensics since reproducibility is embedded in the system design. The research presented in this paper uses the Git version control system for managing all relevant artifacts: documentation, source code, tests, test results, and configuration files.

– *Use of model-driven development*: DaC often uses model-driven development (MDD) approaches, where documentation is generated automatically from models of the system, and the documentation is kept in sync with the model, making it more accurate and up-to-date. In this research, the C4 architecture model [6] has been used to describe the system on various levels of abstraction: system Context, Containers, Components, and Code.

– *Adoption of DevOps practises to enable the continuous delivery process* [5]: DaC often follows the DevOps principles, which emphasise continuous integration and delivery, collaboration, and automation, enabling fast feedback loops and transparency. This provides an opportunity to react as soon as the problem occurs, which makes it much cheaper and easier to resolve. More details about the DevOps tooling landscape used in this research are provided in Section III (“Processes and Tools”).

III. PROCESSES AND TOOLS

To make the most of the DaC approach in the automotive industry, it is essential to have processes and tools in place that support its use. This includes process guidelines, source control systems, collaboration tools, and CI/CD servers, used for managing documentation alongside code and other artifacts. Automated methods should be used to generate documentation from models of the system and to validate various aspects of the generated documentation. Furthermore, traceability and consistency between requirements, design elements, source code, and tests is vital, hence the need for a well-designed synergy between processes and tools. Automation ensures that documentation is accurate and up-to-date with changes in the system.

A. Processes

One of the conditions of the case study used in this research was the Automotive SPICE (ASPICE) standard Level 02 (managed process) requirements Fig. 1. ASPICE is a process assessment model tailored to the automotive industry. It is based on the ISO 15504 (SPICE) standard and provides a structure to evaluate and enhance the software development process in the automotive industry. ASPICE is used in automobile system and software engineering to help automotive suppliers meet the expectations of original equipment manufacturers (OEM). In this research, it has been utilised as the main process guideline/requirement for the implementation of DaC.

The ASPICE process model and the V-model are two widely used models in the automotive industry for software development. The V-model is a graphical representation of the development process, showing the relationships between different stages such as requirements, design, implementation, and testing. ASPICE provides recommended practises and guidelines to assess the current state of the software development process and identify areas for improvement. When used together (Fig. 2), these two models can help ensure that software development is efficient, effective, and safe, thus improving the quality and safety of software development in the automotive industry.

It is essential to note that in this research, feature teams have been structured according to Agile Scrum practises. As such, Sprint was the organisational cycle in which feature teams arranged their work. The regular process would assume that the Sprint planning feature team would agree with the customer about the scope for the following Sprint,

using the System Architectural Design - SysAD (SYS.3) as a Project backlog. Then the set of input requirements from the SysAD is decomposed into User Stories. The User Story would be viewed as a software requirement to be consistent with ASPICE. Also, one User Story can be treated as an Software Engineering (SWE) Group V-model package (Fig. 2). What that implies practically is that User Story cannot be considered finished before all SWE.1-6 artifacts are created or generated. To meet this quality requirement, and still be agile, feature teams tailored User Stories so they can be delivered in just one Sprint, by executing so-called *micro-V* cycles. This is the place where DaC was a key empowering factor and without which this dynamic would not be possible, or it would be simply highly inefficient due to context switching. Treating documentation as code one can simply update what is necessary or create new content, without leaving the integrated development environment (IDE). These tasks should be considered alongside functionality when tailoring and planning User Stories. Using *micro-V* cycles, quality is embedded into the released software ground up, brick by brick (Sprint by Sprint), where User Story cannot be merged into the main branch if the whole package (SWE.1-6) is not wrapped up.

It is worth noting that the OEM defines SysAD, but the feature team can suggest modifications when they find a better design or demonstrate that the existing one is not feasible.

After going through several rounds of internal audits with the Quality Assurance Department, the DaC implementation used in this research, developed incrementally and iteratively executing micro-V cycles, was found to meet all the Base Practises set out for Software Engineering Group Level 2. This was a significant achievement, as it reassured management to adopt DaC practises throughout the organisation.

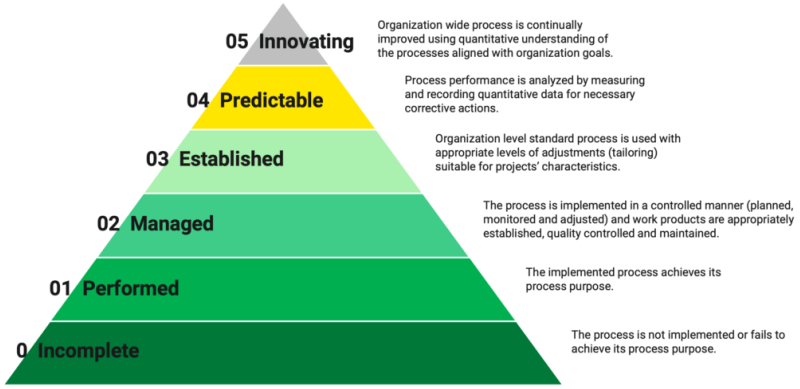

Fig. 1. ASPICE five capability levels.

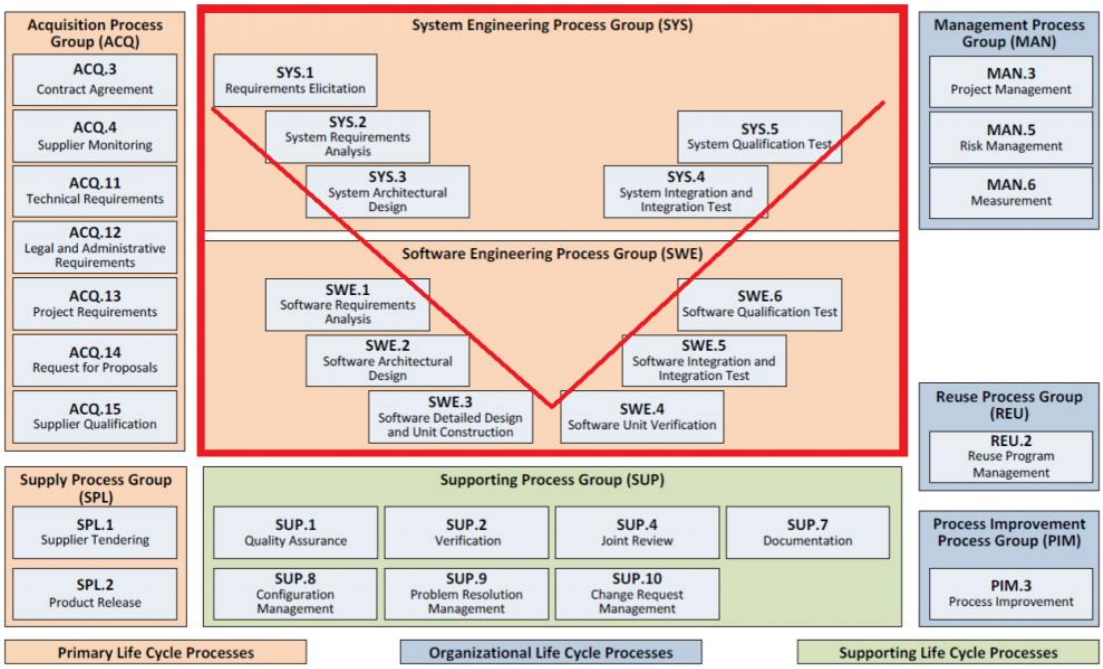

Fig. 2. Organisation of ASPICE V-model.

B. Tools

The DaC methodology involves using a range of tools to facilitate different stages of the software development process, including application lifecycle management (ALM) to track progress, documentation generation for accurate and up-to-date records, source control management to keep versions organised, CI/CD for automated delivery, and custom microtooling to streamline tasks. We used these tools alongside an existing tool (Windchill) to ensure

backward compatibility with the exchange of requirements using the ReqIF format (Fig. 3). This portion of the system requires improvement both in the process and in the tools.

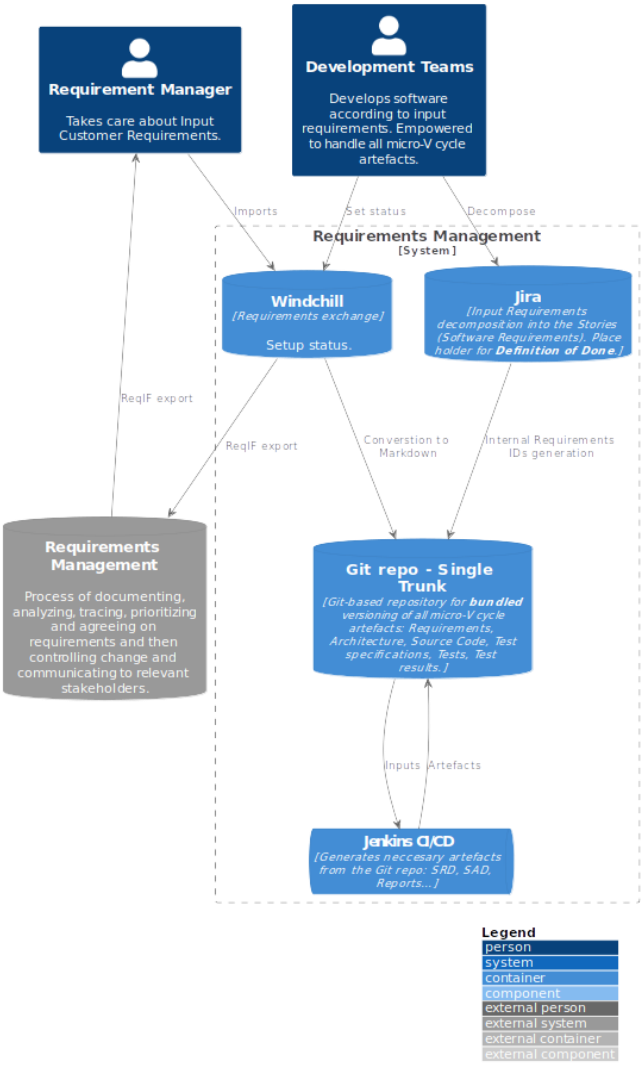

Fig. 3. Requirements exchange process between the OEM and the Tier 1 software supplier.

As can be seen from Fig. 3, the software development process starts once input requirements have been received from the customer in the ReqIF format, in the Windchill tool. This is part of the legacy process, which is unfortunately still part of the software lifecycle management. The author of this study find this to be one of the hindrances to Agile practises in the automotive industry, and it is something that needs to be changed to optimise the software development process and enable continuous delivery. In reality, this step is not followed strictly, and the feature teams find alternate means of communicating directly with the customer and breaking down the problem, rather than passing the ReqIF back and forth. Direct communication with the customer should always be the preferred process, rather than a workaround.

In this study, Jira was used as an application lifecycle management (ALM) tool, but it was also used as a process guideline. Since the proposed system was designed to be team-focused to reduce context switching between different tools and environments, it was observed that the ALM tool could also be used as a process framework. Entities of the ALM tool (Capabilities, Features, Epics, etc.) were used as placeholders for the process definition in the form of a Definition of Done (DoD). The DoD was versioned and stored in the Git repository together with other artifacts. When a feature team starts to work on a new Capability, it will clone the template and the entire structure illustrated in Fig. 4, which serves a dual purpose: artifact lifecycle management and process guideline/framework. It should be noted that Jira can be replaced by any other ALM tool, such as Redmine, Codebeamer, Polarion, etc. Moreover, it is important to recognise the clear relationship between the structure shown in Fig. 4 and the SWE group in the ASPICE V-model, Fig. 2. This is an example of how the process and the tool can be combined to streamline the development process and increase the chances of consistently following the process.

Version control is a crucial component of the approach to the DaC system. For this research, Git was the obvious choice. It is a state-of-the-art version control system and a reliable storage solution. This decision was made because Git had successfully met the needs of versioning and storing the only artifact that brings value to the customer - working software. This strategy simplifies the whole continuous delivery process. When all artifacts related to the release process are stored and versioned in the same place, it becomes much easier to perform automated validations by the CI/CD server before delivering the software to customers, resulting in a better quality of the final product and higher customer satisfaction. Additionally, it is much easier to perform forensics when bugs are found. By simply checking out the released Git repository, all necessary information is available for an investigation into the particular release, including source code, test results, architecture, etc. Furthermore, in this research, it has been demonstrated that adopting a trunk-based development approach [7] is essential for the continuous successful delivery of artifacts, working software and documentation.

Continuous delivery [5] is a process that enables fast feedback loops from customer to the feature teams. This is essential for optimal steering of the software development process and discovering problems in the early stage. Besides the version control system, this CI/CD is the second most important component of the proposed DaC system. Automated builds were used to validate the code and documentation, compile it, and run tests. Automated deployments of documentation and software were also performed. Automating these processes is essential to avoid manual interventions and human errors. Automated builds, tests, and deployments are the core components of a continuous delivery pipeline, which can drastically improve the quality of software and documentation delivered to customers. In this research, the Jenkins CI/CD build server has been used for both continuous delivery pipelines: Software (see the figure in the following section) and documentation.

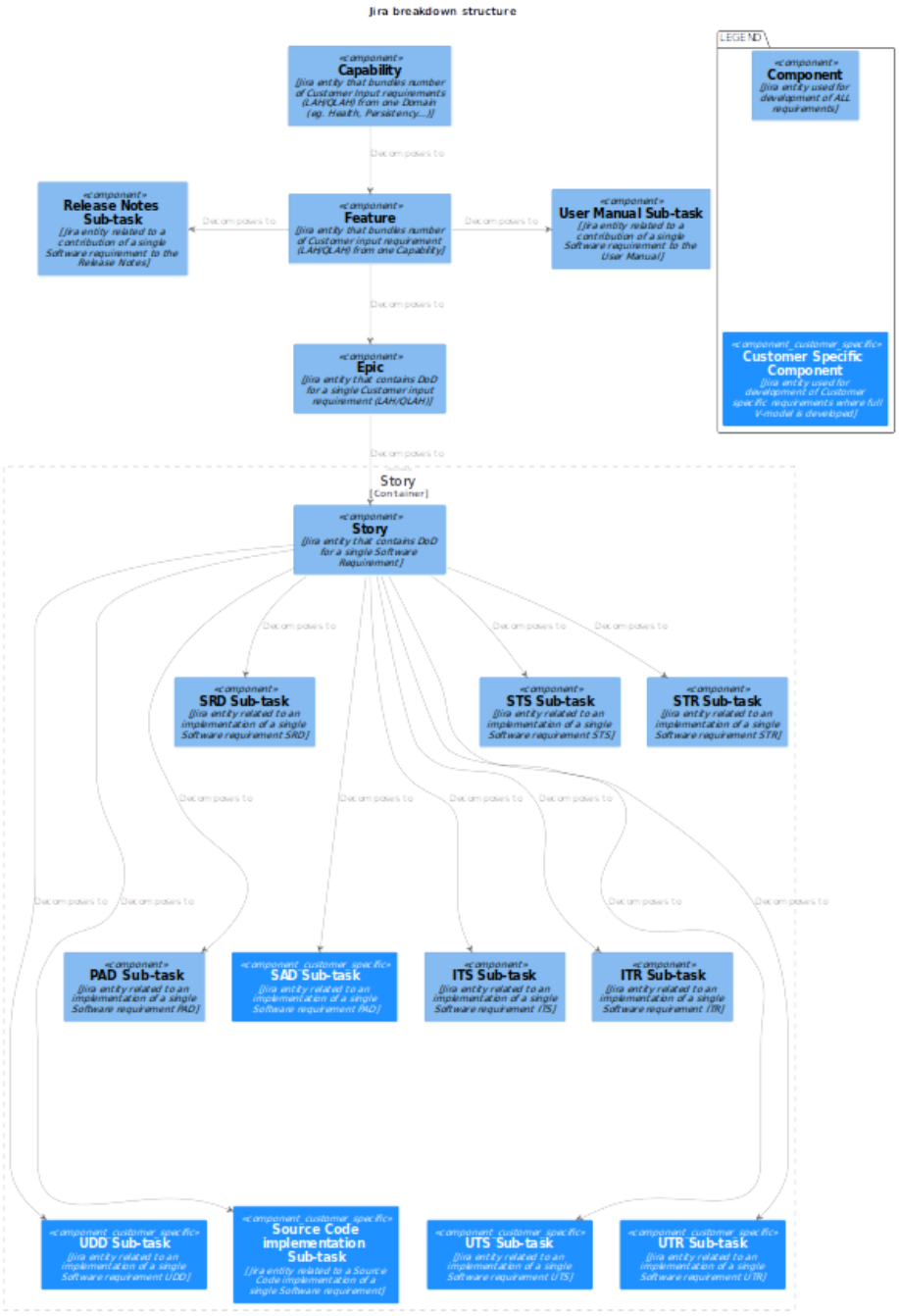

Fig. 4. Jira as a process guideline.

The continuous delivery pipeline for DaC has been divided into CI and CD pipelines to optimise the entire process. The Documentation CI (see the figure in the following section) pipeline is triggered on every pull request (PR) update. Its primary purpose is to keep all architectural diagrams up-to-date, as well as to serve as a quality gatekeeper and to prevent broken diagrams, links, etc. from being merged into the main branch.

Whenever a PR is merged to the main branch, the Documentation CD pipeline (see the figure in the following section) is launched. This pipeline performs additional verifications, builds the documentation, and deploys the documentation as a static website to the designated documentation server.

The main reason for breaking the DaC continuous delivery pipeline into two separate pipelines is execution time. The DaC CI pipeline needs to be as fast as possible (execution time <2 min), as it serves as a gatekeeper to prevent PRs from merging if something goes wrong. On the other hand, DaC CD pipeline does not need to be as dynamic (execution time >30 min) since one can survive with outdated documentation for a half hour.

The selection of a language for writing technical documentation is an important part of DaC system design. Markdown language [8] was chosen for this case study for several reasons: it is lightweight and does not require any prior knowledge, it is portable across different operating systems and editors, it can be rendered directly in the Git repository (GitHub, GitLab, BitBucket, etc.) and can be used with Sphinx [4] to create consistent and well-structured technical documentation from multiple Markdown files. Sphinx has been established as the tool of preference for the fabrication of technical documentation by many ventures [9], such as one of the most notable of all time, Linux [10].

IV. DOCUMENTATION AS CODE - CASE STUDY

At the begginning of this section, let me first identify all relevant documents (Fig. 5) and map them to SWE process group (Fig. 2):

– SWE.1 - Software Requirement Document (SRD);

– SWE.2 - Software Architecture Document (SAD), Platform Architecture Document (PAD);

– SWE.3 - Unit Design Document (UDD) - *Generated*;

– SWE.4 - Unit Test Specification (UTS), Unit Test Results (UTR) - *Generated*;

– SWE.5 - Integration Test Specification (ITS), Integration Test Results (ITR) - *Generated*;

– SWE.6 - Software Test Specification (STS), Software Test Results (STR) - *Generated*.

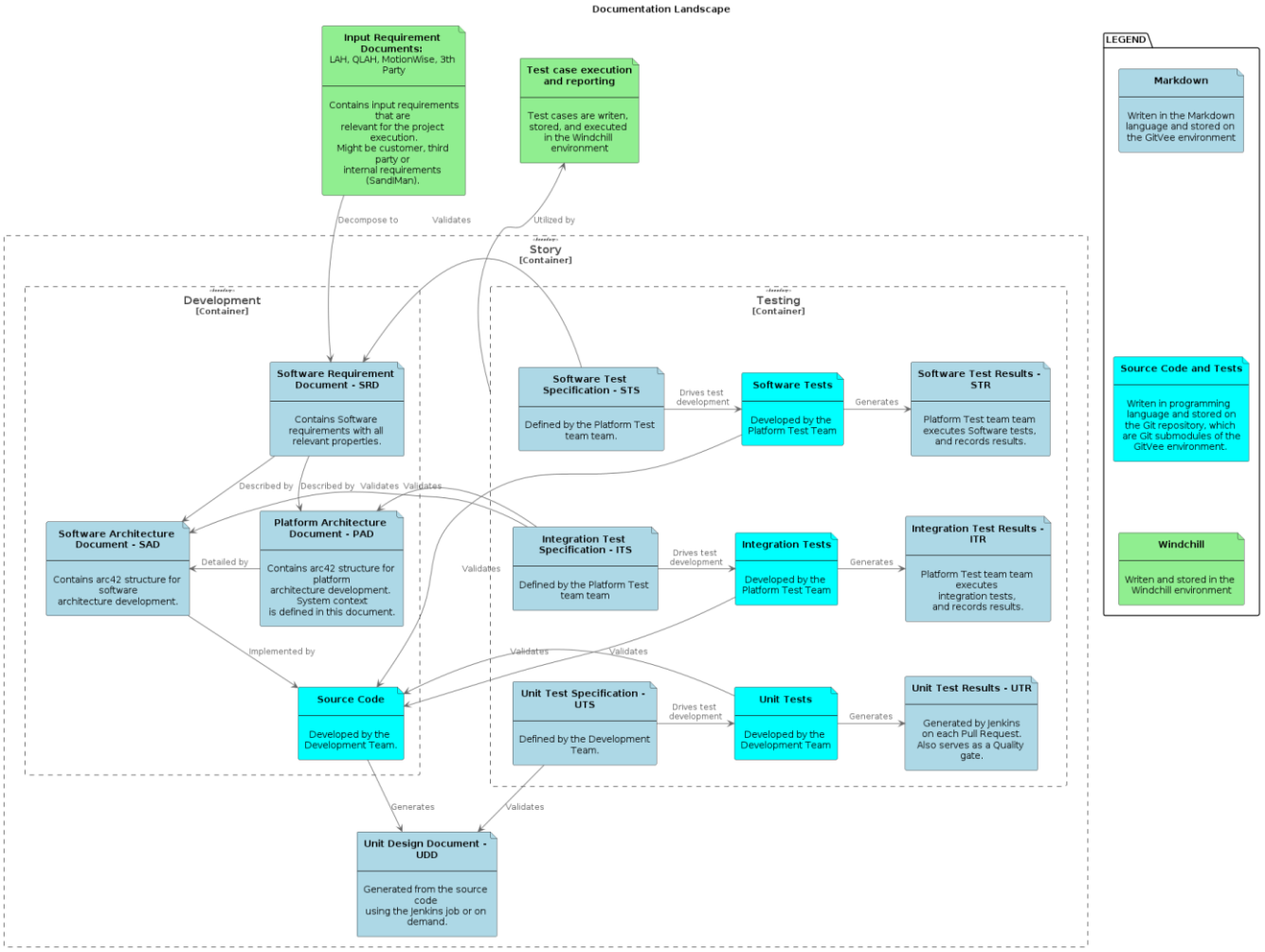

Fig. 5. Documentation landscape compliant with the ASPICE SWE process group.

This research was conducted during a joint effort between a Tier 1 software company and one of the largest German OEMs. During such collaborations, the usual practise is to have a *Lastenheft* and a *Pflichtenheft*. The first one, a *Lastenheft*, is a customer input requirement presented in the form of a SysAD (SYS.3, Fig. 2) or other documents. The second, a *Pflichtenheft*, represents the specification that describes in detail how the Tier 1 software will meet the customer's requirements (*Lastenheft*). The actual implementation only begins after the customer has accepted the *Pflichtenheft*. In this context, *Pflichtenheft* is directly connected with two layers (out of three) of the ASPICE Software engineering group, SWE.1 and SWE.2, consequently with three documents: SRD, PAD, and SAD. As one can notice, these are the only three documents created manually from the entire documentation landscape (Fig. 5). Also, it can be inferred that these three documents (SRD, PAD, and SAD) must always be up-to-date and consistent with the implementation, but also accessible by Tear 1 and OEM to communicate efficiently. For this purpose, it has enabled access to *Pflichtenheft* (SRD, PAD, and SAD) on the documentation server, through the VPN channel, so the customer can access these documents in real time and discuss them with feature teams. This close feedback loop on the documentation level is important since it gives confidence to both Tier 1 and the customer about problem identification and some design choices. It is important to emphasise that a second feedback loop is established once working software is delivered to a production-like environment. Afterward, the next iteration loop can begin, *Pflichtenheft* is adjusted according to new learnings, and the source code is updated accordingly. Without the DaC efficient dynamic of this iteration, the loop would not be feasible, and it would be much harder to maintain pace and consistency between the upfront design, established in SRD, PAD, and SRD, and the implementation. This conclusion has been derived from the comparative analysis between the case study used in this research, where the DaC system approach has been widely adopted, and other projects within the same company, where

the traditional exchange format between Tier 1 and OEM has been performed using ReqIF files (Fig. 2). This is one example where the DaC systematic approach has an immense auspicious influence on the dynamic of the software development process, materialised through the fast feedback loop between the feature team and the customer. This enables incremental and iterative software development processes that usually lead to optimal solutions by any means.

The discipline required to maintain consistency between software specifications, upfront design, and implementation can be difficult to maintain. Tools and processes that facilitate and motivate feature teams during software development to be diligent were the main drivers behind the research described in this paper. First, it is important to make documentation a habit. To do this, documentation should be attractive and easy to create [11]. As feature teams are responsible for creating technical documentation and like to code, providing them with the opportunity to "code" documentation felt like a natural choice. Also, the process of creating documentation can be done in the same integrated development environment (IDE), using the same tools. This reduces context switching (performance killer) and ineffective (extraneous) cognitive load.

There are three types of cognitive load [12]: *intrinsic*, *extraneous*, and *germane cognitive load*. In terms of writing documentation, the intrinsic cognitive load could be knowing the syntax of the language to represent the data. Extraneous cognitive load might be instructions on how to manage documentation files in the third-party document management system. Germane's cognitive load is the only one related to intellectual activities that generate actual value, the documentation content. According to cognitive load theory [13], one should "encourage learner activities that optimise intellectual performance". Thus, DaC system approach has been designed as a function that minimises the intrinsic and extraneous working memory footprint and amplifies the germane cognitive load.

Intrinsic cognitive load has been minimised by selecting a simple Markdown language as a choice for writing documentation. It is something closest to plain text, and therefore it does not require almost any mental effort to express yourself.

The extraneous cognitive load has been reduced by providing feature teams with the opportunity to work on technical documentation without leaving the familiar working environment (IDE), reviewing, storing, and versioning the documentation next to the source code (on the Git repository), and automating documentation verification, build, and deployment.

The germane cognitive load refers to the effort needed to create a lasting storage of information. In DaC context, it is related to creating documentation that fulfils its purpose and brings value to the users: feature teams, customers, etc. Documentation can bring some value only if it is consistent with the source code. Reviewing, storing, and versioning documentation with the source code (and other relevant artifacts) increases the chances for consistency, thus maximising the value produced by engaged germane cognitive load.

Cognitive load can also be directly related to *accidental* and *essential complexity* [14]. Accidental complexity could be processes and third-party tools introduced to "facilitate" documentation management, but instead creates unnecessary extraneous cognitive load; therefore, it should be removed. Essential complexity might be the process of creating consistent usable content through the participation of germane cognitive load.

Besides making a software development-centric documentation creation environment that motivates feature teams to write better documentation more often, there should be also some sort of gating mechanism and protection against undesired behaviour, like introducing broken links, inconsistencies, etc. An important concept that helps detect inconsistencies between implementation and documentation is traceability.

This research highlighted the important concept of traceability, which was explored and established through multiple levels and perspectives. The DaC system was of particular interest to the ASPICE auditors, prompting a careful design of its components. The first perspective of traceability has been established through the use of the ALM tool, which groupes related artifacts into a package called "User Story". A single input requirement can be decomposed into multiple User Stories that can be interlinked and even share some development content, but each Story contains all the related artifacts necessary to deliver the Story in the form of a micro-V model increment. Another aspect of traceability is established through the branching strategy process. Although trunk-based development is promoted as the overall branching strategy, short-lived branches are allowed. The strategy is simple: when one starts to work on a particular subtask (SRD, SAD, etc.), it creates a branch. Since one Story should be completed within a two-week cycle (a Sprint), branches should not have a lifespan longer than that (ideally, no more than a couple of days). It was also instructed to merge at least once a day to avoid merge conflicts and integration problems. This aspect of traceability is important for top-down analysis, as one can easily trace related work in the form of a branch by following User Stories and decomposed micro-V model subtasks. Each subtask should contain the link to the branch and related Pull Request where the review process occured.

Another perspective of traceability has been achieved more in the DaC spirit through the source code (software and documentation). The idea behind this concept was simple: one should leave a piece of evidence in the source code (software and documentation) in the form of a User Story ID (generated by the ALM tool) wherever some work related to that Story occurred: decomposition in the (SRD), architectural design (SAD), writing implementation (source code), tests, etc. This is convenient from a development perspective since one can simply search for the Story ID in the IDE and all related micro-V model artifacts (SRD, SAD, source code, tests, etc.) will appear. If necessary, those can be changed, and afterward, Pull Request should be created where the review process is initiated. This is also convenient for the official ASPICE audit process, since it is straightforward to find all the evidence by searching the Git repository.

From the user's point of view, a top-down traceability

analysis can be performed using the ALM tool or bottom-up by searching for the Story ID in the Git repository. Furthermore, an automated gating system could be integrated into the PR handler to prevent merging the User Story into the main branch if artifacts from the micro-V model are missing, thus ensuring the releasable state of the main branch is maintained at all times. At the time of this research, the automated gating system was still under development, so the review process was the only way to prevent this behaviour. The traceability graph builder was developed as a prerequisite for this automation, so the next step would be to integrate the gating system into the PR handler.

This research was motivated by the fact that software development is a relatively new engineering discipline (especially in the automotive industry) and there are many conflicting views on the proper software development processes. From conventional automotive waterfall processes, which heavily emphasise upfront design, to Agile development techniques that question the need for documentation and prior design. The design of the DaC system, described in this paper, attempts to close this gap by providing some useful recommendations, so feature teams can promote technical excellence through lean software development practises and comply with automotive standards.

A. Requirements as Code - Executable Specifications

Historically, in the automotive industry, requirements elicitation has been a continuous process of clarifying the scope of the work that needs to be done between the customer and Tear 1 software supplier (Fig. 6). In practise, this usually means that the Agile principle "customer collaboration over contract negotiation" is neglected, and the "contract game" occurs by throwing files in the requirement interchange format (ReqIF) files over fence.

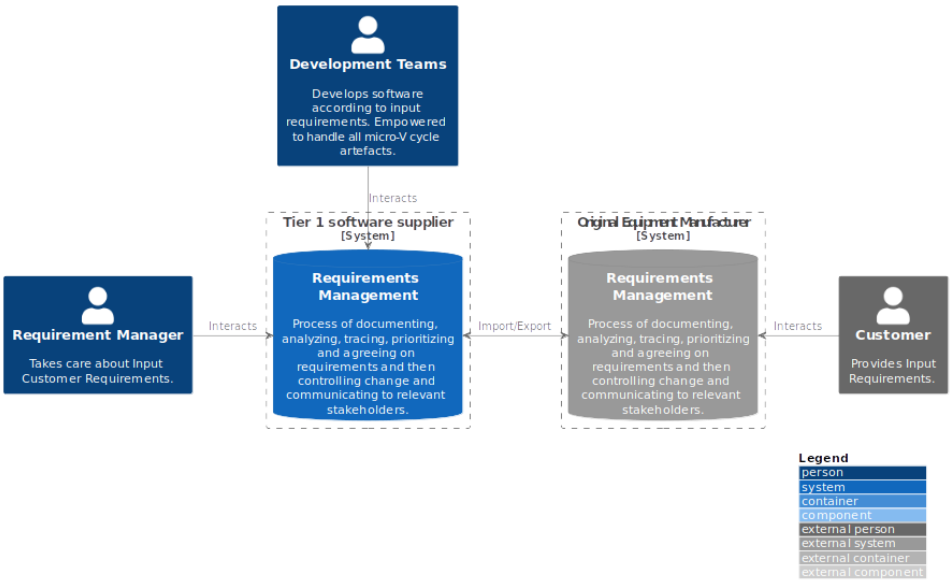

Fig. 6. Requirements elicitation process between Tier 1 and OEM.

The author of this research found this process to be a relic of the past and something that needs to be replaced with direct collaboration between customer and feature teams. Writing good software requirements was never an easy task. This research adopted some practises proposed by the behaviour-driven development (BDD) methodology to explore alternatives to the traditional approach and improve the process of defining the problem that needs to be resolved.

BDD is a software development methodology that emphasises the collaboration between developers, testers, and stakeholders to define and understand the behaviour of a system. It is an extension of test-driven development (TDD) and emphasises the use of natural language and examples to describe the desired behaviour of the system.

BDD uses a specific syntax called "Gherkin" to describe the behaviour of a system in terms of User Stories and related scenarios (executable specifications), which are specific examples of how the system should behave in a certain context. These scenarios are written in a natural language format, making it easier for stakeholders to understand and provide feedback.

The BDD process starts with the stakeholders defining the acceptance criteria/test [15] for the system in the form of scenarios (SWE.1, Fig. 2). These scenarios are then used as a basis for writing automated tests (SWE.6, Fig. 2), which are used to ensure that the system behaves as expected. Developers then implement the system and run automated tests to ensure that the system behaves as described by the scenarios.

BDD is often used in conjunction with Agile development methodologies, such as Scrum, and emphasises the importance of continuous testing and feedback to improve the quality of the system.

Overall, BDD is a methodology that helps to ensure that the system is developed to meet the needs of stakeholders by fostering collaboration between the different roles involved in the development process and providing a clear and common understanding of the behaviour of the system.

In this research, feature teams have used the SRD template [16] to decompose input requirements (SYS.3, Fig. 2) into User Stories and scenarios. SRD is then stored and versioned on the Git repository, in addition to the source code, the software architecture, and other relevant artifacts.

This is important to emphasise because, with this file organisation, it is easy to change User Stories and scenarios from the same IDE, and perform baselining with the same tool (Git) for the whole micro-V model package. This setup enables the incremental and iterative *modus operandi* between feature teams and customers.

B. Architecture as Code

One of the major challenges during system (software) design is managing complexity. This has an immense influence on the maintainability of the system since complexity is what makes software hard to change. Major complexity inceptions are irreversible design decisions and all workarounds that follow. To avoid this and reduce accidental complexity, creating software architecture for such a complex system should be an iterative process [17] in close collaboration with various stakeholders. The most important quality attribute of the software architecture becomes how easily it can be changed.

"*First make the change easy (warning: this might be hard), then make the easy change*" - Kent Beck.

In modern software development practises, creating software architecture is a continuous collaborative process between various stakeholders. Conway's law [18] teaches us that the organisational team structure represents a blueprint when it comes to crafting system (software) architecture and that organisations that recognise this have more chances to succeed [12]. When creating a new system, organisations can apply inverse Conway law manoeuvre, and organise teams in the such constellation to achieve desired system architecture. As one can notice, management of the company becomes a system architect, or at least an influencer, through the creation of teams organisation. This becomes inevitably a large upfront design that is so loathed by the Agile community. Communication between management and feature teams becomes imperative to create an optimal system design; therefore, "there is no silver bullet" solution when it comes to crafting such a design.

There have been many attempts in the past to create graphical, drag and drop, and non-code environments for crafting a system/software architecture. The problem is not to create a such graphical environment that can enable non-technical people to drag and drop software elements, make some connections, and then generate some code out of it. The problem is the maintenance of such a product (system/software architecture): How to establish efficient and sustainable round-trip between these graphical design elements and the source code? When design changes, how do we integrate generated source code with the existing code base? When the code base is changed, how and when to update the graphical representation? Many organisations abandoned the first part (to generate code out of graphical elements), but kept only the second, to regularly update the graphical representation of the code base to ensure consistency. Similarly, as in the rest of the documentation the main enabler to maintain consistency between architectural design and the source code was to make it attractive and easy to change to become a regular habit [11], as well as to make it functional and integral part of the software development cycle.

The role of a software architect has evolved from being the mastermind of system design to being a feature team facilitator and teacher. Now, crafting software architecture is a team activity. To make architectural work more engaging for software developers, the obvious choice is to make it more coding-like. The same conclusion applies when it comes to making the architecture easy to change operationally and functionally. Software engineers like to develop software, so providing them an opportunity to craft architecture in the same manner and using the same working environment increases the chances that the team will treat it equally to source code and keep it consistent. Text is the most powerful abstraction. There were many attempts in the past to create an architectural language like ADL, ArchiMate, ABACUS, etc. In the automotive community, the foundation "Genivi" defined Franca as the interface definition language (IDL) and Franca+ as an extension that enabled a language-based modelling approach for AUTOSAR environments [19]. The main advantage of this approach is that it provides a mechanism to automate source code generation from the model using the CI infrastructure and thus ensuring consistency between the model, source code, and configuration files all the time. A similar approach has been taken in this research, where it has been developed in-house domain-specific language (DSL) based on textX framework to model the AUTOSAR environment at the code level (Level 4, [6]).

In addition to generating the source code from the model, the main purpose of the software architecture is to tell the story of the software [6]. This is important because it exposes the internal structure (static architecture) and behaviour (dynamic architecture) of the system and prevents inceptions of accidental complexity to crawl into the design and make software architecture more difficult to change. In this research, the C4 model [6] has been used to present architecture on four levels of abstraction: System Context, Containers, Components, and Code. As it has been mentioned, the Code level has been modelled using custom DSL to generate AUTOSAR arxml model and source code. Three other levels have been modelled using the PlantUML [20] language and the extension of the C4 model [21] (the same language that was used in this paper to create figures). During this research, PlantUML files were stored and versioned in the Git repository along with the source code and other artifacts. The output of the PlantUML are svg images that are referenced in the software architecture document (SAD). These images are updated by the CI process (Fig. 8) on every Pull Request. Also, one interesting feature of the PlantUML language is that its support includes preprocessing directives, enabling the reusability of PlantUML elements and the creation of composite diagrams. Quality Gate CI pipeline (see the figure at the end of this section) ensures that broken diagrams cannot merge into the main branch.

In this research, the arc42 [22] SAD template [23] has been used to document software architecture. The fifth part of the SAD, known as the "Building Block View", is where the C4 model should be described in detail.

C. Unit Detail Design as Code

Test-driven development (TDD) [24] is an effective software development process that serves primarily as a

design technique. It helps to create code that is reliable and easy to change (maintain). The TDD process involves writing tests before coding, so that one can be sure that the code works as expected. The usual TDD cycle includes the following:

1. Writing test that fails - RED;
2. Writing implementation that makes the test pass - GREEN;
3. Removing duplications and increasing quality - REFACTOR.

Writing the test first has an immense impact on the quality of the source code design. Implementation created this way is written with testability in mind. TDD represents the most powerful mechanism to manage the main software quality properties, such as modularity, cohesion, separation of concerns, abstraction, and coupling management [17]. This mechanism is established through an instant feedback loop in the form of writing tests. If it becomes too hard to write the test for a certain piece of functionality, due to many different reasons, like the setup is too complex, etc., it might be a good point in time to revisit the design. This instant feedback helps to manage the complexity of the system being built. Also, it gives confidence to the feature team to perform source code refactoring more often.

The tests for our software should be understandable, maintainable, repeatable, atomic, necessary, granular, and fast. They should be focused on the behaviour of the system rather than a specific implementation and should be easy to change while remaining true to the system. They should be deterministic and provide the same result every time they run. Tests should be isolated and focus on a single outcome and must be necessary to guide our development choices. They should be small, simple, and focused and provide a clear pass/fail result without needing interpretation. Lastly, they should serve as a tool to guide our development.

When the feature team utilises TDD as a routine and writes tests according to the guidelines written above, then those tests become Unit Detail Design (SWE.3, Fig. 2) and validation (SWE.4, Fig. 2). In this research, not all teams have followed TDD practises, but those who did, used tests produced this way for SWE.3 and SWE.4. The tests have been stored and versioned in the Git repository along with the validated source code.

D. Testing, Validation, and Verification

Testing, validation, and verification are usually connected with the right side of the V-model (Fig. 2). In this research, micro-V iterative loops have been executed throughout regular development cycles daily, including the right side (Fig. 7). During this research, two different testing, validation, and verification contexts were performed, both automated as part of the CI loop: testing, validation, and verification of the software that is developed (Fig. 7) and of technical documentation that is being produced along the way (Fig. 8).

During this research, testing of the software has been performed on three levels (Fig. 2):

1. SWE.6 Software Qualification tests - Acceptance tests (executable specifications) are developed as part of the BDD process of defining acceptance scenarios using Gherkin syntax (for each User Story), before any development activity. These tests (executable specifications) validate the expected behaviour of the software at the highest level of abstraction. There is no need for additional tests on this level. The direct advantage of Requirements as Code approach.
2. SWE.5 Integration tests - Generated from the DSL architectural model, using the integration test framework developed for that purpose. This has been enabled by treating Architecture as Code. These tests verify that the specification of the architecture model is met (interconnections between software components), therefore ensuring consistency between software architecture and implementation.
3. SWE.4 Unit Validation tests - Developed through practising TDD, before implementation, following the red, green, and refactor cycle. These tests validate the behaviour of software units at the lowest level of abstraction. There is no need for additional test development in addition to this, which is a direct consequence of Unit Detail Design as Code approach and following TDD methodology.

This CI/CD pipeline (Fig. 8) is triggered on every PR update. Upon the completion of each execution, test results at all three levels are documented in the Jenkins job and uploaded to the Git repository, thus creating a historical record and ensuring transparency at all times. PRs that do not pass all stages in the CI/CD pipeline (Fig. 8), are marked as unapproved by the system builder and cannot be merged to the main branch before being fixed, thus establishing a direct feedback loop towards the PR author.

Establishing and maintaining consistent technical documentation is a difficult task, which requires the implementation of systematic remedies to ensure its successful implementation. In the case of the DaC approach, the validation and verification of documentation are conferred to CI Fig. 8 and CD Fig. 9 pipelines.

To ensure consistency of PlantUML files and software architecture, on each PR all diagrams are regenerated, and updated svg files are automatically pushed to the Git repository. Consequently, all references to architectural diagrams (svgs) in the documentation are updated, thus keeping SAD up-to-date. Tedious and error-prone manual processes of generating svgs have been delegated to the CI pipeline, thus offloading feature teams of such activity and making more room for participation of germane cognitive load. To optimise the execution time of the CI pipeline (Fig. 8), only modified Markdown files from the PR that triggered pipeline execution are verified. If all stages pass, PR is approved by the *system builder*; otherwise, it is marked as unapproved, and cannot be merged into the main branch, until the pipeline is green.

The CD pipeline (Fig. 9) is triggered by merging to the main branch. It performs additional checks on the entire documentation landscape, not just files modified by the PR. If this stage passes, documentation is built with the Sphinx [4] and deployed to a dedicated documentation server. It might happen that some changes have been merged to the main branch before the issue has been discovered on the documentation deployment pipeline. That will prevent deployment of the latest changes but still keep documentation in a consistent state, slightly outdated but

consistent. The PR creators will be automatically notified to fix the issue. Taking into consideration the dynamic of merging changes multiple times per day (and fixing such issues), it has been decided that this trade-off is acceptable. It is much more important to establish fast feedback loop on the CI pipeline, rather than to be bulletproof. Issues that are missed by the CI are caught by the CD pipeline. The study has shown that these issues rarely occur and establishing a fast feedback loop on the PR is of utmost necessity.

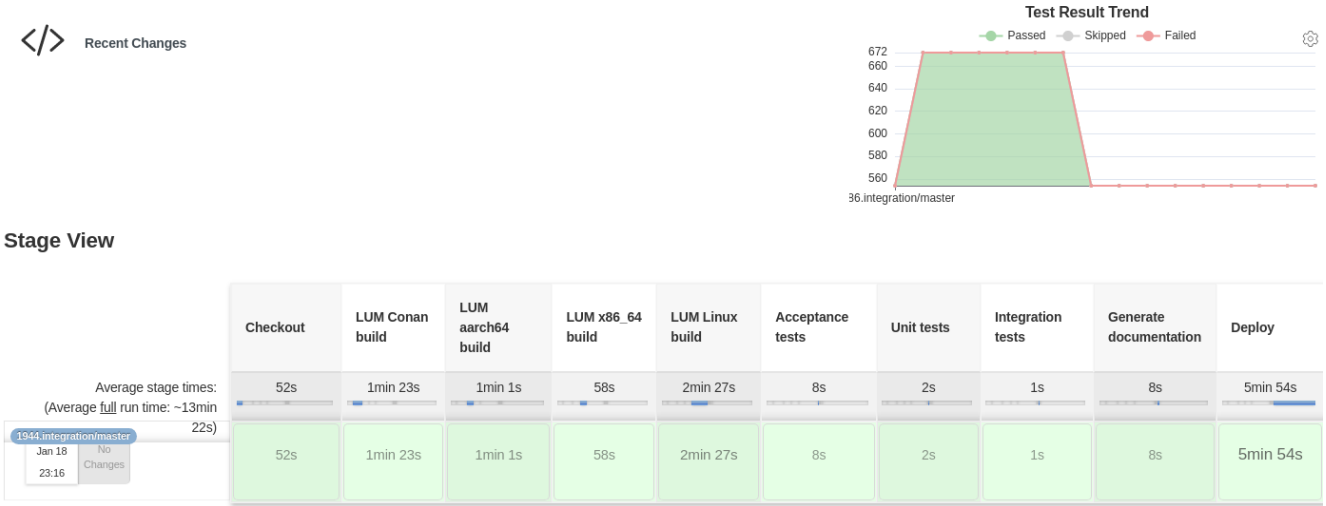

Fig. 7. Continuous delivery pipeline for software.

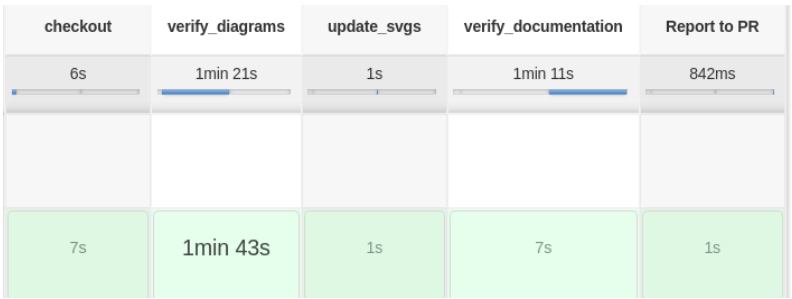

Fig. 8. Documentation as Code Continuous Integration pipeline.

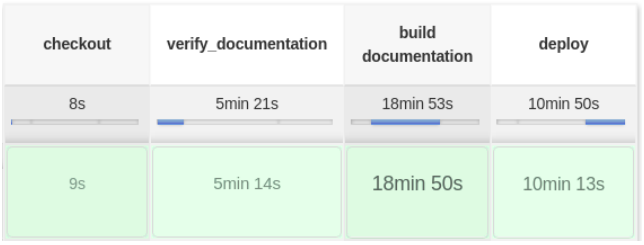

Fig. 9. Documentation as Code Continuous Deployment pipeline.

V. RESULTS AND DISCUSSION

Stability and Throughput are the only two measures that could be used to evaluate changes applied to processes, tools, technology, etc. [17], [25]. When we change a process (or whatever), we can measure the impact of this change on either of these two measures and steer changes accordingly.

Stability was tracked during the research as one of the key metrics, measured by the number of defects reported. The results revealed interesting findings. The blue bars depicted in Fig. 10 represent the defects reported during the research in 2022 when the DaC approach was applied, while the red bars represent the number of defects reported in the legacy project in 2020. To make the comparison more meaningful, data were collected during the same phase of the projects and the customer was not changed. The same feature teams were mostly involved in both projects, with the only difference being the software development methodology. In the project where the DaC approach was applied, 35 % fewer defects were reported on average than in the project where legacy processes and tools were utilised. This number is quite similar and comparable to the results of different studies [26], [27], where the impact of the test-first approach (TDD) on defect reduction ranged from 40 %. In another study [25], it was measured that feature teams that employed techniques like those presented in this paper (BDD, TDD, Continuous Delivery, etc.) spent 44 % more time performing useful tasks.

In addition to the reported defects, Fig. 10 shows additional useful data on the Throughput and the effects of continuous and disruptive delivery processes on the reported defects. It is important to note that this was one of the main philosophical/process-based differences between the two projects observed in the case study. Throughput in the DaC Project was managed through continuous delivery, whilst in the Legacy Project it was disruptive. The three red peaks in Fig. 10 indicate the number of reported defects just after disruptive delivery occurred. In the case of the DaC Project, Fig. 10 shows a linear progression in the number of reported defects. This is expected due to the continuous delivery process and the fact that the number of delivered lines of code (LOC) increased over time, but the ratio [defect]/[LOC] remained constant.

The interesting statistic can be derived from the documentation repository (Fig. 11). The statistic provides

data on the number of commits per month related to the DaC approach during the research. In the first couple of months, the whole infrastructure was created, and feature teams were onboarded. Afterward, there was a steady influx of commits per month related to the creation of technical documentation (executable specifications, architecture, unit design, etc.). When comparing this statistic to the almost non-existent documentation from the Legacy Project, a direct correlation can be made between treating documentation as a first-class citizen (DaC) (Fig. 11) and a reduced number of reported defects (Fig. 10).

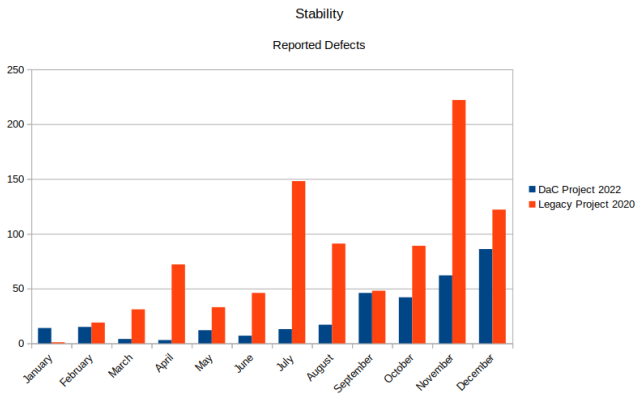

Fig. 10. Stability measured throughout the first year of the Project.

Fig. 11. Statistics of the documentation repository (commits per month).

The cumulative flow diagram in Fig. 12 shows a snapshot of Throughput during the research. It indicates that 1629 subtasks related to the micro-V model (Fig. 4) (Story[container]) were completed within three months. In particular, 184 User Stories were delivered across five different features over the same period, averaging six Stories per Sprint. This high pace was made possible by tailoring the User Stories to include both implementation and documentation.

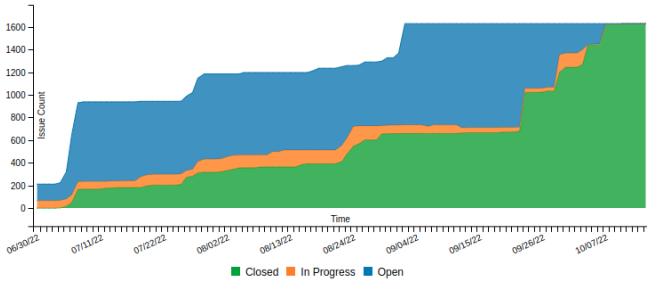

Fig. 12. Release cumulative flow chart.

The DaC approach has enabled a high-paced Throughput, by providing feature teams an opportunity to work on all micro-V model artifacts in a single working environment for software and documentation development. In DaC approach, documentation is treated equally as important to source code and delivered together, whereas in other (legacy) projects it was usually done at the end of the project lifecycle. This has a significant negative impact on the quality of the delivered software (Fig. 10), since if documentation is treated separately from implementation, it usually means that design decisions were taken ad hoc and not communicated properly to other stakeholders. This can lead to a suboptimal system architecture that is difficult to change, negatively impacting the maintainability of the system and other quality attributes.

When comparing state-of-the-art automotive software development practises and our approach that introduces DaC, in terms of quality and efficiency of a delivered product, there are several points to consider.

Process improvement: ASPICE provides a process framework, a set of recommended practises and guidelines for software development, testing, maintenance, etc. to improve the efficiency of the software development process. It emphasises that processes should fulfil their purpose, make sense, and bring value to the user. The DaC approach brings process improvement by removing waste embodied in processes and third-party tool overhead, significantly reducing context switching, and improving performance. The working environment and processes have been designed to be software development-centric, adjusted to the only stakeholders in the entire system that generate actual value for the customer. This has an auspicious impact on quality since feature teams treat documentation as code and keep it consistent with implementation. Up-front design (Architecture as Code) and testability of the system (executable specifications, and UDD as code) became highly integrated and important software development properties in the DaC approach.

Collaboration improvement: The DaC focusses on improving collaboration and accessibility of the documentation to all relevant stakeholders, as well as facilitating inter- and intra-team communication. By treating DaC, developers can work on the documentation in parallel with the codebase, which increases the efficiency of the development process. All important design decisions are communicated through the regular Pull Requests review process (intra-team), leaving a historical record as evidence of evolutionary design. When it comes to cross-cutting decisions affecting multiple domains (feature teams), the DaC approach resolves this systematically by utilising the Git Codeownership mechanism. Improved collaboration and communication prevent accidental complexity from creeping into the design, making architecture more flexible.

Traceability: The DaC approach allows for better traceability of the documentation, as it is stored in version control systems and can be easily linked to the codebase. All micro-V model artifacts are traceable from different perspectives. Most importantly, software developers can search for a Story ID and find all relevant micro-V model artifacts in the working environment, making it convenient for updates and reviews, thus increasing the chances of consistency between implementation and technical documentation.

Automation: The DaC approach makes it easier to automate the documentation process, such as building and

continuously deploying the documentation, which is a prerequisite for continuous delivery. Multiple CI/CD pipelines ensure direct feedback loops between feature teams and quality gateways, thus providing a safe environment for experimentation and learning, which is essential for software engineering and finding optimal solutions. Thousands of tests are automatically executed on every pull request update to prevent undesired behaviour of the system.

Maintainability: By treating DaC, it is much easier to maintain the documentation and the source code, since they are both stored in the same version control system side-by-side. This quality attribute is tightly coupled with the ability to change, which is one of the hallmarks of good architecture. Additionally, using a single version control system simplifies the release process since the entire package (functionality and documentation) can be bundled, tagged, and released together. This also simplifies reproducibility: one can simply check out the released package and all the relevant artifacts are present, including source code, architecture, executable specifications (acceptance tests), test results, etc.

Testability: The DaC approach is all about managing complexity and creating flexible architectures that are easy to change. In complex system environments, it is impossible to make exact predictions about the impact of even trivial changes on the overall behaviour of the system. Therefore, it is necessary to have a different set of tests that can either confirm or discard our predictions about the behaviour of the system after a new feature is added or a single line of code is changed. There is no agility without testability. The DaC integrates behaviour-driven development (BDD) and test-driven development (TDD) methodologies, where software is designed through writing tests first and implementation second, ensuring the system's testability at all times throughout the process at both high (BDD) and low (TDD) levels. Mid-level testability is covered with generated integration tests from the architecture model developed using the Architecture as Code tool set.

Reusability: In the DaC approach, reusability is not considered a must-have under any circumstances. This property is closely related to the Don't Repeat Yourself (DRY) and Single Responsibility principles. Software components are reused only when it is obvious that the reused elements will run in the same problem domain. However, in complex system environments, what is initially obvious can turn out to be untrue. This analysis begins with the BDD and continues through Architecture as Code until TDD. All three levels of support include (reuse element) preprocessing directives, so operational support is given by design. However, it is more important to decide when to reuse for optimal system design.

Accessibility: This is an important aspect of documentation, which DaC approach resolves twofold. First, documentation is embedded directly in the repository close to the source code and other development artifacts, which means that it is accessible within the integrated development environment. This eliminates the need to leave the working environment to access the most up-to-date documentation. Second, documentation is continuously deployed to the server, ensuring it is kept up-to-date and accessible to everyone with the link and necessary project access rights.

Transparency is one of the three pillars of empiricism, alongside adaptation and inspection, which are ubiquitous in the DaC environment. The main infrastructures that enable transparency in DaC approach are Pull Requests and CI/CD pipelines. However, it is the content that is continuously filled in following the DaC methodology that makes the difference. Thousands of tests are executed on all levels for each PR update, and results are published on the CI server as well as in the repository, making test reports transparent from several perspectives. Transparency is also omnipresent in the ALM project structure (Fig. 4), which is important for the MAN process group, Fig. 2. Cumulative flow diagram (Fig. 12) and various other metrics are generated from the ALM structure (Fig. 4), providing insight into the statuses of different user stories and release health checks.

This research was inspired by the idea of continuous and never-ending improvement (Kaizen [28]) of processes and tools to produce better quality software faster [17]. In DaC methodology, quality is built ground up, brick by brick (micro-V cycle by micro-V cycle), through incremental and iterative cycles. This idea was based on the philosophy of W. Edwards Deming, the father of quality, which suggested that organisations that prioritize improving quality will see a decrease in costs, whereas those that prioritize cost-cutting will inherently reduce quality and end up incurring higher costs [29].

"*Inspection to improve quality is too late, ineffective, costly. Quality comes not from inspection, but from the improvement of the production process.*" - *W. Edwards Deming*, Out of the Crisis [30].

VI. CONCLUSIONS

It has been demonstrated that the DaC approach enhances the Stability (quality) and Throughput (efficiency) of the software development project. DaC improves documentation collaboration and accessibility, making it easier to create and maintain. Furthermore, DaC promotes the testability of the system as imperative, employing behaviour-driven development (BDD) and test-driven development (TDD) methodologies.

This research has demonstrated that the DaC approach is feasible even in an area such as automotive, which is highly dependent on consistent documentation. It has elucidated the advantageous effects of the DaC approach and how to ensure consistent and up-to-date technical documentation throughout the project lifecycle management. The major conclusion from this research is that when the task of writing documentation is made attractive and easy, feature teams will regularly update it and keep it consistent. The DaC approach aims to achieve this by adjusting processes and tools to be software development-centric.

Processes and tools should be designed and selected to facilitate creativity and enjoyment during documentation crafting, just as when writing code, ideally in the same working environment. This research offered many incentives for this conclusion on different levels and perspectives.

When writing Requirements as Code (executable specifications) using BDD methodology, such requirements become tightly coupled to the behaviour of the system (not

the implementation details). One side effect is full requirement coverage with acceptance tests.

Architecture as Code provides multiple opportunities for a feature team to express their creativity when designing architecture through the activity they enjoy the most - writing code. The software architecture created in this way can be used as a model from which source code and integration tests can be generated. One side effect is complete interconnection coverage with generated integration tests.

Applying a test-first approach (TDD methodology), feature teams have the opportunity to design software units from the perspective of the user, thus establishing a direct feedback loop between the design and the customer. One side effect is full source code coverage with unit design tests.

This research has provided practical guidelines for the DaC approach. It has been demonstrated that treating all relevant documentation artifacts as source code using the same tools and working environment can have beneficial effects on consistency and software project management. However, it is important to emphasise that the DaC approach presented in this paper does not represent a final solution set in stone, but rather a solid practical process and tool framework for the execution of software projects that embrace and facilitate the philosophy of continual, incremental, and iterative improvements, with feature teams at its focal point as organisational stem cells.

In the DaC approach, the testability of the system is considered one of the most important quality properties. Fast feedback loops embodied in CI/CD are seen as the most effective mechanisms for creating a consistent system that features teams can confidently reshape and refactor, as well as incrementally add new features and iteratively refine toward optimal solutions. This is essential to manage complexity and control variables during development or forensic analysis. Being always close to a safe spot when experimenting and learning is liberating. This is exactly what test coverage, CI/CD feedback loops, and version control systems provide when implemented properly. With every git commit deployed, authors get a genuine sense of continual and incremental improvement of the system. This is such a powerful psychological mechanism that encourages people to commit small and frequent. The author of the research is firmly convinced, based on empirical evidence, in the described approach and has even crafted this paper [31] using the same principles.

ACKNOWLEDGMENT

I express my deepest gratitude to my colleague (and brother-in-law), Dr. Svetozar Miucin, for his invaluable guidance and support throughout this research project. I am also grateful to my colleagues, Dr. Branislav Kordic and Dimitrije Stojanovic, for their helpful comments and suggestions. Furthermore, I am sincerely thankful to SVP Dr. Nemanja Lukic for his selfless support, without which this research would not have been possible. Lastly, I cannot express enough appreciation to overall PM Dr. Nenad Cetic for his invaluable contribution to my professional career (and this research).

CONFLICTS OF INTEREST

The author declares that he has no conflicts of interest.

REFERENCES

[1] K. Hoermann, M. Mueller, L. Dittmann, and J. Zimmer, *Automotive SPICE in Practice: Surviving Interpretation and Assessment*. Rocky Nook, 2008. [Online]. Available: https://books.google.de/books?id=7wcoAQAAMAAJ

[2] *ASPICE Guide*. [Online]. Available: https://knuevenermackert.com/wp-content/uploads/2021/03/ASPICE-Guide-KM2021-01.pdf

[3] "MyST - markedly structured text - parser". [Online]. Available: https://myst-parser.readthedocs.io/en/latest/

[4] "Welcome" - Sphinx Documentation, Sphinx Python Documentation Generator. [Online]. Available: https://www.sphinx-doc.org/en/master/

[5] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[6] "The C4 model for visualising software architecture: Context, Containers, Components, and Code". [Online]. Available: https://c4model.com/

[7] "Trunk Based Development: Introduction". [Online]. Available: https://trunkbaseddevelopment.com/

[8] *Markdown Guide*. [Online]. Available: https://www.markdownguide.org/

[9] Sphinx examples. [Online]. Available: https://www.sphinx-doc.org/en/master/examples.html

[10] Linux technical documentation. [Online]. Available: https://www.kernel.org/doc/html/v4.10/index.html

[11] J. Clear, *Atomic Habits: Tiny Changes, Remarkable Results: An Easy & Proven Way to Build Good Habits & Break Bad Ones*, 1st ed. Avery, an imprint of Penguin Random House Business, New York, 2018.

[12] M. Skelton, M. Pais, and R. Malan, *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution Press, 2019. [Online]. Available: https://books.google.de/books?id=oFdRuAEACAAJ

[13] J. Sweller, J. J. G. van Merrienboer, and F. G. W C. Paas, "Cognitive architecture and instructional design", *Educational Psychology Review*, vol. 10, pp. 251–296, 1998. DOI: 10.1023/A:1022193728205.

[14] Brooks, "No silver bullet: Essence and accidents of software engineering", *Computer*, vol. 20, no. 4, pp. 10–19, 1987. DOI: 10.1109/MC.1987.1663532.

[15] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*. Pearson Education, 2009. Available: https://books.google.de/books?id=QJA3dM8Uix0C

[16] SRD template. [Online]. Available: https://gitlab.com/labsoftdev/docs-as-code/-/blob/main/templates/micro-V/Requirements/SRD-Requirement_Template.md

[17] D. Farley, *Modern Software Engineering: Doing what Works to Build Better Software Faster*. Addison Wesley, 2021. [Online]. Available: https://books.google.de/books?id=ZKxHzgEACAAJ

[18] M. E. Conway, "How Do Committees Invent?", *Datamation*, pp. 28–31, 1968. [Online]. Available: http://www.melconway.com/Home/pdf/committees.pdf

[19] S. Schlichthaerle, K. Becker, and S. Sperber, "A domain-specific language based architecture modeling approach for safety critical automotive software systems", in *Proc. of CEUR Workshop*, 2020, pp. 1–6.

[20] PlantUML. [online]. Available: https://plantuml.com/

[21] PlantUML C4. [Online]. Available: https://github.com/plantuml-stdlib/C4-PlantUML

[22] "arc42". [Online]. Available: https://arc42.org/

[23] SAD template. [Online]. Available: https://gitlab.com/labsoftdev/docs-as-code/-/blob/main/templates/micro-V/Architecture/SAD-Software_Architecture_Document.md

[24] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003. [Online]. Available: https://books.google.de/books?id=CUlsAQAAQBAJ

[25] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science Behind DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution, 2018. [Online] Available: https://books.google.de/books?id=85XHAQAACAAJ

[26] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice", in *Proc. of 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 34–45. DOI: 10.1109/ISSRE.2003.1251029.

[27] R. Jeffries and G. Melnik, "Introduction: TDD–the art of fearless programming", *IEEE Software*, vol. 24, no. 3, pp. 24–30, 2007. DOI: 10.1109/MS.2007.75.

[28] M. Imai, *Kaizen: The Key to Japan's Competitive Success*. McGraw-Hill Education, 1986. [Online]. Available: https://books.google.de/books?id=q0rCTQlvNMoC

[29] W. E. Deming, *A System of Profound Knowledge*. British Deming Association, 1992. [Online]. Available: https://books.google.de/books?id=v-RSMwEACAAJ

[30] W. E. Deming, *Out of the Crisis*, reissue. MIT Press, 2018. [Online]. Available: https://books.google.de/books?id=PTNwDwAAQBAJ

[31] M. Krunic, "Documentation as code in automotive system/software engineering", 2023. [Online]. Available: https://gitlab.com/momcilo_krunic/elektronika_ir_elektrotechnika_2023/"https://gitlab.com/momcilo_krunic/elektronika.