2024 IEEE 3rd Conference on Information Technology and Data Science (CITDS) | 979-8-3503-8788-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CITDS62610.2024.10791390

Availability modeling in Software Architecture

Tamas Bartok
i.) EPAM Systems
ii.) Institute of Informatics
University of Szeged
Szeged, Hungary
0009-0005-3384-1919

Ferenc Hejja
i.) Doctoral School of Informatics
University of Debrecen
ii.) EPAM Systems
Debrecen, Hungary
0009-0008-5770-5577

Gergely Kocsis
Department of Informatics Systems
and Networks, Faculty of Informatics
University of Debrecen
Debrecen, Hungary
0000-0003-0018-4201

***Abstract*—Most engineers and Solution Architects are familiar in general with Availability as a Quality Attribute. Availability on its own can mean different system quality to different stakeholders and could have a different meaning in different stages of a product. In this paper we are going through different definitions of availability, how it links to the basic concepts in probability theory and will illustrate with examples how they can be calculated. Finally we will explain which steps should be followed to model availability with fault tree analysis.**

***Index Terms*—Solution Architecture, Non-Functional Requirements, Quality Attributes, Availability, fault tolerance, Key Performance Indicator (KPI)**

I. Introduction

A. Software Architecture

In the realm of software engineering, the design and implementation of the software architecture are pivotal to achieving high-quality systems.

As described by [1]:

Definition I.1 (Software architecture)**.** The software architecture of a system is the set of structures needed to reason about the system. These structures comprise software elements, relations among them, and properties of both.

It is also important to note that the definition does not emphasize all such decisions to be made very early and reserves the freedom for an architect to judge which decisions are significant enough to be made early and which decisions can be deferred to a later stage in the evolution of the software architecture.

Every software system has a software architecture which should be materialized in architecture documentation so that members joining later will be working along the same standards, as well as understanding the justification for architectural decisions made in the past. A proper architecture documentation allows the architects to reassess situations and alter certain design decisions made in the past in order to meet the new requirements of the system.

B. Quality Attributes

While it is evident that a software architecture needs to support the needs of the stakeholders as well as the product's functional requirements, on the other hand it may not be evident to all, that the quality of an architecture is defined by how well it satisfies the product's quality attribute requirements.

According to [1] a *quality attribute (QA)* is defined as below:

Definition I.2 (Quality attribute)**.** A quality attribute is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders beyond the basic function of the system.

C. Non-Functional Requirements vs. Quality Attributes

The relation between Non-Functional Requirements (NFRs) [2] and Quality Attributes (QAs) are often defined interchangeably in the literature, while not clarifying Constraints either. For the purpose of consistency and clarity, we define the relation as follows:

$$NFRs = QAs \cup Constraints, \tag{1}$$

where *Constraints* are such requirements where there is zero degree of freedom, this has also been clarified in [3].

As such it is easy to see, while we must abide to the system's and product's constraints, the most important area of decision making for a Software or Solution Architect is around Quality Attributes.

II. Availability

A. Basic terms

Availability as a Quality Attribute refers to a property of software —- namely, that it is there and ready to carry out its task when you need it to be.

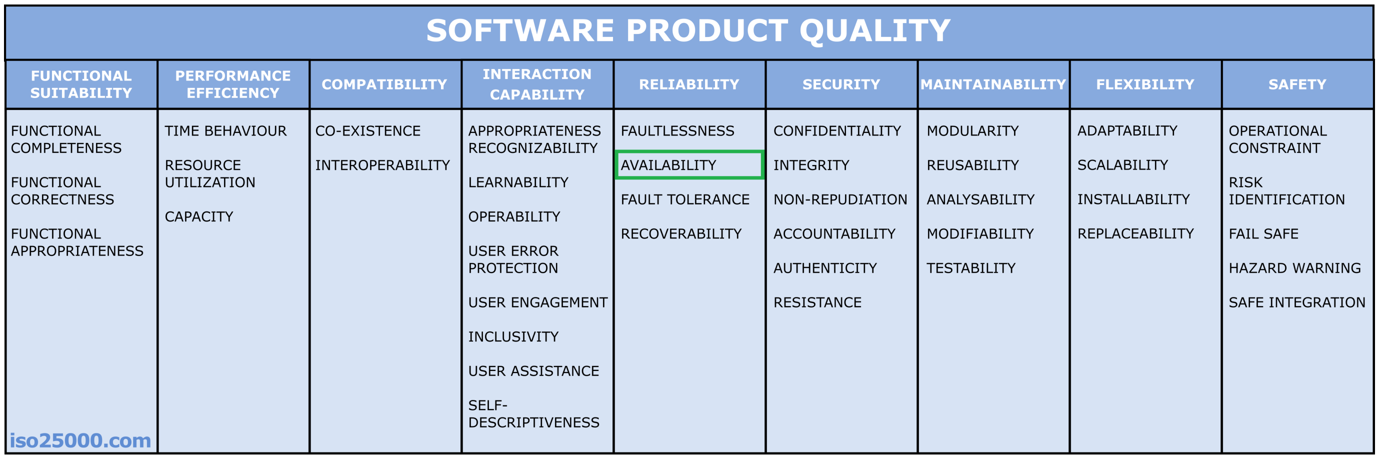
ISO/IEC 25010 [4] defines a product quality model as shown on Fig. 1., which is applicable to ICT (information and communication technology) products and software products.

It defines Availability as follows:

Definition II.1 (Availability)**.** Degree to which a system, product or component is operational and accessible when required for use.

Following the usual definition in IT Software Architecture in [1] availability is defined as follows:

$$PS = \frac{MTBF}{MTBF + MTTR} \tag{2}$$



SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	PERFORMANCE EFFICIENCY	COMPATIBILITY	INTERACTION CAPABILITY	RELIABILITY	SECURITY	MAINTAINABILITY	FLEXIBILITY	SAFETY
FUNCTIONAL COMPLETENESS	TIME BEHAVIOUR	CO-EXISTENCE	APPROPRIATENESS RECOGNIZABILITY	FAULTLESSNESS	CONFIDENTIALITY	MODULARITY	ADAPTABILITY	OPERATIONAL CONSTRAINT
FUNCTIONAL CORRECTNESS	RESOURCE UTILIZATION	INTEROPERABILITY	LEARNABILITY	AVAILABILITY	INTEGRITY	REUSABILITY	SCALABILITY	RISK IDENTIFICATION
FUNCTIONAL APPROPRIATENESS	CAPACITY		OPERABILITY	FAULT TOLERANCE	NON-REPUDIATION	ANALYSABILITY	INSTALLABILITY	FAIL SAFE
			USER ERROR PROTECTION	RECOVERABILITY	ACCOUNTABILITY	MODIFIABILITY	REPLACEABILITY	HAZARD WARNING
			USER ENGAGEMENT		AUTHENTICITY	TESTABILITY		SAFE INTEGRATION
			INCLUSIVITY		RESISTANCE			
			USER ASSISTANCE					
			SELF-DESCRIPTIVENESS					
iso25000.com								

Fig. 1. The product quality model according to ISO/IEC 25010 [4]. Availability is a quality characteristic related to reliability.

where

PS is the probability of the system being available,
$MTBF$ is the mean time between failures and
$MTTR$ is the mean time to repair.

In the constraints of eq. (2) under *failure* we mean:

Definition II.2 (Failure)**.** The deviation of the system from its specification, where that deviation is externally visible, a failure is caused by a fault. Faults typically can be addressed in a number of ways.

B. *Application of availability in cloud systems*

As a representative example having a well modeled system from availability perspective has become more important than ever since the emergence of cloud computing. Cloud Service Providers (CSPs) provide Service Level Agreement (SLA) not to the architecture as a whole, but to individual cloud-hosted architectural components, through the introduction of the term shared responsibility model.

Since the operational cost of a system in the cloud is proportional to the number and type of architectural components, companies have an intrinsic interest in matching their availability needs with the architecture design so that the ideal status is reached where the needs are satisfied, while controlling the operational expenses at the same time.

Today, it is increasingly typical to distribute software system responsibilities among various vendors for infrastructure, design, implementation, operation and several more for SaaS (Software as a Service) products. While this setup can be more efficient, at the same time these vendors need to define their contractual terms for their responsibility area, that often includes availability defined as a contractual KPI.

C. *Availability modeling Definitions and Notations*

In [5] it has been suggested to use conditional probability method to model reliability and in [6] Markov cut-set approach has been used to resolve redundancy in a transmission system.

We apply and combine the above ideas to Software Architecture and apply the conditional probability method to model Availability in Software Architecture.

Let us define:

$P(C1)$ = probability component $C1$ is available
$P(C1|C2)$ = probability component $C1$ is available, if $C2$ is available

Since $C1, ..., Cn$ are s-independent, $P(C1|C2) = P(C1)$. Under no severe external force there is a sufficiently small time window where only at most one component is unavailable

During the modeling steps, we need to first reduce the components to a set of mutually independent components. This can also be achieved by combining dependent components into one for the purpose of modeling.

Note that understandably in modeling we do not want to model severe external forces, i.e. when an external force makes more of the system's components unavailable at the same time.

D. *Availability in relation to Reliability*

As they are often mismatched or used in place of each other, it may be important to emphasize the relation of availability and reliability.

Availability is considered to be a subset of Reliability (see also on Fig. 1). While availability intentionally excludes issues such as implementation and coding errors, reliability defines the quality of a software system from a user's viewpoint. i.e. it defines the system to be reliable if the system is able to satisfy the functional needs of its users. It is easy to see that the system cannot satisfy the above, if it is not available.

On the other hand since the functional needs of the users are so complex to be modeled that in the current state of the industry, reliability is only verified through testing after the system is operational; furthermore cannot be used as a quantified contractual KPI. Availability in the meantime is quite usual to be included in contracts and is often referred in the SLA (Service Level Agreement) definition.

E. *Availability modeling Examples*

1) *Example 1:*

Suppose we have 3, mutually s-independent components, that are present in a sequential call sequence. As such all 3 component must be available at the same time in order to satisfy the business functionality. The call sequence is represented on Fig. 2.

In this case, the system's overall availability, PS can be defined as:

$$\begin{aligned} PS\{C1, C2, C3\} &= P(C1 \wedge C2 \wedge C3) \\ &= P(C1) \times P(C2|C1) \times P(C3|C1 \wedge C2) \\ &= P(C1) \times P(C2) \times P(C3) \end{aligned} \tag{3}$$

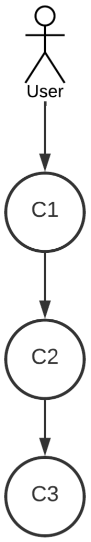


Fig. 2. Figure presenting the simple case of Example 1

2) *Example 2, redundancy used as a tactic:*

In this case we introduce redundancy as an architectural tactic to address a potential case of insufficient result in the previous example, by adding $C2$ as a redundant component to $C1$. Being redundant components it is sufficient to have at least one of $C1$ and $C2$ available at a time besides both of $C3$ and $C4$ in order to satisfy the business functionality. This call sequence is represented on Fig. 3.

Let's not forget that

A^C = if A is not true, i.e. component A is unavailable and that
$P(A^C) = 1 - P(A)$

In this case, the system's overall availability, PS can be

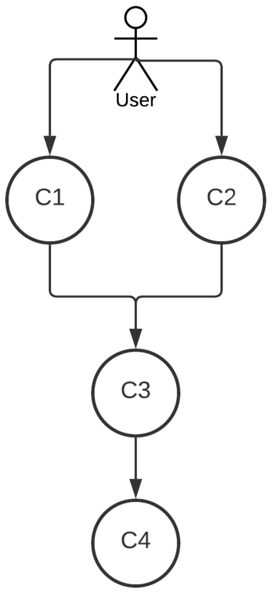


Fig. 3. Figure presenting Example 2

defined as:

$$\begin{aligned} &PS\{C1, C2, C3, C4\} = P((C1 \vee C2) \wedge C3 \wedge C4) \\ &= P(\overline{C1^C \wedge C2^C} \wedge C3 \wedge C4) \\ &= (1 - (1 - P(C1)) \times (1 - P(C2))) \times P(C3) \times P(C4) \end{aligned} \tag{4}$$

III. Availability Modeling with Fault Tree Analysis

While the simple techniques well known from probability theory are sufficient to model simple sequentially dependent and symmetrical (or parallel) redundant cases, they are not deemed sufficient to model complex systems with different asymmetrical call paths, like the one presented on Fig. 4.

For this purpose we need to introduce a technique called minimal cut sets or fault tree analysis. Fault tree analysis has been studied for a long time since [7] and as described in [8], first appearing in security analysis in the aerospace industry. We will apply this technique to model a complex IT Software System.

A. *Minimal cut set approach*

In the followings under *cut set* and *minimal cut set* we mean:

Definition III.1 (Cut set)**.** Cut sets are the unique combinations of component failures that can cause system failure.

Definition III.2 (Minimal cut set)**.** Specifically, a cut set is said to be a minimal cut set if, when any basic event is removed from the set, the remaining events collectively are no longer a cut set.

We will model the system's Availability PS using the following process:

Step 1: We define the system components $C1, ..., Cn$. Software Architects may use Domain-Driven Design (DDD) for this purpose, as suggested by [9].

Step 2: Reduce components to s-independent components. For each Ci and Cj, if their availability have inter-dependency, combine them to a new Ci'. Let $P(Ci')$ be the combined availability probability of Ci and Cj. Then remove Ci and Cj from the set of system components used for the modeling.

Step 3: Define the critical call sequence(s). This is often achieved by taking the targeted most critical business functionalities (shouldn't be more than a handful in order to keep the complexity of modeling realistic).

Step 4: For each targeted critical business functionality define the list and order of components that are required to satisfy the given functionality. Create edges between the respective components accordingly.

Step 5: Define sets of components (cut sets) that cause system failure based on the current dependency graph.

Step 6: Reduce the list of cut sets to minimal cut sets. For each cut set S, if S is not a minimal cut set, remove S from the list of cut sets.

Step 7: Calculate the probability of system failure PS^C by adding all failure probabilities $P(S^C)$ for each S in the list of cut sets.

Step 8: The overall system's availability is obtained by $PS = 1 - PS^C$.

B. Example with Fault Tree Analysis

Let us assume that in **Step 2** we have reduced the components to be modeled to $C1, ..., C6$ and in **Step 3** we achieve the dependency graph as presented on Fig. 4. $C1$ could be our frontend application, $C2$ and $C3$ may be our middleware, $C4$ and $C5$ are our backend components, while $C6$ could be our database.

In **Step 5** we list the possible failure cases:

- $\{Ci\}$ for $i \in \{1, 6\}$
- $\{Ci$ AND $Cj\}$ $\setminus$ {(C2 AND C4), (C2 AND C5), (C3 AND C5)} for $i, j \in \{1, \ldots, 6\}$ and $i \neq j$
- $\{Ci$ AND Cj AND $Ck\}$ for $i, j, k \in \{1, \ldots, 6\}$ and $i \neq j \neq k$

  ...
- $C1$ AND $C2$ AND $C3$ AND $C4$ AND $C5$ AND $C6$

Let us follow **Step 6** and reduce the above list of possible failure cases to just contain minimal cut sets while verifying our dependency graph on Fig. 4.:

- $C1$
- $C6$
- $C2$ AND $C3$
- $C3$ AND $C4$
- $C4$ AND $C5$

In **Step 7** we can model the failure probability PS^C:

$$\begin{aligned} & PS^C\{C1, C2, C3, C4, C5, C6\} \\ & = P(C1^C \vee C6^C \vee (C2^C \wedge C3^C) \ldots) \\ & = P(C1^C) + P(C6^C) + P(C2^C) \times P(C3^C) \\ & + P(C3^C) \times P(C4^C) + P(C4^C) \times P(C5^C) \end{aligned} \tag{5}$$

Then in final **Step 8** using the combined failure probability obtain system availability PS as:

$$PS = 1 - PS^C \tag{6}$$

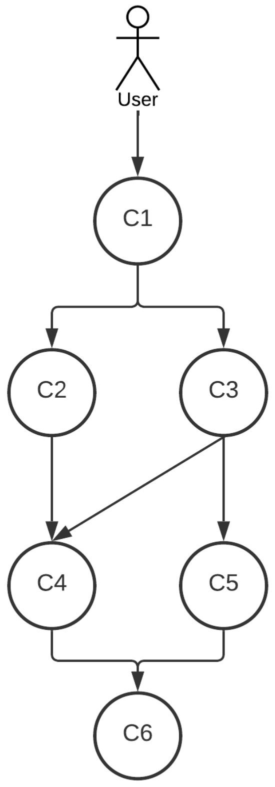


Fig. 4. Figure presenting Example 3, a complex example modeled using minimal cut set approach.

C. Real-life Example with a Cloud System

In this section we present a real-life example of a cloud-hosted system and will model its availability. Let us take an architecture and map to appropriate cloud services - we will take Amazon Web Services (AWS) for this example - and present a dependency graph as on Fig. 5.

For easier reference we have combined components to component groups, referred as $CG1...CG4$. We have also added the availability SLA values for each AWS component as obtained from AWS online reference [10].

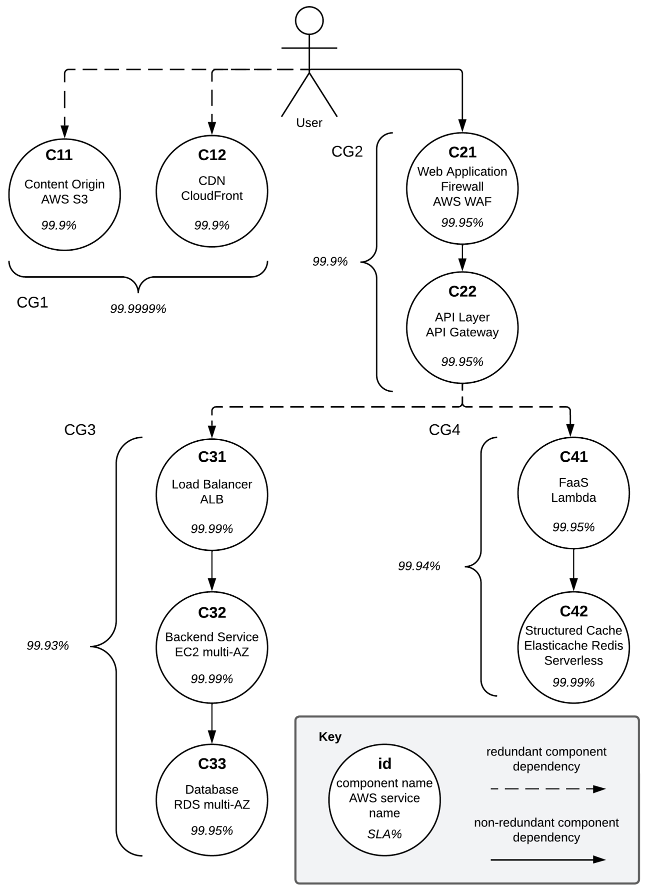


Fig. 5. Single region (multi-AZ) AWS example, presenting a real-life system, along with Service Level Agreement values from AWS SLAs page [10].

1) Availability modeling with Fault Tree Analysis:

Let us assume that in **Step 2** and in **Step 3** we have reduced the system's components to be modeled to the ones presented on Fig. 5 and we achieved the dependency graph as presented. Note that for transparency we kept our example rather simple. While naturally this example would be easy to be extended for example by adding a notification system which working asynchronously does not impact the availability of the system by not being on the critical path.

In **Step 5** we list the possible failure cases and reduce them to minimal cut sets in **Step 6**:

- $C21$
- $C22$
- $C11$ AND $C12$
- $C31$ AND $C41$
- $C31$ AND $C42$
- $C32$ AND $C41$
- $C32$ AND $C42$
- $C33$ AND $C41$
- $C33$ AND $C42$

In **Step 7** we calculate the system failure PS^C by adding all failure probabilities in the list obtained in **Step 6**.

$$\begin{aligned}
&PS^C\{CG1, CG2, CG3, CG4\}\\
&= P(C21^C \vee C22^C \vee (C11^C \wedge C12^C) \dots)\\
&= P(C21^C) + P(C22^C) + P(C11^C) \times P(C12^C)\\
&+ P(C31^C) \times P(C41^C) + P(C31^C) \times P(C42^C) \dots\\
&= 0,0005 + 0,0005 + 0,000001 + 0,00000005 + 0,00000001\\
&= 0,00000005 + 0,00000001 + 0,00000025 + 0,00000005\\
&= 0,00100142
\end{aligned} \tag{7}$$

In **Step 8** We obtain the overall system's availability by $PS = 1 - PS^C$, thus:

$$\begin{aligned}
&PS\{CG1, CG2, CG3, CG4\}\\
&= 1 - PS^C\{CG1, CG2, CG3, CG4\}\\
&= 1 - 0,00100142\\
&= 0,99899858 \approx 99,8999\%
\end{aligned} \tag{8}$$

Based on the modeled results it is also easy to see that the application of redundancy has positively benefited component groups $CG1$ and $CG3$ and $CG4$ combined, thus only a slight decrease is visible below $CG2$.

2) Availability modeling with classic probability theory method:

In order to prove our previous calculation, we will calculate the result using basic probability theory methods presented in earlier sections on the same example presented on Fig. 5.

$$\begin{aligned}
&PS\{CG1, CG2, CG3, CG4\}\\
&= P(CG1 \wedge CG2 \wedge (CG3 \vee CG4))\\
&= P((C11 \vee C12) \wedge (C21 \wedge C22) \wedge\\
&(C31 \wedge C32 \wedge C33) \vee (C41 \wedge C42))\\
&= P(\overline{C11^C \wedge C12^C} \wedge (C21 \wedge C22) \wedge \dots)\\
&= (1 - (1 - P(C11)) \times (1 - P(C12))) \times P(C21) \times \dots\\
&= (1 - (1 - 0,999 \times 1 - 0,999)) \times (0,9995 \times 0,9995) \times \dots\\
&= 0,999999 \times 0,99900025 \times P(\overline{CG3^C \wedge CG4^C})\\
&= 0,998999251 \times (1 - (0,00069989 \times 0,00059995))\\
&= 0,998999251 \times 0,9999995801\\
&= 0,998998832 \approx 99,8999\%
\end{aligned} \tag{9}$$

We can see that using a different method we were able to obtain similar results, which confirms our assumption that both modeling techniques are able to independently yield same results for the presented example. While it is also visible that for certain cases, especially if complexity grows by having more dependencies (and not just more components) the classic method becomes often times more complex to yield a result than using the previously presented fault tree analysis method, which appears to grow complexity less by the volume of dependencies and rather by the number of components.

IV. Conclusion

In this paper, after introducing the basic related concepts, we presented a simple and an advanced technique using fault tree analysis and Markov cut sets to model Availability in IT software architectures, based on techniques previously presented for other industries. We also provided an algorithmic approach to obtain modeled Availability probability based on a software architecture design. We also presented with examples how calculation can be done with both classic method and with fault tree analysis and that for certain cases where there are more dependencies between components, calculation with fault tree analysis can be beneficial over the classic method, while classic method should still be used for simpler cases.

It is also worth noting, that once a software system has been built and deployed to a live environment, it is definitely suggested to monitor the running system's availability and obtain a measurement in such a way. Later it can be compared with the modeled availability metric to test the precision of the modeling.

Acknowledgment

Gergely Kocsis was supported by the project TKP 2021 NKTA of the University of Debrecen Project no. TKP-2021-NKTA-34 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the TKP 2021 NKTA funding scheme.

References

[1] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 4th ed., Addison-Wesley, 2022.

[2] Nurbojatmiko, E. K. Budiardjo, and W. C. Wibowo, Slr on Identification & Classification of Non-Functional Requirements Attributes, and Its Representation in Functional Requirements, In Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence (CSAI '18). Association for Computing Machinery, New York, NY, USA, 151–157, 2018.

[3] M. Glinz, "On Non-Functional Requirements," The 15th IEEE International, 15-19 October, 2007, pp. 21-26.

[4] ISO/IEC 25010, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models

[5] J. Yuan, A Conditional Probability Approach to Reliability with Common-Cause Failures, IEEE Transactions on Reliability, vol. R-34, no. 1, pp. 38–42, April 1985.

[6] C. Singh, J. Mitra, Reliability Evaluation in Transmission Systems, IEEE Transactions on Power Apparatus and Systems, vol. PAS-100, pp. 2719–2725, July 1981.

[7] H.A. Watson, Launch control safety study, Bell Telephone Laboratories, Murray Hill, N.J. USA, 1961.

[8] C. Ericson, Fault Tree Analysis - A History, Proceedings of the 17th International Systems Safety Conference, 1999.

[9] E. Evans, Domain Driven Design, Tackling Complexity in the Heart of Business Software, Addison-Wesley, 2002.

[10] AWS Service Level Agreements (SLAs). [Online]. Available: https://aws.amazon.com/legal/service-level-agreements/?aws-sla-cards (Accessed Jul. 2024.)