Architecture as Code

Alessio Bucaioni*, Amleto Di Salle† Ludovico Iovino†, Patrizio Pelliccione†, Franco Raimondi†
* Mälardalen University (Sweden), *alessio.bucaioni@mdu.se*
† Gran Sasso Science Institute (Italy), *name.surname*@gssi.it

***Abstract*—After more than thirty-five years of research and development in software architecture, several fundamental challenges remain unsolved. First, despite the importance of having a well-defined architecture description aligned with the system, inconsistencies and misalignments are still prevalent. Second, although numerous languages exist to describe architectures, none have achieved widespread use or recognition as a de facto standard. Third, while architecture is dynamic and evolving, with architectural decisions often made by non-architect stakeholders, there are no universally accepted methodologies to capture emergent aspects and incorporate them into the architecture.**

In this paper, we explore the emerging concept of architecture as code. Inspired by the success of infrastructure as code, which enables infrastructure management in a codified, automated, and repeatable manner, architecture as code aims to bring similar benefits to software architecture. To the best of our knowledge, this is the first scientific paper to study this concept in depth within the context of software architecture, providing a comprehensive description and analysis of its characteristics. We also investigate how architecture as code is implemented and applied in practice.

***Index Terms*—Architecture as code, inconsistencies, architecture drift, architectural debt**

I. Introduction

Software architecture plays a critical role in the development of complex systems [43]. This importance extends to agile practices [2], [6], where it is essential to balance the effort spent on defining the upfront architecture (planned before development begins) and accommodating the emerging architecture (decisions made throughout the development process). However, empirical studies show that architectural descriptions and decisions are often not fully considered during development, at least not to the desired extent [18], [26], [49]. In addition, despite the large number of architectural languages proposed by the community, they are not widely adopted, and we are still far from having an accepted de facto standard [36].

Inconsistencies between architecture descriptions and implementation have varying levels of criticality and severity [49]. Inconsistencies related to wording and language are typically less critical, whereas those involving interface specifications, rules, constraints, patterns, and guidelines can have more severe consequences [49]. These inconsistencies are often discovered late in the development process [49]. Furthermore, over 70% of non-conformance issues between architecture descriptions and source code are due to flaws in the documentation [20]. The problem of inconsistency between architecture and implementation, as well as the challenge of achieving better alignment between architecture descriptions and implementation, has been explored in the context of architecture decay [34], architecture drift [38], and architecture erosion [17], [38]. This issue has also been recognized as an anti-pattern: creating a perfect architecture for the wrong system [31]. The problem is further complicated by the fact that architecture is a living entity, incorporating both upfront and emergent aspects and that some decisions are made in collaboration with stakeholders who do not hold the title of architects [45]. Researchers have proposed guidelines for software architects to improve the consistency and usefulness of architecture descriptions [49]. Other researchers highlight the need for mechanisms for increasing architectural awareness and having more case studies [4]. Another branch of research related to the problem of architectural inconsistencies concerns architectural debt: when architectural inconsistencies are not counteracted and fixed in architectural descriptions, they become architectural debt [32].

Years of research and empirical studies highlight that (i) regardless of the nature of the system or the development process, inconsistencies are almost inevitable, (ii) the consequences of inconsistencies can be significant, and (iii) proposed solutions are not one-size-fits-all. The key question remains unanswered: *how can we guarantee consistency between software architecture descriptions and implementation and maintain this consistency over time, even during system evolution?* While various theoretical solutions could potentially address consistency, the real challenge lies in finding practical solutions, technologies, or tools that are effective in practice—ones that clearly demonstrate a Return on Investment (ROI), require minimal training, and have a limited impact on organizational processes.

Infrastructure as Code (IaC) aims to automatically provision and support computing infrastructure, including defining system dependencies and provisioning local and remote instances using code rather than manual processes and settings [40]. IaC is considered a fundamental pillar for implementing DevOps practices [40], enabling infrastructure components, such as operating systems, database connections, and storage, to be automated and aligned with development needs. When IaC includes descriptions of aspects like system structure, de-

This work has been partially funded by (a) the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem, grant ECS00000041 - VITALITY – CUP: D13C21000430001, (b) the MUR (Italy) Department of Excellence 2023 - 2027, (c) the European HORIZON-KDT-JU research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems", HORIZON-KDT-JU-2023-2-RIA, Proposal number: 101140216-2, KDT232RIA_00017, (d) the Swedish Knowledge Foundation through the MoDEV project (20200234), and (e) the Sweden's innovation agency Vinnova through the project iSecure (202301899)

composition, application components, and their relationships, it starts to overlap with the software architecture domain, shifting to Architecture as Code (AaC). Architecture as Code aims to describe, document, manage, and maintain software architecture through a human- and machine-readable, version-controlled code-base integrated with implementation. The concept of AaC appears promising for resolving inconsistencies and preventing them through a seamless process accessible to various stakeholders, promoting efficient development. It also provides a way to treat architecture as a living artifact during system development, incorporating both upfront and emergent aspects within the same description. However, AaC has not been extensively explored in scientific literature or within the research community. Instead, it is a topic that is gaining increasing attention in the industrial sector despite the lack of a clear and widely accepted definition of AaC and its associated concepts.

In this paper, we aim to clarify the concept of AaC by providing a precise definition, describing its characteristics, and identifying the level of architectural expressiveness that can be formulated in an AaC description. More specifically, this paper addresses the following research questions:

- *RQ1: What is AaC, and what are its main characteristics?* To address this question, we provide a definition of AaC along with a description of its characteristics and challenges. We gathered this information through a multivocal literature review, including scientific papers and grey literature such as white papers, blogs, and websites.
- *RQ2: How is AaC implemented and applied in practice?* Our response to RQ2 is twofold. First, we examine the supporting tools and technology providers identified through the multivocal literature review. Second, we provide an in-depth analysis of how AaC is implemented in the context of Amazon AWS. As it is better detailed in the next sections, Amazon AWS was selected for this analysis because of its comprehensiveness and integrated ecosystem that aligns well with key AaC characteristics, such as automation, version control, and integration with CI/CD practices.

The paper is structured as follows. Section II describes the research methodology we followed to conduct this study. Section III addresses RQ1 by describing AaC and its characteristics. Section IV and Section V address RQ2 by discussing tools, and technologies for AaC, and how it is realized in the context of Amazon AWS. Section VI reviews related works, while Section VII discusses the findings of the work and concludes the paper with final remarks and future research directions.

II. Research methodology

Our research process employs a combination of complementary methods, including a systematic literature review (SLR) [30], a gray literature review [24], and use case analysis [28]. The SLR helps us identify the current state of the art based on scientific papers. Given the limited number of scientific studies available in this field and recognizing that practitioners also widely discuss the topic in blogs, white papers, and other non-academic sources, we supplemented the SLR with a grey literature review to capture the state of practice. Additionally, we conducted a use case analysis based on a large international company to gain deeper insights into practical implementation.

The process we followed consists of three phases: planning, conducting, and documenting. The main objectives of the planning phase were to establish the need for this study on AaC, identify the Research Goal (RG) and Research Questions (RQs), and define the research protocol for carrying out the study systematically. A detailed research protocol was the primary output of this phase. In the conducting phase, we performed activities outlined in the research protocol, including search and selection, data extraction form definition, data extraction, and data analysis. The main objectives of the analyzing phase were to analyze and document potential threats to validity and to record the study's results. To facilitate independent replication and verification, we provide a complete replication package [1] containing search and selection data, and the list of primary studies.

A. Research goal and questions

Following the Goal-Question-Metric (GQM) approach [7], we defined the research goal, which is presented in Table I.

TABLE I
Research goal expressed using the GQM perspectives.

Purpose	Identify, classify, and evaluate
Issue	definitions, founding concepts, supporting tools and challenges
Object	of architecture as code
Viewpoint	from the point of view of researchers and practitioners.

We refined the goal in the research questions that are described in the introduction.

B. Search and selection

Following the steps illustrated in Figure 1, we collected relevant research studies for our investigation. We conducted two parallel reviews: one focused on peer-reviewed literature and another on gray literature. Both reviews followed a consistent process. For simplicity, we refer to included studies from either source as primary studies, unless otherwise specified. For the peer-reviewed literature, we selected four major scientific databases: IEEE Xplore, ACM Digital Library, SCOPUS, and Web of Science, along with the scholarly search engine Google Scholar. For the gray literature search, we used Google that accounts for 92.2% of global web searches.[1] We crafted a search string based on our research goal and questions to capture as many relevant studies as possible, given that AaC is a relatively new field. The search string used was:

"Architecture as Code"

[1]https://gs.statcounter.com/search-engine-market-share

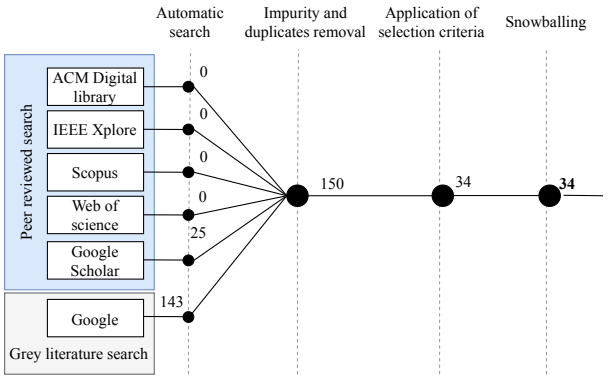

Fig. 1. Search and selection

The automated search for peer-reviewed literature, limited to studies from 2019 onward, initially identified 25 potential studies. The grey literature search yielded 143 relevant entries. Google automatically filtered similar entries to reduce duplicates and impurities. After further manual filtering, we removed impurities and duplicates and refined the total set to 150 potential primary studies. Using selection criteria suggested by Ali and Petersen [3], we systematically applied inclusion and exclusion criteria to ensure objective selection of studies. The criteria used are as follows:

Peer-reviewed literature	
Inclusion criteria	
I1	Studies focusing on software engineering and architecture
Exclusion criteria	
E1	Studies not in English
E2	Studies not peer-reviewed
E3	Studies whose full-text is not available
E4	Studies not focusing on AaC
Grey literature	
Inclusion criteria	
I1	Websites focusing on software engineering and architecture
Exclusion criteria	
E1	Websites referring to repositories only containing code
E2	Websites referring to impure sources such as advertisement or Pinterest pages
E3	Websites referring to peer-reviewed literature[2]
E4	Websites not in English
E5	Websites addressing complementary aspects of software architecture and using AaC only as buzzword
E6	Websites older than 2019

To proceed to the next stage, studies had to meet all inclusion criteria and none of the exclusion criteria. Applying these criteria resulted in a new set of 1 peer-reviewed study and 33 gray literature sources. We then conducted closed recursive backward and forward snowballing [48] to mitigate potential construct validity biases [25], but no additional sources were identified. The final set of 34 primary studies is listed in the Primary Studies appendix.

[2]We kept the peer-reviewed and grey literature selection processes separated due to the corresponding guidelines; however, we made sure that any potentially relevant peer-reviewed entry excluded for the grey literature was already considered in the peer-reviewed literature selection.

C. Data extraction

To extract and collect data from the primary studies, we designed a data extraction form, as shown in Table II. The data extraction form consists of two facets, each corresponding to a research question. RQ1 includes three categories: definition, foundational concepts, and challenges. RQ2 focuses on two categories: tools and providers. For all the categories, we first collected all the data from the primary studies and then applied the grounded theory methodology [12]. This method systematically breaks down data, categorises it, and establishes connections between categories to identify emerging themes. This approach enabled us to organize qualitative data in a structured manner, facilitating a deeper understanding of foundational concepts, challenges, tools, and providers. For studies involving videos, we first extracted the entire video transcript using NotebookLM[3], and then performed the data extraction on the transcript.

Facet	Category	Description	Value
RQ1	Definitions	Definitions as identified in the studies	String
	Foundational concepts	Foundational concepts as identified in the studies	String
	Challenges	Challenges as identified in the studies	String
RQ2	Tools	Tools as identified in the studies	String
	Providers	Providers as identified in the studies	String

TABLE II
DATA EXTRACTION FORM

D. Data analysis and synthesis

We analyzed, and synthesized the extracted data following the guidelines of Cruzes et al. [16]. In this study, we used both quantitative and qualitative analyses, combining content analysis [22] and narrative synthesis [42]. We began by analyzing each primary study individually, classifying key features based on the parameters in the data extraction form. Then, we examined the entire set of primary studies to identify and interpret emerging patterns.

E. Threats to validity

One primary threat to external validity in systematic reviews is that the selected papers may not fully represent the state-of-the-art and practice of AaC. To address this, we conducted a search across five major software engineering databases and supplemented the automatic search with recursive backward and forward snowballing, as well as a gray literature search. While we acknowledge that the search string construction and inclusion/exclusion criteria are simple, this was an intentional choice to ensure inclusivity given the limited number of studies in this field. Additionally, we restricted our review to English-language studies, as English is the de facto standard for scientific work in computer science and software engineering. While this minimizes language-related validity threats, it does not entirely eliminate them. Another potential limitation

[3]https://notebooklm.google.com

is the use of alternative terms for AaC, which may have constrained the search scope. To mitigate this, we expanded our discussion to include related works where relevant.

To address internal validity, we followed established guidelines for systematic and multi-vocal studies, minimizing potential bias. We used descriptive statistics and cross-analyzed extraction form categories, performing sanity checks to ensure data consistency.

Construct validity may be affected by poorly designed search strings; however, we used a straightforward string for both peer-reviewed and gray literature searches, requiring minimal adjustment.

For conclusion validity, we consistently applied and documented well-defined processes and provided a publicly accessible replication package for reproducibility [1]. All authors contributed to defining the extraction form and engaged in data extraction, analysis, and synthesis, utilizing established taxonomies and values derived from primary studies.

III. What is AaC, and what are its main characteristics? (RQ1)

In this section, we address the first research question: What is AaC, and what are its defining characteristics? We begin by providing a clear definition of AaC, followed by an in-depth description of its primary characteristics. Subsequently, we discuss key open challenges. The definition, characteristics, and challenges were synthesized through an analysis of the 34 primary studies

A. AaC definition and characteristics

From the primary studies, we extracted 11 definitions. Due to space limitations, we have omitted the full definitions here; however, they are available in the provided replication package [1]. Among these definitions, four emphasized the concept of a "readable and version-controlled codebase" [P1], [P8], [P12], [P22]. Automation was another key aspect emphasized by several definitions, each offering a different perspective on its role [P1], [P8], [P9]. For instance, some definitions focused on integrating automation with DevOps practices, stressing that "architectural artifacts should emerge from the code pipeline and be continuously updated throughout the programming lifecycle" [P9]. Beyond technical aspects, stakeholder engagement was identified as a critical factor by three definitions. These definitions emphasized that for AaC to provide real value, stakeholders must be willing to adopt and use it effectively. Hence, extensibility was also a prominent theme, reflecting the need for adaptability given the complexity of software architectures [P2], [P4], [P21]. Based on these 11 definitions, we can define AaC as follows:

Definition 1 (Architecture as Code)**.** *Architecture as Code is an approach where software architecture is continuously defined, managed, and evolved through a machine-readable, version-controlled code-base. This practice ensures alignment between design and implementation by directly embedding prescriptive architectural elements, such as structure, rules, and constraints, within the code. Leveraging automation and development practices, AaC supports the continuous evolution of architecture and the generation of up-to-date architectural artifacts throughout the software lifecycle. By integrating familiar tools and emphasizing accessibility, AaC fosters active engagement from developers and stakeholders, making architectural practices both adaptable and collaborative.*

Table III summarizes the main characteristics of AaC: (1) code-centric approach to define and manage architecture, (2) automation-driven evolution of architecture, and (3) approachability and stakeholder engagement.

Characteristics	Aspects
(1) Code-Centric approach to define and manage architecture	Diagram as code (modeling)
	Alignment between architecture and code
	Versioning
	Traceability
(2) Automation-driven evolution of architecture	Automatic generation of architectural diagrams, documentations, etc.
	Automatic analysis and quality assessment
	Integration with development practices
(3) Approachability and stakeholder engagement	Accessible languages (DSLs)
	Structured vocabulary
	Ubiquitous language
	Software development life cycle models fostering stakeholder engagement
	Software development life cycle models supporting emergent architecture

TABLE III
AaC main characteristics and foundational aspects.

Characteristic 1: code-centric approach to define and manage architecture. This characteristic highlights how AaC enables defining software architecture directly in code, ensuring consistent alignment between conceptual design and implementation [P1]–[P3], [P5], [P8], [P11], [P14], [P15], [P17]–[P19], [P21], [P24], [P25], [P28], [P29], [P33], [P34]. By using code as the single source of truth for the architecture [P1], [P8], [P12], [P22], AaC merges design and development, allowing both to evolve seamlessly and reducing architectural debt – the misalignment between high-level design and implementation [50]. This integration is achieved by modeling architecture through machine-readable diagrams that support vertical (top-down structure) and horizontal (component interaction) traceability [P2]. Prescriptive architectural elements—such as structure, rules, and constraints—are embedded within diagrams, which are no longer static and manually-drawn representations. Instead, these diagrams are machine-readable, enabling them to be queried, transformed, tracked, and version-controlled alongside other parts of the software [P19], [P33]. This approach aligns with recent studies examining the interplay of Model-Driven Engineering (MDE) and software architecture, particularly in modeling architectures through models and metamodels [9]. The result is a living architecture that evolves with the software, continuously refining architectural decisions fostering adaptability and precision. An example is the AaC Modeling Language [P2], which enables architects to maintain a collection of machine- and human-readable YAML files. Further examples are the

C4 model that supports this structured approach with layered diagrams, such as System Context and Container Diagrams, which clarify static structure and system relationships [P13].

Characteristic 2: automation-driven evolution of architecture. This characteristic highlights the role of automation in ensuring that architecture evolves dynamically, supporting agility and adaptability in architectural practices [P2], [P5]–[P7], [P9], [P17], [P26], [P33]. Machine-readable diagrams serve as the foundation of this automation-driven evolution, integrating seamlessly with development practices such as CI/CD pipelines and employing Command-line Interface (CLI) tools for the automatic generation of architectural diagrams, documentation, analysis, and more. For instance, tools like C4 Inteflow can start from a C4 model and auto-generate architecture models (as code) in formats such as C#, YAML, or JSON [P17]. Additionally, these tools can auto-generate comprehensive architecture documentation, incorporating defined properties and custom attributes from the architecture model to enhance traceability and provide richer contextual understanding [P17]. Another example is the use of the AaC Modeling Language for implementing automated quality assurance in pipelines using diagrams [P2]. Just as automation is crucial for effective infrastructure management, it is equally vital for the success of AaC [P6]. Without automated workflows, AaC risks to become static and quickly outdated. Therefore, AaC extends beyond traditional Infrastructure as Code (IaC), incorporating elements of Configuration as Code (CaC) and integrating with CI/CD and other development practices to ensure continuous evolution. Automation has been extensively studied by the research community in recent years, with contributions focusing on automatic code generation [46], automating architectural conformance checking through CI and Model-Driven Engineering (MDE) [10], [39], and more.

Characteristic 3: approachability, extensibility, and stakeholder engagement Making AaC accessible to all stakeholders is essential to embed architectural practices naturally into daily activities, fostering broader participation [P2], [P4], [P13], [P21], [P30], [P32], [P34]. This can be achieved by integrating AaC into standard developer workflows, using familiar tools, promoting adaptability, and ensuring extensibility. Concepts like "approachability for stakeholders" emphasize the importance of using accessible languages and tools that enable contributions from all team members, not just architects [P2], [P13]. Examples include the use of domain-specific languages (DSLs) like the AaC Modeling Language, where tools such as C4InterFlow can generate diagrams from code written in C#, YAML, and more, effectively engaging developers, architects, and non-technical stakeholders alike [P13]. Another example is the use of established patterns like Enterprise Integration Patterns (EIPs), which provide a structured vocabulary for describing message and event flow within distributed systems [P4]. Implementing EIPs establishes a ubiquitous language that goes beyond specific technologies, fostering clear communication and collaboration across teams. The use of DSLs has become a de-facto standard practice, widely recognized through various initiatives by both the research community and international industrial consortia [5], [9]. Traditional architectural practices should be embedded within Software Development Life-cycles (SDLCs) to promote stakeholder engagement and support a fully evolutionary, incremental architectural approach—commonly referred to as emergent architecture [P30], [P32], [P34]. Agile SDLCs, such as Scrum, XP, and DevOps, exemplify models that balance emergent and intentional design, continuously developing and extending the architectural foundation that supports the development of future business value [11], [35].

B. Challenges

Table IV presents the identified challenges, organized into overarching categories for clarity. The table also maps the challenges to the AaC characteristics presented in Section III.

Challenge 1: ensuring conformance. This challenge is two-fold, involving conformance among architectural artifacts and adherence to regulatory requirements in regulated industries [P8], [P10], [P19], [P31]. Although these sectors adopt structured approaches to comply with strict policies [P8], they often lack mechanisms to verify the implementation of the agreed-upon design or to monitor divergence over time. AaC may exacerbate this issue if emergent design is not balanced with intentional design, or if alignment and traceability processes are insufficient. Similarly, maintaining conformance between diagrams and overarching structures, such as patterns or DSLs, remains challenging. Recently, several works have been proposed to address conformance in software architectures [10], [39].

Challenge 2: minimizing drift. Drift occurs when changes in code, infrastructure, and other components are not accurately reflected in the architecture, causing a divergence between documented and actual system states [P7], [P8], [P11]. For example, architecture drift refers to inconsistencies between the code and documented architecture [P8], [P11], while infrastructure drift pertains to mismatches between defined configurations and the actual infrastructure state [P7]. Both types of drift often result from manual changes, misconfigurations, or unauthorized modifications, leading to issues in conformance, operations, and reliability. Drift can be seen as the flip side of technical debt: while technical debt often results from intentional choices favoring speed over structure [50], drift represents the unintentional byproduct of system evolution.

Challenge 3: ensuring scalability. While AaC practices are beneficial for smaller, simpler systems, they often struggle to scale effectively for larger, more complex architectures [P5], [P29]. Scalability issues arise as AaC practices become cumbersome and less efficient with extensive systems, limiting their applicability. For example, users report that tools like PlantUML, Mingrammer's Diagrams, Structurizr, and Eraser work well for small diagrams but become unwieldy with increased complexity, leading to messy, hard-to-read visuals [P5]. Additionally, limited customization options in these tools further complicate efforts to maintain clarity at scale [P5].

Categories	Challenges	Code-Centric approach to define and manage architecture	Automation-driven evolution of architecture	Approachability and stakeholder engagement
Conformance and drift management	(1) Ensuring conformance	✓	✓	
	(2) Minimizing drift	✓	✓	
Quality Attributes	(3) Ensuring scalability		✓	
	(4) Ensuring performance		✓	
Modeling complexity	(5) Learning new diagramming language	✓		✓
	(6) Balancing abstraction	✓		✓
Process	(7) Supporting collaboration		✓	✓
	(8) Balancing upfront and emergent design	✓	✓	✓
	(9) Steep learning curve		✓	✓
	(10) Selecting tools		✓	✓

TABLE IV
CHALLENGES AND THEIR MAPPING WITH CHARACTERISTICS.

Challenge 4: ensuring performance. Performance challenges similarly reflect the limitations of AaC approaches when applied to sizable systems, where the complexity and volume of code required to represent the architecture can hinder responsiveness and manageability [P22].

Challenge 5: learning a new diagramming language. Many diagramming languages and modeling tools have a steep learning curve, which can hinder adoption and limit effective use within teams [P5], [P7]. Team members often struggle to select suitable tools or master the necessary languages, leading to inconsistencies in architectural representations and increasing the on-boarding burden. As models scale and require more detailed configurations, they often demand specialized knowledge and complex setups, amplifying these challenges. This aligns with findings from surveys on architectural languages, which indicate that steep learning curves frequently deter practitioners [36].

Challenge 6: balancing abstraction. Another major challenge in modeling complexity is achieving the right level of abstraction [P25], [P31]. High-level models that omit essential details may lack sufficient insight, while overly detailed models can overwhelm users, hindering a clear understanding of the system. Tools like PlantUML, Structurizr, and Mermaid are effective for simpler diagrams but often fall short in supporting clarity and organization for large, complex systems. Consequently, these tools can produce cluttered, less readable diagrams that impede comprehension, especially when models need to capture extensive architectural details or cross-domain interactions.

Challenge 7: supporting collaboration. Collaboration is crucial as AaC involves cross-functional teamwork, and effective communication is needed to maintain alignment on architecture goals [P31]. A significant challenge with AaC is the inherent difficulty of collaborative editing, as it still relies on tools primarily designed for code management [P31]. While version control systems like Git are essential for collaborative coding, they often introduce friction when applied to tasks beyond coding, such as diagramming and documentation. Concurrent edits to the same document frequently lead to merge conflicts that require time-consuming resolution, particularly with non-technical content, potentially discouraging team members from collaborative editing.

Challenge 8: balancing upfront and emergent design. Balancing upfront and emergent design within AaC is challenging, as it requires flexible planning that accommodates new insights without locking the system into a rigid structure, while also allowing for future evolution [P30]. AaC must enable rapid adaptations for continuous delivery while maintaining enough intentional structure to avoid chaos. The need for clear communication and decentralized decision-making adds complexity, as architects must convey a cohesive vision that empowers teams to innovate locally without disrupting the architectural foundation. This balance is crucial in AaC to prevent inconsistencies and ensure the architecture evolves smoothly, aligning both immediate and long-term business goals.

Challenge 9: steep learning curve. One of the primary challenges in adopting AaC is the steep learning curve [P7], [P28]. This challenge stems from the variety of tools and techniques involved, requiring teams to familiarize themselves with new methodologies and best practices, which can create significant initial hurdles. For example, desktop-based modeling tools [P7], while benefiting from an underlying data model that may be familiar, often require substantial effort to learn and master.

Challenge 10: selecting tools. Selecting the right tools and technologies for infrastructure provisioning, configuration management, and automation is crucial for the success of AaC initiatives [P7]. Poor tool choices can lead to inefficiencies, compatibility issues, and maintenance challenges. Conducting thorough research and evaluating tools based on scalability, flexibility, community support, and integration capabilities can support informed decisions. Additionally, considering future scalability and extensibility needs helps ensure alignment with long-term goals.

IV. WHICH ARE THE CURRENT SOLUTIONS SUPPORTING AAC? (RQ2)

This section, along with Section V, addresses the second research question on how AaC is implemented and applied in practice. Specifically, this section reviews current solutions

that support AaC, while Section V provides a use case analysis of AaC implementation within the Amazon AWS ecosystem. From our analysis of 34 primary studies, we identified over 40 tools and technologies, which we grouped into three categories: (1) design tools, (2) infrastructure tools, and (3) CI/CD and automation tools. Due to space constraints, the complete list of tools is omitted from this section, but is available in the replication package [1].

Design tools. This category encompasses a range of technologies for specifying, configuring, modeling, visualizing, and diagramming software architectures. These tools play a crucial role in implementing AaC by representing architectural structures, relationships, and behaviors in a standardized, codified form. Together, these tools facilitate precise and scalable architecture documentation and visualization, ensuring that architectural information is accessible, consistent, and well-aligned with software development practices. Tools in this category include PlantUML, C4, C4 sharp, Structurizr, Graphviz, Cloudgram, Eraser, Reflections library, Archi, MagicDraw, Software Architect, LucidChart, ArchiMate, DOT language, Vis.js, Draw.io, Visio, Mermaid, diagrams Python library, and C4-PlantUML.

Infrastructure tools. This category includes technologies for managing, provisioning, and automating infrastructure in cloud environments. These tools enable organizations to provide a foundation for defining, automating, and scaling infrastructure and configurations in a consistent and repeatable manner. By allowing infrastructure specifications to be codified, these tools facilitate and accelerate AaC. Tools in this category include Terraform, AWS CloudFormation, Azure Resource Manager, Pulumi, Kubernetes, AWS EKS, Azure AKS, Google Cloud GKE, AWS CDK, Snapblocks, Ansible, Chef, Puppet, Terraform, Pulumi, Google Cloud Platform, AWS Lambda, Azure Functions, Google Cloud Functions, Amazon EventBridge, and Google Cloud.

CI/CD and automation tools. This category includes technologies that streamline and automate key aspects of the software development life-cycle as CI, CD, and configuration management. These tools facilitate reliable and efficient software delivery by ensuring that infrastructure, code, and configurations are continuously monitored and aligned with development and deployment practices. Additionally, automation tools include drift detection solutions, which help maintain configuration consistency and monitor for any deviations from intended system states. Tools in this category include Jenkins, GitLab CI/CD, CircleCI, Swagger, Python, and Jackson library.

It is important to emphasize that these tools were mentioned directly in the primary sources. While these tools contribute valuable functionality, their scope may be either more general than AaC or limited to specific characteristics or aspects of AaC identified in Section III. For example, design tools such as PlantUML, Structurizr, and C4 primarily support aspects like diagram as code, alignment between architecture and code, and accessible languages for representing architecture. However, these tools do not typically include capabilities for automatic generation of architectural diagrams or documentation, which limits their effectiveness in automation-driven evolution. Similarly, infrastructure tools such as Terraform, AWS CloudFormation, and Kubernetes focus on the core tasks of managing, provisioning, and automating infrastructure. These tools enable versioning and traceability of infrastructure as code, aligning with a code-centric approach. However, they generally lack features related to diagram as code or accessible languages, as their primary aim is infrastructure management rather than architectural modeling.

In the same way, CI/CD and automation tools like Jenkins and GitLab CI/CD are instrumental in achieving integration with development practices and automation-driven evolution, supporting continuous delivery and deployment processes. Yet, these tools do not contribute directly to aspects stakeholder engagement through accessible languages.

Through our analysis of the selected tools, we found that most support the 'diagram as code' aspect, offering ways to create architectural diagrams. Some tools employ DSLs in a purely AaC style, focusing solely on code-based models, such as YAML specifications, without generating diagrams. Not all DSLs are openly accessible; some tools provide documentation and explicit grammar definitions, while others do not. Only a few tools provide analysis of declared architectures, including syntax and conformance checks with language specifications. Alignment between architecture and code is typically ensured by construction, and versioning is naturally supported by the textual nature of these specifications. Integration with development practices is a feature in relatively few tools, highlighting an area with room for improvement.

Based on this analysis, we selected tools that support the highest number of AaC aspects and summarized them in Table V. Table V lists the selected tools alongside the foundational AaC aspects identified in Section III, with a tick indicating each aspect implemented by the tool. Note that two aspects —software development life cycle models fostering stakeholder engagement and software development life cycle models supporting emergent architecture— are not included in Table V, as they pertain to the process rather than the tool itself. These aspects depend on how the tool is applied within the process context. Moreover, it is worth highlighting that none of the selected tools support the aspect of structured vocabulary.

Structurizr, defined as a "diagrams as code" tool, allows the creation of multiple software architecture diagrams from a single model. It offers a standalone and a web-based DSL editor allowing you to push and pull workspaces to and from a reserved cloud service. C4Sharp is a "diagrams as code" .NET library based on the C4 Model. It works like a superset of C4-PlantUML for managing C4 Model diagrams as code (C#). C# allows the creation of diagrams from existing application code. Cloudgram generates diagrams for cloud architectures directly in a web-based app, using code in a syntax similar to the DOT language. Diagrams permits to draw cloud system architectures in Python and as the other approaches allows the versioning of the specifications. It supports the definition

Aspects	Tools								
	Structurizr	**C4Sharp**	**Cloudgram**	**Diagrams**	**Eraser**	**C4Interflow**	**arch-as-code**	**AasC**	**AWS Cloud-Formation**
Diagram as code	✓	✓	✓	✓	✓	✓			✓
Alignment between architecture and code	✓	✓	✓	✓		✓	✓	✓	✓
Versioning	✓				✓	✓	✓		✓
Traceability								✓	
Automatic generation of architectural diagrams, documentations, etc.	✓	✓	✓	✓	✓		✓	✓	✓
Automatic analysis and quality assessment						✓		✓	✓
Integration with development practices		✓			✓	✓			✓
Accessibile languages (DSLs)		✓					✓	✓	?
Structured vocabulary									
Ubiquitous language	JSON, DSL	C#	DOT	Python	DSL	JSON, C, YAML	YAML	JSON, YAML	JSON, YAML

TABLE V
TOOLS SUPPORTING AAC

of the major providers including: AWS, Azure, GCP, Alibaba Cloud and Oracle Cloud. Eraser is a diagram as code tool with a specific DSL to define architectural diagrams and generates also visual documentation. It is integrated with CI/CD tools.

C4InterFlow[4] is a comprehensive application architecture framework that allows the expression of Application Architecture as Code (AaC) and Business Processes as Code (BPaC), generating application and business architecture diagrams intended for documentation. Based on the C4 Model [47], C4InterFlow supports a DSL in C#, YAML, or JSON and includes a CLI for generating diagrams and documentation, potentially publishing them from the AaC specification. It also enables querying of the AaC specification for patterns or specific architectural elements.

Architecture-as-Code[5] allows defining systems in YAML code, featuring a modular CLI for AaC model validation and artifact generation. Although a GUI is planned, other documentation artifacts can be generated, with the DSL explained in detail to guide users.

Architecture as Code (AasC)[6] supports the Common Architecture Language Model (CALM) Specification, providing a machine- and human-readable format in JSON and YAML for defining, validating, and visualizing architectures. The documentation outlines language constructs, though the DSL is not explicitly available.

AWS CloudFormation[7] serves as AWS's core service for modeling and managing AWS and third-party resources using CloudFormation templates. These templates are YAML or JSON files that define resources and dependencies, which can be created in Python, Java, or TypeScript, or visually with AWS Infrastructure Composer[8], supporting iterative design of applications and their architecture.

It is worth noting that, while most tools cover specific characteristics or aspects of AaC, established players and cloud solution providers typically offer integrated suites of tools with a broader scope, often covering nearly the full spectrum of AaC aspects. An example of this is AWS CloudFormation as shown in Table V.

V. AMAZON WEB SERVICES CASE STUDY (RQ2)

This section demonstrates how Amazon's suite of tools supports AaC in practice. We chose Amazon for this use case analysis because of its comprehensive, natively integrated ecosystem, which aligns closely with key AaC characteristics such as automation, version control, and integration with CI/CD practices, as highlighted inTable V. Amazon Web Services (AWS) is the leading cloud provider, holding a 31% market share in Q3 2024[9], and operates on a pay-as-you-go model. Customers pay based on a combination of (auto-)scaling, hardware, operating system, software, and networking features, tailored to quality attributes like availability, redundancy, and security. AWS offers over 200 services across computing, storage, databases, machine learning, and the Internet of Things. One of AWS's most popular services is Amazon Elastic Compute Cloud (EC2), which provides a virtually unlimited cluster of computers. Other key services include storage and database solutions such as Amazon S3, Amazon Aurora, and Amazon DynamoDB, which focus on scalability, availability, security, and performance for storing objects, relational data, or NoSQL schemas. AWS Lambda exemplifies AWS's serverless offerings, allowing developers to build and deploy applications without managing underlying infrastructure, using their preferred programming languages. These services are typically deployed within a Virtual Private Cloud (VPC), a user-defined virtual network, with access

[4]https://github.com/SlavaVedernikov/C4InterFlow
[5]https://arch-as-code.org/
[6]https://github.com/finos/architecture-as-code
[7]https://aws.amazon.com/cloudformation/
[8]https://aws.amazon.com/infrastructure-composer/
[9]https://shorturl.at/FIn3w

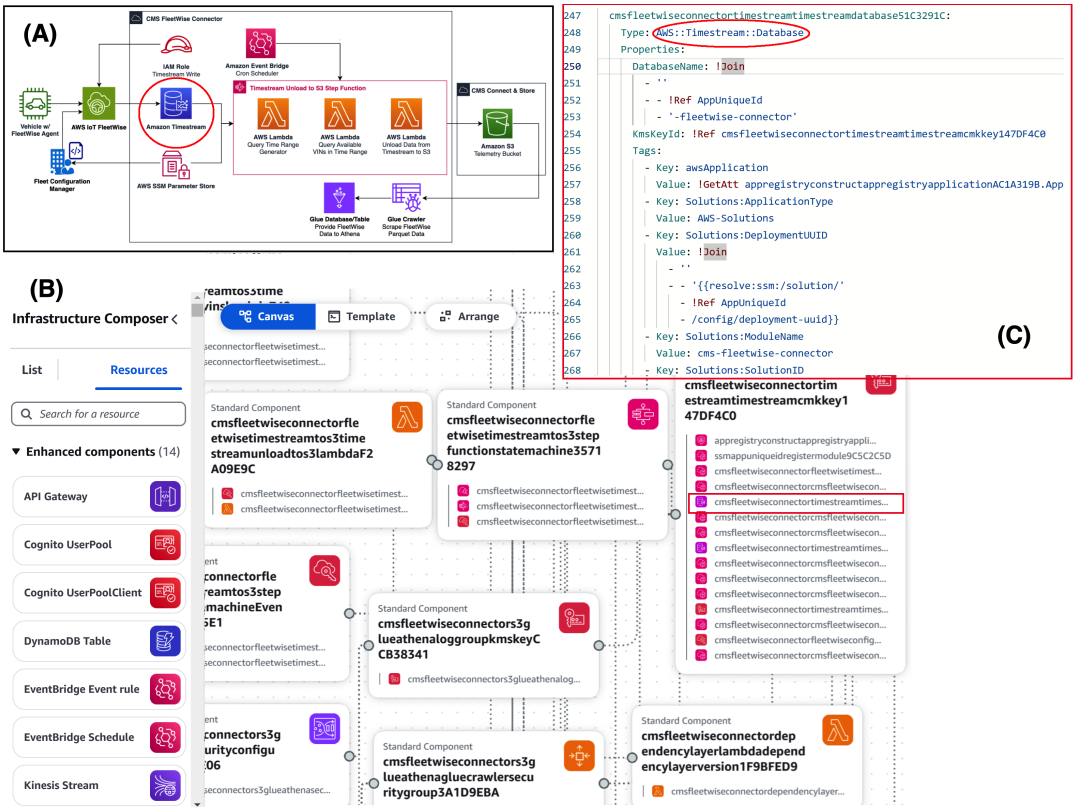

Fig. 2. Fleetwise Connector Module

regulated through AWS Identity and Access Management (IAM).

In the following, we describe how the characteristics identified in Table III are accomplished using the Amazon Web Services (AWS) ecosystem, using the Connected mobility solution on AWS[10] as a running example. The Connected Mobility Solution (CMS) on AWS satisfies customer needs for fleet management for increased efficiency and reduced vehicle downtime through preventive maintenance, location tracking, and improved safety and security. Users can employ various modular features, manage them from a centralized platform, and integrate custom modules as needed. CMS is composed of 14 modules. Figure 2 shows one of these modules, namely the FleetWise Connector module, which concerns the consuming of data captured by AWS IoT FleetWise campaigns. The figure is organized in three parts. Part A shows the logical architecture of the FleetWise Connector module. It is part of the documentation and it is not modeled using the AWS infrastructure composer and it is not synchronized in any way with the modeled physical architecture shown in part B of the figure. The physical architecture is composed of more than 30 components (called resources in the AWS CloudFormation YAML file), which implement the logical architecture. A mapping to the logical architecture is not straightforward as shown with the following example. The Amazon Timestream logical component (see the red circle in Part A) is within one component of the physical architecture (see the red rectangular box in Part B). Part C of the figure is an excerpt of the AWS CloudFormation YAML file describing the Timeseries logical component. Part C can be visualized and edit in the AWS infrastructure composer by clicking on the Template button. It is important to highlight that the physical architecture contains much more information than the logical architecture, e.g., it includes infrastructure aspects.

Code-Centric approach to define and manage architecture. As mentioned above, the *Diagram as code* aspect can be somehow achieved using the AWS Infrastructure Composer. The logical architecture is not explicitly visible and the AWS Infrastructure Composer mixes the infrastructure logic, e.g., the definition of network interfaces, with the architecture structure, e.g., services with their dependencies. *Alignment between architecture and code* is guaranteed by the AWS Infrastructure Composer, which directly maps to the CloudFormation template. However, this is limited to the physical architecture view. *Versioning* and *traceability* should be manually defined and managed.

Automation-driven evolution of architecture. The *automatic generation of architectural diagrams* aspect is obtained by construction since, during architecture modeling, the AWS Infrastructure Composer automatically creates the YAML file. Moreover, Amazon AWS provides services and architectural

[10]https://aws.amazon.com/solutions/implementations/connected-mobility-solution-on-aws/

and development best practices for *automatic analysis and quality assessment*. In particular, the AWS CloudGuard product enables users to specify and check rules on the YAML or JSON files. In contrast, the Well-Architected tool enables the review of workloads against current AWS best practices and guides on enhancing cloud architectures. This tool uses the AWS Well-Architected Framework based on six pillars: operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability.

Approachability and stakeholder engagement. The CloudFormation templates in YAML (or JSON) are a Domain-Specific Language (DSL) that provides a human-readable format and can therefore be considered an *accessible language*. The format is also *ubiquitous*, describing the whole architecture. AWS can offer support for *Software development life cycle models fostering stakeholder engagement* in two ways, with varying degrees of automation. In terms of life cycle *of* the architecture and infrastructure, the CloudFormation templates can be managed in a CI/CD pipeline using standard techniques for stakeholder engagement through requirements traceability in an Agile setting: this aspect is, therefore, *partially* supported. In terms of the life cycle of software *in* the architecture, AWS offers a service called CodePipeline that can be used to implement a CI/CD pipeline from source code versioning to deployment to production.

VI. Related Work

To the best of our knowledge, this work is the first scientific paper to thoroughly examine AaC within the context of software architecture, offering a comprehensive description and analysis of its characteristics. However, some prior works have used the terminology AaC over the years. For example, Krunic explored documentation as code within automotive software and systems engineering, employing the term AaC to refer to a model-based approach in which architecture is represented as a model from which source code and integration tests are generated [33]. A key outcome of this approach is complete interconnection coverage through generated integration tests. In his work on DevOps automation, Ganne employed the term AaC to describe a suite of automation tools integrated within Azure DevOps [23]. These tools facilitate the definition and deployment of infrastructure using code, enabling version control and consistency across environments. In a similar vein, Pandi et al. used the term AaC to describe IaC specifically applied to cloud architecture [37].

Although some prior works have used AaC as a synonym for IaC, this is a misconception, as the two concepts have distinct scopes and purposes [P7]. IaC focuses on automating the provisioning and management of infrastructure resources—such as virtual machines, networks, and storage—through code, eliminating the need for manual configuration [41]. In recent years, IaC has increasingly centered on cloud infrastructure, owing to its widespread adoption. AaC, however, goes further by encapsulating the logical design and behavior of entire software systems as code. It aims to codify architectural decisions, patterns, component relationships, and quality attribute requirements.

A concept closely related to AaC, researched in recent years, is agile architecture [8]. Agile architecture encompasses values, practices, and collaborations that support the continuous evolution of a system's architecture in alignment with Agile principles, emphasizing flexibility, adaptability, and a balance between intentional and emergent design. It also includes the concept of just-in-time architecture, which involves providing the right architectural knowledge at the right time [19]. While distinct, agile architecture and AaC share foundational principles such as flexibility, adaptability, and balanced design. Similarly, AaC aligns with the just-in-time architecture principle by abstracting and delivering architectural knowledge as needed. However, AaC goes further by offering a holistic approach to represent architectural decisions and configurations as code, leveraging version control, automation, and collaborative practices drawn from software development.

VII. Final Discussion and Future work

In this paper, we explored the concept of AaC and its benefits, particularly in enhancing consistency between architectural descriptions and implementation, as well as maintaining this consistency over time, even as the system evolves. However, several questions remain open and warrant further research and exploration in the future.

To start with, it is widely recognized in the software architecture community that architectures are described through various views and viewpoints, each addressing different stakeholder concerns [27]. This raises the question: *Which views and viewpoints can be effectively represented using an AaC approach?* A related question is: *To what extent can architectures be represented as code?* Given the characteristics of AaC, it seems more practical to represent the prescriptive aspects architecture as code [18], [26]. Prescriptive models, which are used to prescribe a subject (e.g., develop a system), are typically of greater interest to developers [26]. In contrast, descriptive aspects —often used for documentation— could likely be automatically generated as needed.

In addition, as stated in [51]: *Do architectural design decisions of an (envisioned) system satisfy the stakeholders' concerns and satisfy the business goals?* This question is multifaceted. Architectural design decisions are recognized as a core responsibility of software architects [31], yet are perceived as challenging to make [21]. AaC does not currently provide tools to support decision-making, and we found no mature instruments in existing AaC tools for documenting the rationale behind decisions. It may be easier in AaC to represent the effect of a decision rather than the decision itself, but documenting rationale remains crucial. Future work could draw on literature related to architectural decisions and the IEEE/ISO/IEC 42010 standard [27], which underscores the importance of documenting decision rationale and offers recommendations on which architectural decisions should be

recorded and what properties should be included in the decision log. Alignment with business goals is widely recognized as critical [11], [13]–[15] for achieving financial objectives, managing market position, ensuring product quality and reputation, and fulfilling responsibilities to society, countries, shareholders, and other stakeholders. However, alignment with business goals in the context of AaC remains an underexplored area.

Finally, as stated in [27], [51]: *Is the architecture description complete?* This question highlights the importance of covering all essential aspects in line with stakeholder concerns. We did not find ready-to-use solutions for this in the available tools. However, it should be feasible to automatically extract views from AaC artifacts and code, which could then support a semi-automated analysis of 'completeness' based on metrics [44] or scenario-based methods [29], [51] in relation to stakeholder concerns.

In conclusion, advancing our understanding of AaC requires focused study and empirical investigation, particularly in areas such as decision-making support, alignment with business goals, completeness of architectural descriptions, and the effective representation of stakeholder concerns. Addressing these gaps will be crucial to realizing the full potential of AaC in modern software architecture.

PRIMARY STUDIES

[P1] Finos, Architecture as Code https://devops.finos.org/docs/working-groups/aasc/, (2024).

[P2] AaC, Architecture-as-Code (AaC) https://arch-as-code.org, (2024).

[P3] Milan Milanović, Software Architecture As Code Tools https://medium.com/@techworldwithmilan/software-architecture-as-code-tools-331a11222da0, (2023).

[P4] Christian Bonzelet, FOLLOWING THE PATH OF ARCHITECTURE-AS-CODE https://cremich.cloud/following-the-path-of-architecture-as-code, (2024).

[P5] Architecture as Code. What's the Point? https://www.reddit.com/r/softwarearchitecture/comments/1g4s81c/architecture_as_code_whats_the_point/?rdt=64228, (2024).

[P6] Gregor Hohpe, Application Architecture as Code https://architectelevator.com/cloud/iac-architecture-as-code/, (2023).

[P7] Dhanraj Mekala, Driving innovation with architecture as code: A game-changer in software development https://www.enlume.com/blogs/driving-innovation-with-architecture-as-code/, (2024).

[P8] Matthew Bain, Architecture as Code https://www.accidental-architect.com/architecture-as-code, (2023).

[P9] Zaidul Alam, Revolutionizing Development with Solution Architecture as Code (SAoC) https://www.linkedin.com/pulse/revolutionizing-development-solution-architecture-code-zaidul-alam/, (2023).

[P10] Slava Vedernikov, My journey to C4InterFlow — Open-Source Architecture as Code Framework https://www.linkedin.com/pulse/my-journey-c4interflow-open-source-architecture-code-slava-vedernikov-9fbnf/, (2024).

[P11] Software architecture as code https://nljug.org/java-magazine/software-architecture-as-code/.

[P12] Sophia Parafina, Architecture as Code https://www.pulumi.com/blog/architecture-as-code-intro/, (2020).

[P13] C4InterFlow - Architecture as Code Samples https://c4interflow.github.io/architecture-as-code-samples-visualiser/.

[P14] Fasih Khatib, Software Architecture as Code https://fasihkhatib.com/2023/12/11/Software-Architecture-as-Code/.

[P15] Architectural programming https://zuehlke.github.io/machines-code-people/articles/aprg.html.

[P16] Ard, Treat architecture as code? https://craftsmen.nl/treat-architecture-as-code/, (-).

[P17] https://www.c4interflow.com.

[P18] Simon Brown, Software Architecture as Code https://dzone.com/articles/software-architecture-code, (2024).

[P19] mydeveloperplanet, Software Architecture as Code With Structurizr https://mydeveloperplanet.com/2024/03/20/software-architecture-as-code-with-structurizr/, (2024).

[P20] Architectures https://justinoconnor.codes/architectures/.

[P21] Christian Eder, Architecture Modelling & Diagramming As Code https://blog.devgenius.io/architecture-modelling-diagramming-as-code-3636b42fdd17, (2022).

[P22] Ruslan Korniichuk, Architecture as Code (AaC) with Python https://www.youtube.com/watch?v=-UgumFMUs5M.

[P23] Architecture as a Service - AaaS https://www.snapblocs.com/architecture-as-a-service.

[P24] riduidel, Architecture as code is not an easy task https://riduidel.wordpress.com/2021/09/24/6461/.

[P25] Brian McKenna, Architecture diagrams should be code https://brianmckenna.org/blog/architecture_code, (2023).

[P26] Automatically evaluating application architecture through architecture-as-code https://patents.google.com/patent/US11526775B2/en, (2021).

[P27] Sophia Parafina, Architecture as Code: Kubernetes https://www.pulumi.com/blog/architecture-as-code-kubernetes/, (2020).

[P28] Josh Kaplan, Agile Architecture in Practice https://jdkaplan.com/articles/agile-architecture-in-practice, (2023).

[P29] Klaus Lehner, Architecture as Code using C4 + PlantUML https://engineering.cloudflight.io/architecture-as-code-using-c4-plantuml, (2022).

[P30] Agile architecture https://scaledagileframework.com/agile-architecture/.

[P31] Open Security Summit, Lessons Learned From Trying to Create Architecture Diagrams As Code https://www.youtube.com/watch?v=DmwPPJcVYa4, (2022).

[P32] GOTO Conferences, Coevolution of Architecture & Code - Closing The Gap • Dave Thomas • YOW! 2022 https://www.youtube.com/watch?v=slGZMTFPElo, (2023).

[P33] Ruslan Korniichuk, Architecture as Code (AaC) with Python or way to become your own boss https://www.youtube.com/watch?v=CWuVaB13nRs, (2023).

[P34] Gaie et al., An architecture as a code framework to manage documentation of IT projects, Applied Computing and Informatics ahead-of-prin, (2021).

REFERENCES

[1] "Replication package," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.14135535

[2] P. Abrahamsson, M. A. Babar, and P. Kruchten, "Agility and architecture: Can they coexist?" IEEE Software, vol. 27, no. 2, pp. 16–22, 2010.

[3] N. B. Ali and K. Petersen, "Evaluating strategies for study selection in systematic literature studies," in Procs of ESEM, 2014.

[4] N. Ali, S. Baker, R. O'crowley, S. Herold, and J. Buckley, "Architecture consistency: State of the practice, challenges and requirements," Empirical Softw. Engg., vol. 23, no. 1, p. 224–258, Feb. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9515-3

[5] E.-A. Association, "About east-adl association," 2024. [Online]. Available: http://east-adl.info

[6] M. A. Babar, "An exploratory study of architectural practices and challenges in using agile software development approaches," in 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, 2009, pp. 81–90.

[7] V. R. Basili, G. Caldiera, and H. D. Rombach, "The Goal Question Metric Approach," in Encyclopedia of Software Engineering. Wiley, 1994, vol. 2, pp. 528–532.

[8] S. Bellomo, I. Gorton, and R. Kazman, "Toward agile architecture: Insights from 15 years of atam data," IEEE Software, vol. 32, no. 5, pp. 38–45, 2015.

[9] A. Bucaioni, A. Di Salle, L. Iovino, I. Malavolta, and P. Pelliccione, "Reference architectures modelling and compliance checking," Software and Systems Modeling, pp. 1–27, 2022.

[10] A. Bucaioni, A. Di Salle, L. Iovino, L. Mariani, and P. Pelliccione, "Continuous conformance of software architectures," in 2024 IEEE 21st International Conference on Software Architecture (ICSA). IEEE, 2024, pp. 112–122.

[11] A. Bucaioni, P. Pelliccione, and R. Wohlrab, "Aligning architecture with business goals in the automotive domain," in IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE (ICSA 2021), March 2021. [Online]. Available: http://www.es.mdu.se/publications/6151-

[12] K. Charmaz and L. L. Belgrave, "Grounded theory," The Blackwell encyclopedia of sociology, 2007.

[13] H.-M. Chen, R. Kazman, and A. Garg, "Bitam: an engineering-principled method for managing misalignments between business and it architectures," Sci. Comput. Program., vol. 57, no. 1, p. 5–26, jul 2005. [Online]. Available: https://doi.org/10.1016/j.scico.2004.10.002

[14] P. Clements and L. Bass, "Business goals as architectural knowledge," in Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge, ser. SHARK '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 9–12. [Online]. Available: https://doi.org/10.1145/1833335.1833337

[15] P. C. Clements and L. J. Bass, "Relating business goals to architecturally significant requirements for software systems," 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:110270852

[16] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in Procs of ESEM, 2011.

[17] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," Journal of Systems and Software, vol. 85, no. 1, pp. 132–151, 2012, dynamic Analysis and Testing of Embedded Software. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121211002044

[18] U. Eliasson, R. Heldal, P. Pelliccione, and J. Lantz, "Architecting in the automotive domain: Descriptive vs prescriptive architecture," in 2015 12th Working IEEE/IFIP Conference on Software Architecture, 2015, pp. 115–118.

[19] R. Farenhorst, R. Izaks, P. Lago, and H. van Vliet, "A just-in-time architectural knowledge sharing portal," in Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008). IEEE, 2008, pp. 125–134.

[20] M. Feilkas, D. Ratiu, and E. Jurgens, "The loss of architectural knowledge during system evolution: An industrial case study," in 2009 IEEE 17th International Conference on Program Comprehension, 2009, pp. 188–197.

[21] M. Fowler, "Design - who needs an architect?" IEEE Software, vol. 20, no. 5, pp. 11–13, 2003.

[22] R. Franzosi, Quantitative narrative analysis. Sage, 2010, no. 162.

[23] A. Ganne, "Applying azure to automate dev ops for small ml smart sensors," International Research Journal of Modernization in Engineering Technology, vol. 4, no. 12, 2022.

[24] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," Information and Software Technology, vol. 106, pp. 101–121, 2019.

[25] T. Greenhalgh and R. Peacock, "Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources," BMJ, vol. 331, no. 7524, pp. 1064–1065, 2005.

[26] R. Heldal, P. Pelliccione, U. Eliasson, J. Lantz, J. Derehag, and J. Whittle, "Descriptive vs prescriptive models in industry," in Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, ser. MODELS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 216–226. [Online]. Available: https://doi.org/10.1145/2976767.2976808

[27] I. ISO, "Ieee: 42010: 2011 systems and software engineering, architecture description," International Standard, 2011.

[28] I. Jacobson, "Use cases–yesterday, today, and tomorrow," Software & systems modeling, vol. 3, pp. 210–220, 2004.

[29] R. Kazman, M. Klein, and P. Clements, ATAM: Method for architecture evaluation. Carnegie Mellon University, Software Engineering Institute Pittsburgh, PA, 2000.

[30] B. Kitchenham and P. Brereton, "A systematic review of systematic review process research in software engineering," Information and software technology, 2013.

[31] P. Kruchten, "What do software architects really do?" Journal of Systems and Software, vol. 81, no. 12, pp. 2413–2416, 2008, best papers from the 2007 Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, April 10-13, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121208002057

[32] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," IEEE Softw., vol. 29, no. 6, p. 18–21, Nov. 2012. [Online]. Available: https://doi.org/10.1109/MS.2012.167

[33] M. V. Krunic, "Documentation as code in automotive system/software engineering," Elektronika ir Elektrotechnika, vol. 29, no. 4, pp. 61–75, 2023.

[34] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in 2018 IEEE International Conference on Software Architecture (ICSA), 2018, pp. 176–17 609.

[35] J. Madison, "Agile architecture interactions," IEEE software, vol. 27, no. 2, pp. 41–48, 2010.

[36] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," IEEE Transactions on Software Engineering, vol. 39, no. 6, pp. 869–891, 2013.

[37] S. S. Pandi, P. Kumar, and R. Suchindhar, "Integrating jenkins for efficient deployment and orchestration across multi-cloud environments," in 2023 International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems (ICSES). IEEE, 2023, pp. 1–6.

[38] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," SIGSOFT Softw. Eng. Notes, vol. 17, no. 4, p. 40–52, Oct. 1992. [Online]. Available: https://doi.org/10.1145/141874.141884

[39] A. F. Pinto, R. Terra, E. Guerra, and F. São Sabbas, "Introducing an architectural conformance process in continuous integration." J. Univers. Comput. Sci., vol. 23, no. 8, pp. 769–805, 2017.

[40] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," Information and Software Technology, vol. 108, pp. 65–77, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918302507

[41] ——, "A systematic mapping study of infrastructure as code research," Information and Software Technology, vol. 108, pp. 65–77, 2019.

[42] M. Rodgers, A. Sowden, M. Petticrew, L. Arai, H. Roberts, N. Britten, and J. Popay, "Testing methodological guidance on the conduct of narrative synthesis in systematic reviews: effectiveness of interventions to promote smoke alarm ownership and function," Evaluation, vol. 15, no. 1, pp. 49–73, 2009.

[43] M. Shaw and P. Clements, "The golden age of software architecture," IEEE Software, vol. 23, no. 2, pp. 31–39, 2006.

[44] S. Silva, A. Tuyishime, T. Santilli, P. Pelliccione, and L. Iovino, "Quality metrics in software architecture," in 2023 IEEE 20th International Conference on Software Architecture (ICSA), 2023, pp. 58–69.

[45] R. Taylor, N. Medvidovic, and E. Dashofy, Software Architecture: Foundations, Theory, and Practice. Wiley, 2009. [Online]. Available: https://books.google.it/books?id=j9pdGQAACAAJ

[46] N. K. Turhan and H. Oğuztüzün, "Metamodeling of reference software architecture and automatic code generation," in Proccedings of the 10th European Conference on Software Architecture Workshops, 2016, pp. 1–7.

[47] A. Vázquez-Ingelmo, A. García-Holgado, and F. J. García-Peñalvo, "C4 model in a software engineering subject to ease the comprehension of uml and the software," in 2020 IEEE Global Engineering Education Conference (EDUCON). IEEE, 2020, pp. 919–924.

[48] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in Procs of EASE. ACM, 2014, pp. 38:1–38:10.

[49] R. Wohlrab, U. Eliasson, P. Pelliccione, and R. Heldal, "Improving the consistency and usefulness of architecture descriptions: Guidelines for architects," in 2019 IEEE International Conference on Software Architecture (ICSA), 2019, pp. 151–160.

[50] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in Proceedings of the 38th international conference on software engineering, 2016, pp. 488–498.

[51] S. M. Ågren, E. Knauss, R. Heldal, P. Pelliccione, A. Alminger, M. Antonsson, T. Karlkvist, and A. Lindeborg, "Architecture evaluation in continuous development," Journal of Systems and Software, vol. 184, p. 111111, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221002089