Architecture as Code in Industry: A Multiple-Case Empirical Study

Valeria Pontillo *, Alessio Bucaioni †, Italo Carnevale ‡,
Amleto Di Salle *, Lorenzo Manzoni ‡, Patrizio Pelliccione *, Francesco Maria Sprotetto ‡
* Gran Sasso Science Institute (Italy), *name.surname*@gssi.it
† Mälardalen University (Sweden), *alessio.bucaioni@mdu.se*
‡ Imola Informatica (Italy), *icarnevale, lmanzoni, fsprotetto@imolainformatica.it*

***Abstract*—Architecture as Code has been promoted as a way to bring code-centricity, automation, and approachability to software architecture, yet empirical evidence on how it is realized in industry remains scarce. This paper reports a multiple-case study conducted with a European consulting company serving the financial and insurance sectors. We analyze three engagements of the company with different customers, which offer three complementary instantiations of Architecture as Code. Data were collected over six months through biweekly online workshops, follow-up emails, slide decks, and detailed researcher notes. Analysis proceeded iteratively, alternating between internal consolidation and researcher–industry validation, and concluded with a cross-case synthesis against the existing Architecture as Code framework. Findings show that the Architecture as Code paradigm is present in all cases, albeit at varying levels of maturity. The study provides empirical evidence of how Architecture as Code is practiced across various industrial settings, sharpening its operational meaning by indicating where its characteristics are sufficient and where they require refinement.**

***Index Terms*—Architecture-as-Code; Model-Driven Architecture; Case Studies; Empirical Software Engineering.**

I. Introduction

Software architecture shapes the structure, quality, and evolution of complex systems [12]. As organizations adopt continuous engineering, architecture must support rapid change while preserving integrity and traceability [2], [4], [8], [14], [18]. This shift increasingly blurs design, implementation, and operations, fostering executable and continuously synchronized architectural practices.

In such a context, the *"as Code"* paradigm has transformed Software Engineering (SE) practices by codifying traditionally manual activities into version-controlled, automatable artifacts [13], [17]. This shift, started by Infrastructure as Code (IaC) [10], has enabled reproducibility, consistency, and collaboration on an unprecedented scale. The recent *Architecture as Code* (AaC) concept extends this philosophy to architectural knowledge, capturing decisions, structures, and quality concerns as analyzable and evolvable code [4].

This work has been partially funded by (a) the MUR (Italy) Department of Excellence 2023 - 2027, (b) 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, (c) the Clean Energy Transition Partnership through the project "FLEXI: Human-centered AI and digital twin powered energy system integration for flexibility markets" (101069750), and (d) the Sweden's innovation agency Vinnova through the project iSecure (202301899).

Despite growing industrial interest, AaC lacks substantial empirical validation within software architecture. Recent academic work has begun to formalize the concept, identifying a set of core characteristics like code-centricity, automation, and approachability that define what it means to treat architecture *"as Code"* [4]. However, it remains unclear how these characteristics manifest in practice, how practitioners interpret and operationalize AaC principles in real projects, and whether the original conceptual framework [4] adequately captures the diversity of industrial implementations.

To address this gap, we present an empirical multiple-case study conducted in collaboration with a European consulting company specializing in software architecture management and digital transformation services for the financial and insurance sectors. The study examines three industrial cases that reflect distinct applications of the AaC paradigm as definition and governance of Architectural Decision Records (ADRs) [15], [16] linked to C4 models [3]. Through these cases, we investigate the implementation of AaC in industrial practice and aim to answer the following research questions:

- **RQ$_1$:** How are Architecture as Code characteristics and aspects manifested in industrial practice?
- **RQ$_2$:** To what extent do the previously defined Architecture as Code characteristics and aspects adequately describe the practices observed in real-world settings?
- **RQ$_3$:** What additional insights or refinements emerge from the empirical analysis of Architecture as Code implementations across multiple cases?

Our analysis provides both validation and refinement of the AaC concepts, bridging the gap between theoretical formulation and industrial application. The contributions of this work are threefold: (i) we provide an in-depth, multi-case empirical examination of AaC practices in an industrial context; (ii) we assess the adequacy of the established AaC characteristics and identify areas where they require extension or reinterpretation; and (iii) we distill lessons learned and observed benefits and challenges, offering guidance for organizations adopting or scaling AaC practices.

II. Architecture as Code: state of the art

The concept of AaC builds on the *Everything as Code* paradigm [17], which automates SE concerns by representing them as version-controlled, machine-processable artifacts.

AaC can be considered a specialization that focuses on codifying architectural knowledge and decisions.

Existing studies approach AaC from multiple perspectives such as documentation and project management support [5], model-driven approaches where architecture generates code and tests [7], and automation-centric views extending IaC within DevOps and cloud contexts [6], [9].

Recently, the definition of AaC has been studied through a systematic, multivocal literature review to identify its core characteristics and principles [4]. Building on that foundation, this study empirically examines how these characteristics manifest in real industrial settings.

AaC has been defined as *"an approach in which software architecture is continuously defined, managed, and evolved through a machine-readable, version-controlled code base. It embeds structural elements, rules, and constraints directly into codified artifacts, and leverages automation to ensure that architectural representations are generated and updated throughout the software lifecycle"* [4].

Accordingly, AaC is characterized by three dimensions [4]. First, its code-centric representation captures architectural knowledge in machine-readable artifacts (e.g., YAML/JSON) rather than static documents, enabling source-code workflows to govern architecture and reducing drift. Second, AaC relies on automation-driven synchronization. Because artifacts are executable or machine-processable, tool-chains and CI/CD pipelines can regenerate views and documentation, validate constraints, and enforce modeling or policy rules when the system changes. For example, tools can transform C4 specifications into typed models and metadata-enriched documentation, keeping architecture synchronized with code and configuration; without regeneration, AaC reverts to conventional documentation maintenance [4]. Third, AaC promotes approachability and collaboration through familiar notations, lightweight DSLs, and shared vocabularies. In Agile and DevOps contexts, this enables incremental architectural evolution, continuous visibility, and cross-team contribution.

In summary, AaC is gaining traction in industry and is widely discussed in practitioner forums and tooling ecosystems. However, systematic empirical studies of how AaC manifests in practice remain limited. This paper addresses this gap through an investigation of AaC concepts, practices, tools, and challenges within a medium-sized consultancy.

III. Research Process

We employ a multiple-case study design, following Yin [19] and Runeson and Höst [11], to examine how AaC principles manifest in industrial practice. The aim is to validate and, if necessary, refine or extend AaC characteristics through empirical evidence. A case study approach was selected to enable in-depth investigation of a contemporary phenomenon in its real-world, academia–industry collaborative context.

The study was conducted in partnership with a medium-sized consultancy in the financial and insurance sector. The company delivers software architecture management and strategic advisory services to organizations undergoing digital transformation, including the definition of architectural governance frameworks, documentation of design decisions, and implementation of automation mechanisms for architectural knowledge management. Three client engagements (cases) were selected through purposive sampling to represent different applications of architecture management automation and contrasting applications of AaC:

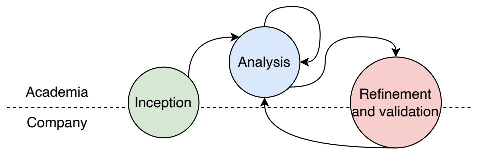

Fig. 1. Overview of the adopted research process.

- Case A (Architectural Practice and Governance): a large European private bank. The engagement focused on architecture governance and C4-based model management.
- Case B (Documentation and Views): a major European financial and insurance group. The project implemented automated generation of architectural documentation and views aligned with industry and regulatory standards.
- Case C (Architectural Knowledge): a top-tier European banking group. The engagement modernized enterprise-wide architectural governance and knowledge management, enabling decision traceability across subsidiaries through a shared repository.

Data were collected over six months through biweekly Microsoft Teams workshops involving at least three researchers and two enterprise architects (Figure 1), supplemented by email exchanges. Initial sessions were exploratory, focusing on organizational context, the three selected cases, and joint formulation of the RQs to align academic and industrial perspectives (green circle in Figure 1).

As the study progressed, meetings became iterative technical briefings in which emerging analytical insights were discussed (blue and red circles in Figure 1). Data collection and analysis were closely intertwined at two levels. First, within the research team, members independently analyzed the material, identified manifestations of AaC characteristics, and compared interpretations using shared matrices. Discrepancies were resolved through discussion in synthesis meetings, aiming at conceptual validation rather than statistical agreement.

Second, consolidated interpretations were presented to the company for validation and refinement (red circle in Figure 1). Practitioners confirmed, nuanced, or challenged the findings, leading to iterative adjustments of both the AaC framework and its mapping to observed practices.

For privacy and confidentiality reasons, no recordings or verbatim transcripts were made, while detailed manual notes were taken during each meeting. We conducted a dozen structured sessions, and all notes were collaboratively reviewed within the research team to ensure accuracy and completeness.

Threats to Validity

We discuss the threats to validity classification proposed by Runeson and Höst [11].

a) Construct validity: Data triangulation was ensured through multiple sources (meeting notes, slides, observations) and iterative validation sessions with practitioners. Research questions and scope were jointly defined with the company to ensure alignment with the industrial context.

b) Internal validity: Several actions were taken to strengthen interpretive rigor. Multiple researchers independently analyzed the same data and subsequently reached consensus through internal discussions, reducing individual bias. The iterative cycles between the academic and industrial teams further support interpretive validation, as preliminary interpretations were systematically reviewed, confirmed, or challenged by practitioners.

c) External validity: As the study focuses on three cases within the same industrial domain, the results are not statistically generalizable. However, the case studies were intentionally selected through purposive sampling to represent contrasting applications of AaC, thus allowing analytical generalization to other industrial contexts where architecture management and automation practices are comparable. The detailed descriptions of each case enable readers to assess the transferability of the findings to their own settings.

d) Reliability: A consistent research protocol was followed across the cases, including structured meeting schedules, shared analysis matrices, and collaborative note reviews. All data were anonymized, stored in an access-restricted repository, and cross-checked by multiple researchers. While no audio recordings were taken for confidentiality reasons, comprehensive manual notes were produced and verified by all researchers to maintain traceability.

IV. Case A (Architectural Practice and Governance)

Case A adopts a governance-driven, process-oriented approach to architectural decision-making in a large European private bank. Structured checklists and standardized solution design templates guide a staged workflow (early assessment, design validation, post-implementation review). Architectural knowledge is documented using the ARC42 standard,[1] including ADRs and C4 diagrams, to ensure consistent abstraction, traceability, and alignment with enterprise constraints.

a) Code-centric representation: Architecture is maintained as a structured, version-controlled system representation. Diagram-as-code, based on the C4 model and implemented with Structurizr, ensures a shared modeling language and reproducible views. Listing 1 illustrates a Structurizr DSL excerpt defining system boundaries, containers, components, and their runtime relationships as renderable, versioned code.

Alignment with implementation is ensured through manual reviews that update the "Application Landscape," a portfolio-level view of systems and integrations. Architectural artifacts are versioned manually using a conventional `vX.Y` scheme. Traceability relies on disciplined management of C4 definitions and versions within the modeling tool and repository.

[1]https://arc42.org/

Listing 1. Excerpt of the Structurizr DSL template

```
workspace {
  name "Solution Design - <project>"
  description "Solution design for <project>"
  model {
    person_user = person "<User type>"
      "E.g., financial advisors, customers"
    system = softwareSystem "<System>" "<Description>" {
      container_api = container "<Container>"
        "<Description>" "<Technology>" {
        component_auth = component "<Component>"
          "<Description>" "<Technology>"
      }
    }
    system2 = softwareSystem "<System2>" "<Description>"
    person_user -> system.container_api "User access"
    system -> system2 "Information exchange"
    system.container_api -> system.container_api
    "REST API call" "HTTPS"
  }
}
```

b) Automation-driven evolution of architecture: No automation is in place for the generation, validation, or synchronization of architectural artifacts. Diagrams and documentation are authored manually using the diagram-as-code DSL and then attached to solution documentation. Quality assessment and architectural analysis are conducted through manual reviews. There is no integration with development workflows or CI/CD pipelines: no automated regeneration of diagrams, constraint checking, or link validation runs on code/configuration changes. Consequently, consistency depends on process discipline and review cadence rather than on repeatable, tool-enforced checks.

c) Approachability and Stakeholder Engagement: Approachability is supported by accessible notations and a structured vocabulary (ADRs, C4 elements, JSON/DSL snippets), which establishes a shared language among architects and developers. To support a consistent and interpretable visual vocabulary, Case A also relies on a predefined set of Structurizr views and styling rules. Listing 2 illustrates an excerpt of the template used to standardize system, container, and component views, including layout directives and the color semantics applied to the different architectural elements. Stakeholder engagement occurs through continuous review of diagrams and documentation during design activities, keeping roles aligned on decisions and rationale. Each architectural change or relevant design decision is explicitly recorded, sustaining visibility and shared understanding as the system evolves.

Listing 2. Excerpt of the Structurizr view and style definitions

```
views {
  systemLandscape * { autoLayout tb 300 300 }
  container system * { autoLayout tb 300 300 }
  styles {
    element "Person" {shape Person
                      background #771e28 }
    element "System" {shape RoundedBox
                      background #9f2936}
    element "Container" {shape RoundedBox
                         background #d86b77}
    element "Component" {shape RoundedBox
```

```
                        background #e59ca6}
    }
}
```

To summarize, Case A demonstrates clear code-centric treatment of architectural knowledge (ADRs and diagram-as-code) and strong engagement through shared language and recurring reviews. Because of the absence of automation (no pipelines for regeneration, conformance checking, or link validation), synchronization and traceability rely on manual governance rather than tool-supported, repeatable workflows.

V. Case B (Documentation and views)

Case B automates component-level architectural views triggered by code changes, using industry- and regulation-aligned templates. Artifacts are treated as first-class configuration items: repositories are scaffolded from standard templates, metadata is pre-populated, and conventions are enforced via CI/CD. Documentation is maintained in machine-processable formats (e.g., Markdown, YAML, diagram-as-code). Each update triggers formatting, regeneration, semantic versioning, and publication to the internal portal, ensuring synchronization with development and regulatory compliance.

a) Code-centric representation: Adoption is explicit and operational: architecture is managed as code from the outset using diagram-as-code. Document skeletons are auto-generated in Git repositories, and all artifacts are authored in Markdown. Alignment between architecture and implementation is maintained for component-level views through synchronized workflows spanning Git, issue tracking, and CI/CD pipelines: each change triggers automated jobs that enforce structure, apply templates, and synchronize architectural metadata with development work items. Versioning is fully automated: upon merge request approval, the pipeline tags artifacts, updates semantic version numbers (for example, 1.0.0 -¿ 1.1.0), and handles major version bumps based on fields in the metadata.yaml file. Traceability is ensured through the Git history and cross-links among issues, commits, and merge requests, which record every modification to documents and diagrams. Listing 3 illustrates an auto-generated repository containing Markdown artifacts, media assets, and metadata descriptors structuring the architectural knowledge base.

Listing 3. Simplified structure of the Git repository

```
/
|-- media/
|    |-- images/
|    |-- graphs/
|
|-- metadata.yaml
|-- solution-design.md
```

b) Automation-driven evolution of architecture: Architectural documentation is automatically generated by the CI/CD pipeline: after merge approval, CI jobs render PDFs from Markdown/YAML sources, bundle versioned outputs, and CD jobs publish them to the documentation portal. Figure 2 outlines this workflow across Git, CI/CD, and publication stages, ensuring continuous synchronization between architecture and implementation. Automated quality analysis is not yet implemented, and reviews remain manual. However, structured metadata enables future completeness and conformance checks. Integration with development is strong: documentation workflows are embedded in the DevOps toolchain, and architectural artifacts evolve in lockstep with the code base.

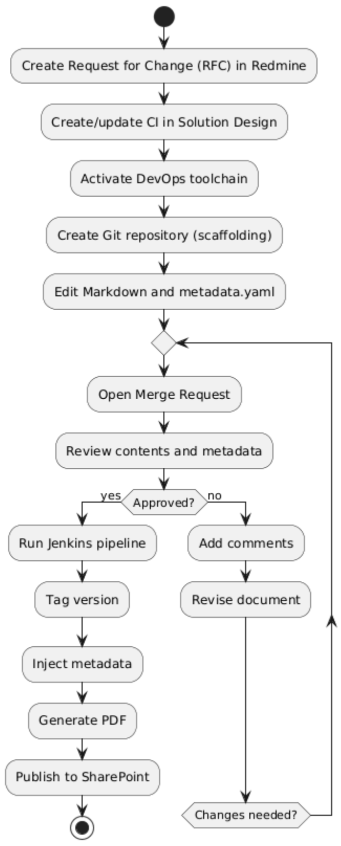
Create Request for Change (RFC) in Redmine
Create/update CI in Solution Design
Activate DevOps toolchain
Create Git repository (scaffolding)
Edit Markdown and metadata.yaml
Open Merge Request
Review contents and metadata
yes
Approved?
no
Run Jenkins pipeline
Add comments
Tag version
Revise document
Inject metadata
Generate PDF
Publish to SharePoint
Changes needed?

Fig. 2. Representation of the automated documentation pipeline in Case B, from the request to change creation to publication.

c) Approachability and stakeholder engagement: Approachability is supported through accessible notations, including a C4-based DSL rendered with PlantUML,[2] and Markdown for narrative content. A structured vocabulary derived from C4 ensures consistent terminology across teams. Listing 4 shows a `metadata.yaml` excerpt enabling naming conventions, classification, semantic versioning, and future automated validation. Stakeholder engagement is embedded via role-based workflows (author, reviewer, approver), merge/pull requests, automated notifications, and distributed reviews. Component-level documentation and diagrams reside alongside the code and are versioned with it, capturing emergent architectural changes in sync with implementation.

Listing 4. Excerpt of the metadata.yaml

```
title: "Architectural Guidelines"
document_type: "Technical specifications"
office: "Architecture and Data Management"
project: Name Project
version: 1.0.0
```

[2]https://plantuml.com/

```
classification: "internal use only"
```

In summary, Case B reflects a mature realization of AaC through standardized documentation, continuous artifact evolution, and structured review workflows. Although quality assessment is still manual, Markdown/YAML and metadata-driven processes establish a strong basis for further automation. Role-based reviews and formalized contribution workflows institutionalize stakeholder engagement and sustain shared understanding as the system evolves.

VI. Case C (Architectural knowledge)

Case C modernizes architectural governance and enterprise-wide knowledge management by consolidating artifacts in a shared repository to ensure traceability across subsidiaries and business lines. Enforceable standards define notation, vocabulary, mandatory metadata, and review roles, ensuring artifacts are consistent, comparable, and aligned for integration with centralized EA and GRC processes.

a) Code-centric representation: Case C adopts the C4 Model expressed as strict diagram-as-code with machine-readable notation. Diagrams are generated from design-time declarations or validated against runtime data, and are version-controlled under governed syntax. Alignment is maintained through solution processes that update architectural schemes and automated runtime checks detecting deviations between declared and actual states. Versioning policy requires architectural schemes to represent the latest "photograph" of the system, while as-is and to-be versions coexist to support planned evolution. Traceability is enabled by integrating EA and GRC tools, linking systems, architectures, and organizational processes. This approach conceptually mirrors a modeling and styling patterns similar to the one illustrated in Listing 1.

b) Automation-driven evolution of architecture: Automatic generation of architectural diagrams is provided by a custom toolchain that takes a structured spreadsheet as input and, using official C4 libraries, produces PlantUML diagrams. This mechanism enforces consistency, reproducibility, and standardization across representations. Although standardized fields (e.g., protocols, security patterns, and node types) enable potential automated analysis and quality assessment, qualitative validation of architectural information is still performed manually by solution architects. With respect to integration with development practices, no automated DevOps processes are currently in place. The management of diagrams and documentation relies on manual activities, while automation is limited to generating diagrams from structured inputs.

c) Approachability and stakeholder engagement: Approachability is ensured through a C4-based DSL rendered with PlantUML and a structured vocabulary with unified notation and color semantics, enabling consistent interpretation across teams. This aligns conceptually with the styling conventions in Listing 2. Stakeholder engagement is encouraged by involving development groups directly in the process: teams provide component inventories and integration details that feed the automatic diagram generator, fostering shared ownership of architectural information. The approach supports emergent architecture through continuous updates of architectural schemes and the coexistence of as-is and to-be views.

To summarize, Case C exhibits strong code-centric diagrams and governance-oriented practices with partial automation focused on diagram generation from standardized inputs. Integration with CI/CD and automated conformance checks is not yet present, so quality assurance and synchronization beyond diagram generation remain manual. Nevertheless, shared vocabulary, unified notation, and collaborative data provision enhance accessibility, traceability, and cross-subsidiary alignment of architectural knowledge.

VII. Answering the RQs

Each question targets a distinct perspective: in RQ$_1$, we analyse how AaC principles and characteristics are manifested in industrial practice across the three cases; for RQ$_2$, we evaluate the adequacy of the existing AaC characteristics; and in RQ$_3$, we derive refinements and extensions to the AaC framework grounded in real-world scenarios.

A. RQ$_1$: AaC Characteristics and Aspects in Practice.

To answer RQ$_1$, we refer to the three characteristics of AaC.

a) Code-Centric approach to define and manage architecture: All cases externalize architectural knowledge into machine-readable sources (for example, ADRs, C4-style definitions, Markdown/YAML) governed in repositories or enterprise platforms, and all employ a shared vocabulary that enables multi-role contribution.

b) Automation-driven evolution of architecture: Where automation is limited, organizations compensate by implementing disciplined review and governance to maintain alignment. Where automation is stronger, provenance and publication are managed through pipelines and templates rather than manual document work. Two implementation profiles emerge: (i) a pipeline-integrated profile (Case B), where CI/CD pipelines regenerate, version, and publish architectural artifacts in sync with code; (ii) an enterprise-portfolio–integrated profile (Case C), where registries and EA/GRC integration ensure standardization and cross-subsidiary alignment, albeit with weaker coupling to development workflows. Case A represents a governance-based baseline relying on manual review rather than tool-enforced checks.

Across the profiles, three operational enablers support workable AaC: a minimal metadata schema that makes artifacts parseable (e.g., metadata.yaml, standardized fields for decisions, interfaces, and policies), template that standardizes outputs for internal and regulatory audiences, and explicit ownership and review roles embedded in routine work. A trade-off emerges between speed of change and assurance: (i) pipeline-backed synchronization reduces drift but requires investment in schemas, templates, and trigger policies; (ii) enterprise integration increases portfolio visibility but can lag behind fast-moving code unless fed by development workflows. We also note two anti-patterns that hinder AaC outcomes: (i) "docs-as-code without synchronization", where artifacts are authored

TABLE I
AAC MAIN CHARACTERISTICS AND FOUNDATIONAL ASPECTS.

Characteristics	Aspects	Case Study A	Case Study B	Case Study C
(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, documentation, 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	●	●	●

as code but neither regenerated nor validated (readable yet prone to silent drift), and (ii) "generator-only", where views are produced from registries but lack automated conformance checks, shifting quality assurance back to manual review.

c) Approachability and stakeholder engagement: All cases adopt accessible notations and shared vocabularies (C4-based DSLs, naming conventions, visual standards) to support multi-role contribution. Engagement is embedded through recurring reviews (Case A), role-based merge workflows (Case B), or collaborative data provision for diagram generation (Case C). Architectural changes are explicitly recorded, supporting emergent architecture across contexts.

B. RQ$_2$: Adequacy of Existing AaC Characteristics and Aspects in Describing Practice.

This RQ assesses how well the previously defined AaC characteristics [4] account for the practices we observed. Table I summarizes the mapping: a half-filled circle (◐) indicates partial coverage, a filled circle (●) denotes full alignment, and an empty circle (○) indicates no relevant evidence.

For the code-centric representation, all organizations employ machine-readable architectural sources, but the extent of operationalization varies. Case A uses C4 definitions authored with Structurizr DSL and JSON descriptors. Alignment and versioning (vX.Y) are maintained by manual reviews rather than pipelines, and traceability relies on disciplined repository practices, hence ◐. Case C generates PlantUML/C4 diagrams from structured spreadsheets via an official C4 library and stores them in enterprise repositories. As-is and to-be views coexist, yet continuous linkage to the running system and development workflow is not enforced, also ◐.

Only Case B achieves a comprehensive realization (●): (i) Solution Designs, Guidelines, PlantUML diagrams, and Markdown are stored in Git, (ii) metadata.yaml drives generation and packaging, and (iii) artifacts are versioned automatically and co-evolve with code through merge workflows.

For automation-driven synchronization, we again see graded adoption. Case A has no automation beyond tool-assisted rendering (○): no CI/CD regeneration or constraint checking, while publication relies on Structurizr's native rendering rather than automated pipeline steps. Case C implements limited but meaningful automation (◐): a generator produces diagrams from standardized inputs (protocols, security patterns, node types), but validation, synchronization, and DevOps integration remain manual.

Case B provides an extensive pipeline (●) for component-level architectural views: (i) CI/CD renders PDFs from Markdown/YAML, applies organizational templates (including compliance-oriented formatting), increments semantic versions on merge approval, bundles outputs, and publishes them to an internal portal, and (ii) provenance is captured through issues, commits, and merge requests. This pattern suggests that the characteristic recognizes automation in a conceptual sense but does not distinguish its operational depth, such as generation, validation, or publication, or the integration points through which it is performed, which is important in practice.

For approachability and stakeholder engagement, all three cases align (●). Each adopts accessible notations and a shared vocabulary grounded in C4 (and Markdown/YAML where applicable), enabling multi-role contribution. Engagement mechanisms differ by context: (i) Case A relies on recurring multi-role reviews during design, (ii) Case B embeds roles (author, reviewer, approver) and notifications into merge workflows, keeping artifacts near the code, and (iii) Case C involves development teams in supplying component inventories and integration data that feed the generator. In all settings, changes are explicitly recorded and reflected in architectural views, supporting emergent architecture.

C. RQ$_3$: Additional Insights or Refinements.

Across the three cases, the original AaC characteristics remain conceptually sound, but their realization in industry depends on concrete operational mechanisms and on enterprise-level capabilities that the original formulation treats as implicit. Our structured sessions with the partner consultancy

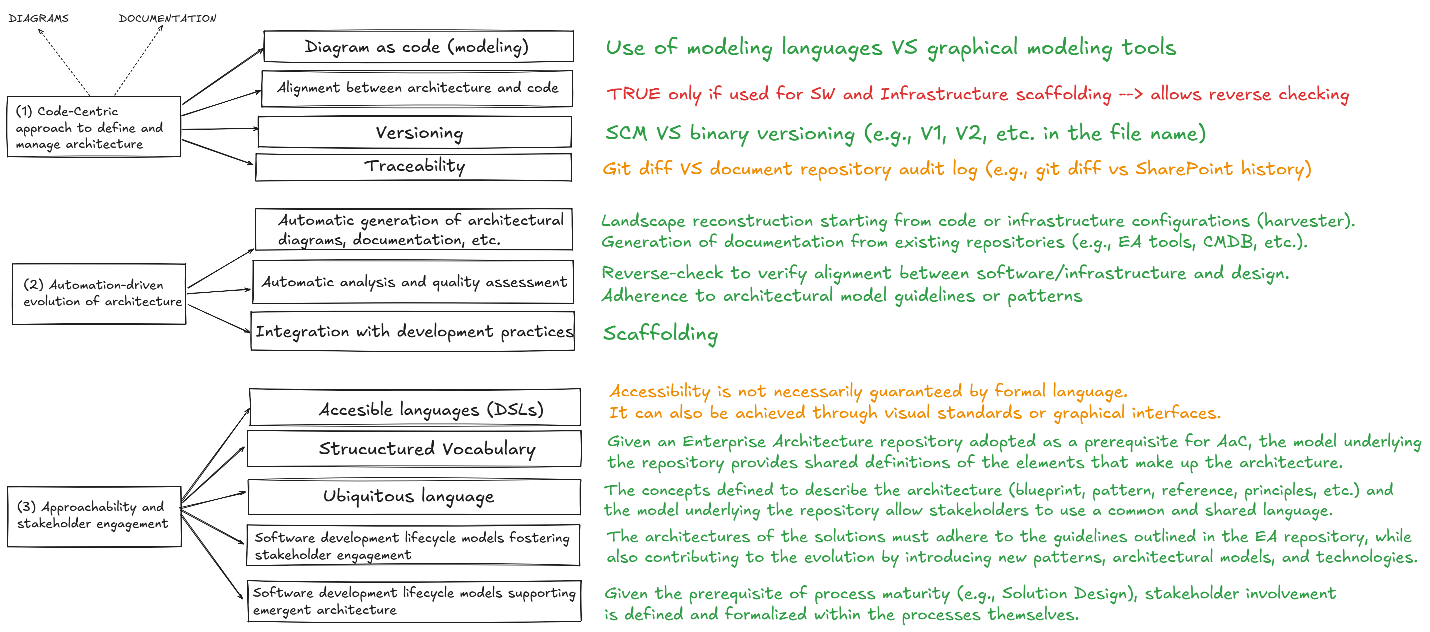
DIAGRAMS
DOCUMENTATION
(1) Code-Centric approach to define and manage architecture
Diagram as code (modeling)
Alignment between architecture and code
Versioning
Traceability
Use of modeling languages VS graphical modeling tools
TRUE only if used for SW and Infrastructure scaffolding --> allows reverse checking
SCM VS binary versioning (e.g., V1, V2, etc. in the file name)
Git diff VS document repository audit log (e.g., git diff vs SharePoint history)
(2) Automation-driven evolution of architecture
Automatic generation of architectural diagrams, documentation, etc.
Automatic analysis and quality assessment
Integration with development practices
Landscape reconstruction starting from code or infrastructure configurations (harvester).
Generation of documentation from existing repositories (e.g., EA tools, CMDB, etc.).
Reverse-check to verify alignment between software/infrastructure and design.
Adherence to architectural model guidelines or patterns
Scaffolding
(3) Approachability and stakeholder engagement
Accesible languages (DSLs)
Strucuctured Vocabulary
Ubiquitous language
Software development lifecycle models fostering stakeholder engagement
Software development lifecycle models supporting emergent architecture
Accessibility is not necessarily guaranteed by formal language.
It can also be achieved through visual standards or graphical interfaces.
Given an Enterprise Architecture repository adopted as a prerequisite for AaC, the model underlying the repository provides shared definitions of the elements that make up the architecture.
The concepts defined to describe the architecture (blueprint, pattern, reference, principles, etc.) and the model underlying the repository allow stakeholders to use a common and shared language.
The architectures of the solutions must adhere to the guidelines outlined in the EA repository, while also contributing to the evolution by introducing new patterns, architectural models, and technologies.
Given the prerequisite of process maturity (e.g., Solution Design), stakeholder involvement is defined and formalized within the processes themselves.

Fig. 3. Summary of the operational mechanisms required to instantiate each AaC aspect. Colors indicate practical feasibility: green for fully implementable with standard tooling; yellow for partially implementable or context-dependent; and red for mechanisms requiring strict enterprise-level prerequisites.

surfaced two complementary layers of refinement. To make AaC more actionable, it is important to have (i) fine-grained guidance on how each characteristic can be operationalised in practice (summarized in Figure 3), and (ii) higher-level architectural prerequisites that enable AaC to function within an enterprise ecosystem (depicted in Figure 4).

At the operational level, the analysis clarifies the conditions under which each AaC characteristic is effective. For the code-centric representation, effectiveness requires that (i) architectural sources live in a software configuration management system (e.g., EA repository) with semantic versioning and provenance, (ii) architectural views are regenerated from source (e.g., C4/PlantUML or Structurizr DSL) rather than maintained as static binaries, and (iii) bidirectional links connect model elements, and implementation artifacts. Alignment with implementation is achievable only when architecture and infrastructure scaffolding support harvesting and reverse checks (e.g., pipelines that rebuild views on code/configuration changes and verify model–implementation consistency). Versioning becomes meaningful when applied to the architectural source and its templates, rather than merely to the produced PDFs. Traceability relies on repository-based auditing and structured metadata (e.g., metadata.yaml, which includes standardized fields for protocols, security patterns, and node types) rather than on free-form documents. Finally, approachability requires more than a DSL: shared visual standards, naming conventions, and contribution workflows (roles, reviews, approvals) are needed to sustain multi-role participation.

Figure 3 makes visible operational mechanisms that the definition of AaC treated as implicit: scaffolding, EA metadata, standardized vocabularies, reverse checks, and repository-based auditing.

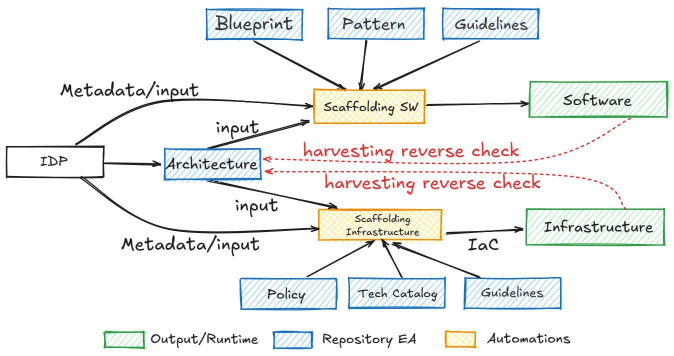

Fig. 4. Enterprise-level backbone enabling Architecture as Code.

At the enterprise level, AaC does not operate in isolation. Figure 4 summarizes an enabling backbone composed of an EA repository (as the semantic hub for vocabularies and relationships), internal developer platforms and scaffolding (to seed compliant services and integrations), blueprint and pattern libraries (to encode reusable solutions), policy and technology catalogs (for constraints and choices expressible as code), and automation modules that integrate with both software and infrastructure delivery. In this arrangement, architectural knowledge flows from the EA repository into scaffolding and pipelines, which generate and publish software and infrastructure. Then, runtime and configuration data flow back for reverse checks, conformance validation, and continuous coherence. This systemic perspective explains why (i) Case A relies on manual governance (absence of regeneration and reverse checks despite code-centric models),

(ii) Case B achieves continuous synchronization (tight CI/CD coupling and metadata-driven generation), and (iii) Case C achieves uniformity and cross-subsidiary alignment (registry-driven generation integrated with EA/GRC platforms).

In conclusion, the AaC characteristics describe the target capabilities, whereas their robust industrial realization depends on both characteristic-specific mechanisms (e.g., regeneration, reverse checks, repository auditing, shared visual and naming standards) and an enterprise backbone (EA repository, IDP/scaffolding, pattern catalogs, policy-as-code, and connectors to CI/CD and GRC). This refines the original conceptualization by making explicit the operational and infrastructural prerequisites that render AaC effective and sustainable at scale.

VIII. Discussion and Conclusion

This paper examined how AaC is realized in practice through a multiple-case study with a European consultancy in the financial and insurance sector. Across three engagements, we analyzed how established AaC characteristics manifest in practice. All characteristics were present, but with uneven maturity as shown in Table I. Two profiles emerged: a pipeline-integrated model (Case B), embedding architectural sources in CI/CD for automated generation, versioning, and publication; and an enterprise-portfolio–integrated model (Case C), standardizing data in registries linked to EA/GRC platforms for portfolio alignment but with weaker development coupling. Case A represents a code-centric baseline with largely manual synchronization and validation. The study contributes by empirically grounding AaC, evaluating the adequacy of its core characteristics, and refining them with operational and enterprise-level conditions for scalable adoption.

Baseline guidance is broadly applicable, while enterprise-level integrations (e.g., EA/GRC linkage, audit-ready outputs) should reflect organizational scale and regulatory context. Adoption should match maturity. Architecture teams should start with version-controlled, machine-readable sources, shared conventions, and clear ownership, then incrementally automate generation, validation, and publication. In regulated settings, integrate with EA and governance processes to ensure traceability without compromising readability.

Consultancies should provide context-aware playbooks aligned with two profiles: pipeline-integrated (delivery-focused) and enterprise-portfolio–integrated (alignment-focused). They should supply reusable templates, metadata schemas, scaffolding, and integration blueprints, ensuring pipelines can produce audit-ready evidence directly from architectural sources.

From a research perspective, evaluate AaC beyond code-centricity by assessing automation depth (generate–validate–publish) and integration locus (delivery pipeline vs. enterprise platforms). Report operational prerequisites (metadata schemas, reverse checks, repository auditing) and enterprise backbone elements (EA repositories, pattern libraries, scaffolding), as these explain outcome variance.

Future work will replicate the study across domains to assess transferability and longitudinal AaC maturation, including drift and governance sustainability. We will also define outcome-oriented indicators (e.g., update lead time, decision latency, conformance violations, time-to-compliance) to evaluate measurable impacts.

Data Availability

Due to non-disclosure agreements (NDA) with the collaborating company, only the materials that the company has authorized for public release are included in the replication package [1]. Data analyzed during the study cannot be shared publicly to ensure compliance with confidentiality obligations.

References

[1] "Online appendix," 2026. [Online]. Available: https://doi.org/10.6084/m9.figshare.30772790

[2] 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.

[3] S. Brown, The C4 Model: Visualising Software Architecture. O'Reilly Media, 2024, o'Reilly online learning.

[4] A. Bucaioni, A. Di Salle, L. Iovino, P. Pelliccione, and F. Raimondi, "Architecture as code," in 2025 IEEE 22nd International Conference on Software Architecture (ICSA). IEEE, 2025, pp. 187–198.

[5] C. Gaie, B. Florat, and S. Morvan, "An architecture as a code framework to manage documentation of it projects," Applied Computing and Informatics, vol. 21, no. 1/2, pp. 24–36, 2025.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," Empirical software engineering, vol. 14, no. 2, pp. 131–164, 2009.

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

[13] V. Stirbu, M. Raatikainen, J. Röntynen, V. Sokolov, T. Lehtonen, and T. Mikkonen, "Toward multiconcern software development with everything as code," IEEE Software, vol. 39, no. 4, pp. 27–33, 2022.

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

[15] J. Tyree and A. Akerman, "Architecture decisions: demystifying architecture," IEEE Software, vol. 22, no. 2, pp. 19–27, 2005.

[16] U. Van Heesch, P. Avgeriou, and R. Hilliard, "A documentation framework for architecture decisions," Journal of Systems and Software, vol. 85, no. 4, pp. 795–820, 2012.

[17] H. Wei, N. Madhavji, and J. Steinbacher, "Understanding everything as code: A taxonomy and conceptual model," in 2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2025, pp. 1–12.

[18] 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.

[19] R. K. Yin, Case study research and applications. Sage Thousand Oaks, CA, 2018, vol. 6.