**Journal of Engineering, Management and Information Technology**

ASPUR PUBLISHING

**Vol. 04, No. 04 (2026) 397-422, DOI:10.61552/JEMIT.2026.04.005 - http://jemit.aspur.rs**

# ARCHITECTURE DESCRIPTION LANGUAGES AND THE "4+1" VIEW MODEL: A STRUCTURED CRITICAL REVIEW OF SOFTWARE ARCHITECTURE REPRESENTATION APPROACHES

Kingsley Chinazaekpere Ndupu [1]
Serif Oyindamola Oyesiji
Chukwudera Obumneke Anunagba

Received 29.06.2026.
Revised 02.08.2026.
Accepted 04.09.2026.

Keywords:

*Software architecture, architecture description language, 4+1 view model, ISO/IEC/IEEE 42010, C4 model, microservices, continuous architecture, architecture decision records, architecture as code, architecture erosion.*

**Original research**

A B S T R A C T

*Software architecture description emerged as a formal subdiscipline of software engineering in the early-to-mid 1990s, producing two influential and complementary contributions: Kruchten's "4+1" view model, which organizes architectural description into logical, process, development, and physical views unified by representative scenarios, and Medvidovic and Taylor's classification and comparison framework for architecture description languages (ADLs), which evaluates formal modeling notations along component, connector, and configuration dimensions. Three decades later, mainstream software development has largely abandoned the formal ADL tooling that these frameworks were built to evaluate, while informally retaining much of the conceptual vocabulary they introduced. This review conducts a structured critical synthesis of the classical software architecture description literature and contemporary architecture documentation practice in order to establish what has transferred, what has not, and why. The review first reconstructs the 4+1 view model and the Medvidovic and Taylor framework in depth, together with the ten classical ADLs the framework was used to compare, namely Aesop, C2, Rapide, MetaH, UniCon, Weaves, Darwin, Wright, SADL, and ACME. It then surveys five contemporary developments: the ISO/IEC/IEEE 42010 standardization of architecture description; lightweight documentation approaches such as the C4 model, arc42, and architecture decision records; architecture description for microservice and cloud-native systems; model-driven and architecture-as-code approaches; and the emerging use of large language models for architecture documentation and recovery. The critical analysis argues that the separation-of-concerns logic underlying the 4+1 view model persists in modern notations, but that three assumptions embedded in classical ADLs break down under contemporary conditions: that a system possesses a single configuration rather than a population of request-scoped topologies; that reconfiguration is an act performed against a specification by an identifiable agent; and that the architecture description is the source from which an implementation is derived rather than an artifact recovered from one. Each of the ten classical ADLs is then revisited individually against contemporary practice, and the review proposes a five-factor contingency*

[1] Corresponding author: Kingsley Chinazaekpere Ndupu
Email: Kingsleynd34@gmail.com

*framework, comprising defect cost, architectural volatility, organizational scope, structural recoverability, and the relative half-lives of structure and rationale, that yields testable predictions about when heavyweight formal description remains warranted and which artifact should absorb documentation effort when it does not. Three summary tables consolidate the comparison, and the review closes with an agenda for empirical work on architecture representation adequacy.*

© 2026 Journal of Engineering, Management and Information Technolog

## 1. INTRODUCTION

Every non-trivial software system has an architecture, whether or not anyone has taken the trouble to write it down (Bass et al., 2021). The question that has occupied software architecture researchers since the early 1990s is not whether architecture exists, but how it should be described, communicated, and reasoned about by the people who must build, maintain, and evolve a system over its lifetime. Two lines of work from that founding period have proven unusually durable in the software engineering curriculum and in professional practice: Kruchten's (1995) "4+1" view model of software architecture, which decomposes architectural description into five interrelated views, and the broader research program on architecture description languages (ADLs), most comprehensively synthesized in Medvidovic and Taylor's (2000) classification and comparison framework. Both works responded to the same underlying problem, namely that a single diagram or a single notation cannot simultaneously satisfy the concerns of an end user, a maintenance programmer, a system integrator, and a project manager, and that architecture description therefore requires either multiple coordinated views or a formal language expressive enough to support multiple derived views and analyses.

Thirty years is a long time in software engineering. The industry that Kruchten (1995) and Medvidovic and Taylor (2000) were writing for and about was one of monolithic, single-vendor, single-deployment systems, typically built by a single organization under waterfall or early iterative processes, running on a comparatively small number of hardware nodes connected by local or wide area networks. The industry that exists today builds systems as constellations of independently deployable services, often owned by different teams, running on elastic cloud infrastructure whose topology changes many times per day, and released through continuous integration and continuous deployment (CI/CD) pipelines that assume architecture is never finished (Erder et al., 2021). Practitioner surveys conducted in the intervening decades report that formal ADLs of the kind Medvidovic and Taylor (2000) catalogued are rarely used in industrial practice, and that when architects do document their systems, they gravitate toward informal or semi-formal notations, box-and-line diagrams, and natural-language decision records rather than languages with executable or analyzable semantics (Malavolta et al., 2013; Ernst & Robillard, 2023).

This apparent divergence between the classical academic architecture description literature and contemporary industrial practice is the central tension the present review investigates. Rather than treating the divergence as evidence that the classical literature was simply wrong or has become irrelevant, the review asks a more structured question: which specific claims, distinctions, and design goals embedded in the 4+1 view model and the ADL classification framework transfer to modern practice in some recognizable form, which have been abandoned, and what mechanism explains the difference? Answering that question requires more than a restatement of either classical paper. It requires situating both works within the broader history of software architecture research, tracing the intermediate standardization efforts that formalized and partially superseded them, most notably the IEEE 1471-2000 recommended practice and its successor ISO/IEC/IEEE 42010, and comparing their design assumptions against the lightweight documentation practices, microservice architectures, and model-driven or architecture-as-code approaches that dominate contemporary software engineering discourse. It also requires taking seriously an intermediate retrospective literature that already sensed the shift underway; Kruchten et al. (2006), writing a decade after the original 4+1 paper, characterized software architecture as a field whose research agenda was migrating from notation and analysis toward decision capture, organizational alignment, and lifecycle integration, a migration this review traces to its contemporary conclusion.

The review makes four contributions. First, it reconstructs both classical frameworks in sufficient depth that their design assumptions, rather than merely their conclusions, can be examined. Second, it identifies three specific structural assumptions embedded in the classical ADL research program, concerning the cardinality of configurations, the agency behind reconfiguration, and the direction of derivation between description and implementation, and shows how each fails under contemporary cloud-native conditions in a way that is traceable to published empirical evidence rather than asserted. Third, it revisits each of the ten classical ADLs individually against contemporary practice, showing that several of them anticipated mechanisms that reappeared, unattributed, in modern infrastructure, and identifying the one classical capability, semantic connector-protocol compatibility checking, whose contemporary demand is clearest and whose contemporary supply is weakest. Fourth, it proposes a five-factor contingency framework

for selecting an architecture description approach, stated in a form that generates falsifiable predictions rather than restating that the choice depends on context.

The remainder of the review is organized as follows. Section 2 provides background on the definition of software architecture and of architecture description languages, and situates the 1990s ADL research program historically. Section 3 describes the review methodology, which is a structured narrative critical synthesis rather than a database-executed systematic review, and discloses its search strategy and scope limitations. Section 4 presents the classical foundations in depth, reconstructing Kruchten's (1995) 4+1 view model, Medvidovic and Taylor's (2000) classification framework, and the ten classical ADLs surveyed against that framework, consolidated in Table 1. Section 5 surveys contemporary developments across five threads, including a thread on artificial intelligence support for architecture documentation and recovery that has emerged largely since the most recent comprehensive treatments of this literature. Section 6 presents the review's central critical analysis, assessing continuity and discontinuity between classical and contemporary approaches, revisiting each classical ADL individually, and proposing the contingency framework. Section 7 discusses limitations, and Section 8 concludes with directions for future research.

The contribution of this review is not a new empirical dataset but a structured argument, grounded in a wide reading of both the classical and the contemporary literature, about the conditions under which the intellectual apparatus of 1990s architecture description research remains fit for purpose, and the conditions under which it has been superseded by lighter-weight alternatives better suited to the pace and organizational structure of contemporary software delivery.

## 2. BACKGROUND

### 2.1 Defining Software Architecture

Software architecture as a distinct object of study is usually traced to Perry and Wolf's (1992) foundational paper, which argued that architecture could be understood as a composition of processing elements, data elements, and connecting elements, together with a set of constraints on how those elements may be combined and a rationale explaining why a particular composition was chosen. Garlan and Shaw (1993) extended this framing, arguing that once a system reaches a certain scale, the problems of interest shift from selecting algorithms and data structures to selecting architectural structures, that is, the gross organization of a system as a collection of interacting components and the protocols governing their interaction. Shaw and Garlan (1996) later consolidated this perspective into an influential textbook framing software architecture as a discipline in its own right, distinct from both requirements engineering and detailed design, concerned with system-level structure, coordination, and the quality attributes that follow from that structure.

By the time Bass et al. (2021) published the most recent edition of Software Architecture in Practice, the field had settled on a widely cited definition holding that the software architecture of a system is the set of structures needed to reason about the system, comprising software elements, relations among them, and properties of both. This definition deliberately foregrounds the plurality of structures, echoing Perry and Wolf's (1992) original insight that no single structural view is sufficient, a point that motivates both the multi-view approach exemplified by Kruchten (1995) and the later standardization of architecture description through ISO/IEC/IEEE 42010. Kruchten's (1995) 4+1 model was not the only multi-view documentation methodology to emerge from this period. Clements et al. (2010), in the Software Engineering Institute's Documenting Software Architectures: Views and Beyond method, and Rozanski and Woods (2005), in their viewpoints-and-perspectives framework, each independently arrived at broadly similar conclusions, namely that architecture documentation should be organized as a small set of named, stakeholder-oriented views, each following a documented viewpoint convention, supplemented by cross-cutting information that does not belong to any single view. The convergence of three independently developed methodologies on a similar underlying structure is itself evidence that the multi-view principle reflects a genuine property of the documentation problem rather than an idiosyncrasy of any one author's approach.

A parallel strand of the same period reframed architecture less as a structure than as an accumulation of commitments. Bosch (2000), writing from the software product-line tradition, argued that an architecture is most usefully understood through the design decisions that produced it and the variation those decisions permit or foreclose across a family of related systems, since it is the decisions rather than the resulting boxes and lines that determine what a product line can subsequently absorb without redesign. That reframing, later sharpened by Jansen and Bosch (2005) and Tyree and Akerman (2005), supplies the intellectual lineage of the decision-centric documentation practices examined in Section 5.2, and it matters to the argument of this review because a decision-centric conception of architecture is far less dependent than a structure-centric one on the assumption that a system has a single, stable, enumerable configuration.

### 2.2 Architecture Description Languages

An architecture description language, in the sense developed by the research community in the 1990s, is a notation, typically with some degree of formal or semi-formal syntax and semantics, intended specifically for representing the components, connectors, and configurations of a software system's architecture, ideally in a way that supports automated analysis, consistency

checking, or code generation (Medvidovic & Taylor, 2000). This is a narrower notion than any notation used to draw architecture diagrams. General-purpose modeling languages such as the Unified Modeling Language (UML) were explicitly excluded from the core ADL research program on the grounds that UML's semantics were not precise enough, and its built-in modeling elements not architecture-specific enough, to support the kind of analysis that motivated the ADL community in the first place, although subsequent work explored the extent to which UML could be profiled or extended to serve architecture description purposes (Hilliard, 1999; Medvidovic et al., 2002).

The ADL research program that produced the languages surveyed by Medvidovic and Taylor (2000) grew out of a shared observation across several independent research groups in the late 1980s and early 1990s: architecture-level structure was being expressed informally, inconsistently, and without tool support, using ad hoc box-and-line diagrams whose informal semantics made them unsuitable for anything beyond communication between humans. Groups at Carnegie Mellon University, the University of California, Irvine, Stanford University, and SRI International, among others, each developed a language intended to formalize some subset of architectural concerns, typically emphasizing either structural composition, behavioral specification, or a specific application domain such as real-time embedded systems (Garlan et al., 1994; Luckham et al., 1995; Shaw et al., 1995; Moriconi & Riemenschneider, 1997). This proliferation of largely incompatible languages, each with different underlying assumptions about what an architecture is and how it should be represented, was itself a motivating problem for Medvidovic and Taylor (2000), whose classification framework was explicitly designed to permit comparison across a heterogeneous population of languages rather than to propose yet another one. Taylor et al. (2010) later consolidated this entire research tradition into a comprehensive textbook treatment, situating the individual ADLs and their comparison frameworks within a unified account of architectural modeling, analysis, and implementation that remains the most complete single synthesis of the classical ADL era available to students of the field.

### 2.3 The Historical Context of the 1990s ADL Era

It is important to the argument of this review to recognize that the 1990s ADL research program was conducted against a specific and now largely superseded set of industrial assumptions. Systems of the era were predominantly developed by a single organization, deployed to a bounded and enumerable set of hardware nodes, and released on a cadence measured in months or years rather than multiple times per day. Distribution, where it existed, generally meant client-server or a small number of tiers communicating over local or wide area networks, not the dynamically orchestrated, horizontally scaled service meshes of contemporary cloud-native computing (Erder et al., 2021; Newman, 2021). Under those assumptions it was reasonable to treat an architecture as a largely static artifact, produced early in a project's lifecycle, against which a system's implementation could later be checked for conformance, and equally reasonable to invest in formal languages whose value proposition depended on the architecture remaining stable long enough to justify the cost of formal specification and tool investment.

Kruchten's (1995) 4+1 view model and Medvidovic and Taylor's (2000) classification framework are best understood as the two most durable products of this research context: the former a pragmatic, industrially oriented framework for organizing architectural documentation around stakeholder concerns, developed at Rational Software and closely associated with what would become the Rational Unified Process, and the latter an academically oriented, more formally grounded framework for comparing the expressive power of research-grade architecture description notations. Understanding both works requires understanding this shared origin, because much of what has and has not transferred to contemporary practice can be traced directly to assumptions about system scale, deployment stability, and organizational structure that no longer hold universally.

## 3. METHODOLOGY

This review is best characterized as a structured narrative critical synthesis rather than a formal systematic literature review in the PRISMA sense. The distinction matters and is disclosed explicitly here rather than obscured, because a systematic review conducted by a single reviewer without independent dual screening, a registered protocol, or database-level reproducible search logs would not meet the methodological bar that term implies, and presenting it as one would misrepresent the rigor actually applied. What follows instead is a transparent account of how the literature underlying this review was identified, selected, and synthesized, so that a reader can judge the scope and limits of the evidence base for themselves.

### 3.1 Search Approach

Literature was identified through iterative search of IEEE Xplore, the ACM Digital Library, SpringerLink, ScienceDirect, Google Scholar, and general web search, using combinations of the following search term families: (a) foundational terms, including "software architecture," "architecture description language," "software architecture definition," and the names of specific classical works and their authors; (b) 4+1 view model terms, including "4+1 view model," "Kruchten architectural blueprints," and "logical process development physical view"; (c) ADL survey terms, including "architecture description language survey," "ADL systematic mapping study," and the names of individual classical ADLs (Aesop, C2, Rapide, MetaH,

UniCon, Weaves, Darwin, Wright, SADL, ACME); (d) standardization terms, including "ISO/IEC/IEEE 42010," "IEEE 1471," and "architecture description standard"; (e) contemporary practice terms, including "C4 model software architecture," "arc42 template," "architecture decision records," "ArchiMate," "microservices architecture documentation," "continuous architecture," "architecture as code," and "cloud-native architecture documentation"; (f) empirical terms, including "how practitioners document software architecture," "industrial survey architectural languages," "architecture erosion," "architectural drift," and "empirical study software architecture practice"; and (g) terms addressing the most recent thread surveyed here, including "large language models software architecture," "LLM architecture recovery," and "automated architecture documentation." No date restriction was applied to the classical literature; for the contemporary literature, search emphasized work published from 2020 onward, with particular attention to 2023 and later, in order to capture the microservice, architecture-as-code, and language-model threads that postdate the most recent comprehensive syntheses of this area.

### 3.2 Inclusion and Exclusion Criteria

Sources were included if they satisfied at least one of the following conditions: (a) they were an original, primary publication describing a classical ADL, the 4+1 view model, or a foundational definition of software architecture; (b) they were a peer-reviewed empirical study, survey, or systematic mapping study addressing architecture description language usage, adoption, or documentation practice, published in a recognized software engineering venue; (c) they were a primary or authoritative source describing a contemporary architecture description standard, notation, or practice, for example the official ISO/IEC/IEEE 42010 standard text, the C4 model's originating publications, or the Open Group's ArchiMate specification; or (d) they were a widely cited practitioner or textbook source that materially shaped the discourse this review examines, such as Michael Nygard's architecture decision record proposal or the Documenting Software Architectures text. Sources were excluded if their claims could not be independently verified against at least one authoritative bibliographic record, whether a publisher page, a digital library entry, or the venue's own archive, consistent with this review's commitment to citing only real, verifiable literature. Blog posts and vendor marketing material were used sparingly and only where they represent the original or most authoritative statement of a practice that has no equivalent peer-reviewed treatment, such as Fowler and Lewis's (2014) widely cited definitional article on microservices or Nygard's (2011) original architecture decision record proposal. A small number of very recent sources on language-model support for architecture work are available only as preprints; these are cited as preprints and are used to characterize the direction of an emerging research thread rather than to support load-bearing empirical claims.

### 3.3 Synthesis Approach

Given the heterogeneity of the source material, spanning peer-reviewed empirical studies, formal standards documents, and practitioner literature, a narrative thematic synthesis was used rather than a quantitative meta-analysis. Sources describing the classical ADL era were synthesized around the structure of the Medvidovic and Taylor (2000) framework itself, since that framework already provides a validated basis for comparing the languages it surveys. Sources describing contemporary practice were synthesized thematically around the five areas identified in Section 5: standardization, lightweight documentation, microservice and cloud-native architecture description, model-driven and architecture-as-code approaches, and language-model support for architecture documentation and recovery. The critical analysis in Section 6 was then constructed by systematically working through each major claim or design assumption of the classical works and asking, for each one, what the contemporary literature says about its continued validity.

Three summary tables consolidate the synthesis. Table 1 compares the ten classical ADLs; Table 2 maps the 4+1 views to their contemporary analogues; and Table 3 states the contingency framework proposed in Section 6.4. These tables are analytical constructions produced by the author from the sources cited within them, not extracted datasets, and they should be read as compact restatements of the argument developed in the surrounding prose rather than as independent evidence. Where a table cell characterizes a classical language, the characterization follows Medvidovic and Taylor's (2000) published assessment and the primary source for that language as cited in Section 4.3; where a cell identifies a contemporary analogue, the correspondence is an interpretive claim advanced by this review and defended in Section 6.

## 4. THE CLASSICAL FOUNDATIONS

### 4.1 Kruchten's 4+1 View Model

Kruchten (1995) proposed that the architecture of a software-intensive system is best described not by a single diagram but by five concurrent, partially overlapping views, four of which address the concerns of distinct stakeholder groups, with the fifth serving as an integrating and validating device rather than a standalone structural perspective. The model's central premise is that no individual stakeholder needs, or can absorb, all architectural information simultaneously, and that forcing all concerns into a single representation produces documentation that is either too abstract to be useful for detailed engineering work or too detailed to

communicate the system's overall organization to non-specialist stakeholders.

The logical view supports the functional requirements of the system, that is, what the system is supposed to do for its end users. It is expressed in terms of an object model of the design, or, in a less object-oriented system, in terms of the key abstractions and their relationships. Kruchten (1995) recommended organizing the logical view using object-oriented decomposition into classes grouped into categories or subsystems, with an eye toward identifying common design mechanisms and abstractions that recur across the functional requirements, so that common infrastructure can be recognized and shared rather than reimplemented per feature.

The process view addresses the non-functional requirements bound up with concurrency and distribution: system performance, availability, and integrity, and the way the system's runtime processes and threads are organized and communicate with one another. Kruchten (1995) described the process view as decomposing the system into a set of independently executing logical processes, called tasks, each a separate thread of control, that communicate and synchronize through defined interprocess mechanisms. Because distributed processes must be mapped onto physical processing nodes connected by local or wide area networks, the process view is also where architectural styles governing inter-process communication, such as pipes-and-filters or client-server arrangements, are made explicit, since the choice among such styles has direct consequences for the throughput, latency, and fault-tolerance properties the system can achieve.

The development view, sometimes called the implementation view, describes the static organization of the software in its development environment: how the system is decomposed into modules, libraries, and subsystems for the purposes of build management, code reuse, and team ownership, independent of runtime behavior. Kruchten (1995) recommended a layered organization in which the system is decomposed into four to six layers, each with a clearly defined responsibility, and each subsystem within a layer permitted to depend only on subsystems in the same layer or in layers strictly below it. This layering discipline was intended to make the development view directly useful for allocating work across teams, since a well-formed layered decomposition tends to correspond to a set of relatively independent units of work with well-defined interfaces between them, and to constrain the kinds of circular or tangled dependencies that make large systems difficult to build incrementally.

The physical view, sometimes called the deployment view, maps the software architecture, and specifically the elements identified in the process view, onto the physical hardware on which the system executes: processing nodes, network links, and the assignment of processes to nodes. Kruchten (1995) framed this view as the natural place to address non-functional requirements concerned with the physical realization of the system, including reliability, availability, fault tolerance, scalability, and performance, since these properties depend not only on the logical organization of the software but on how that organization is realized across physical or virtual infrastructure, including considerations such as redundancy, geographic distribution, and network topology.

The "+1" scenarios, sometimes called the use case view, are not a fifth independent structural perspective in the same sense as the other four. Rather, scenarios are instances of the system's use cases, described as sequences of interactions between objects and between processes, that are used both to discover architectural elements during design and, retrospectively, to illustrate and validate that the four other views work together coherently to deliver the system's required behavior. A scenario typically combines objects from the logical view with the processes and connectors identified in the process view, tracing a concrete path of execution through the architecture to demonstrate, for example, that a particular user-facing operation can be completed within its required latency budget while satisfying the system's concurrency and layering constraints. Kruchten (1995) treated this integrating role as essential precisely because the other four views are constructed somewhat independently, from different stakeholder perspectives, and therefore need an explicit mechanism to confirm that decisions made for functional reasons, decisions made for concurrency reasons, decisions made for build and team-organization reasons, and decisions made for deployment reasons are in fact mutually consistent.

### 4.2 Medvidovic and Taylor's ADL Classification Framework

Where Kruchten's (1995) contribution is organizational, aimed at practicing architects and structured around stakeholder concerns, Medvidovic and Taylor's (2000) contribution is comparative and formal, aimed at researchers and tool builders and structured around the expressive power of specification languages. Medvidovic and Taylor (2000) observed that by the late 1990s a substantial number of architecture description languages had been proposed by different research groups, each embodying different assumptions about what architectural elements are and how they should be represented, and that the lack of a common basis for comparison made it difficult for either researchers or practitioners to judge which language, if any, was appropriate for a given purpose. Their framework addresses this gap by defining what an ADL must be able to model, and then defining a set of dimensions along which any given language's treatment of that required modeling can be compared with any other's.

Medvidovic and Taylor (2000) held that architecture description must, at minimum, model three kinds of elements: components, the primary computational and data-storage elements of a system; connectors, the elements that model interactions among components and the rules that govern those interactions; and configurations, the connected graphs of components and

connectors that describe architectural structure. This tripartite ontology, inherited from the broader architecture research tradition established by Perry and Wolf (1992) and Garlan and Shaw (1993), is significant because it treats connectors as first-class entities in their own right, on equal ontological footing with components, rather than as mere incidental wiring. This was, at the time, a genuinely distinguishing feature of architecture-level description relative to lower-level design notations, in which connections between modules are typically implicit in call graphs or import statements rather than being independently named, typed, and reasoned about.

For components and connectors, Medvidovic and Taylor (2000) proposed six comparison dimensions. Interface captures whether and how a language allows the specification of the points of interaction a component or connector exposes. Types captures whether a language supports the definition of reusable component or connector types, as opposed to requiring each instance to be specified from scratch, and if so, what type system, if any, governs how those types relate to one another. Semantics captures whether and how a language specifies the meaning of a component's or connector's behavior beyond its interface signature, ranging from purely informal natural-language description to fully formal behavioral specification amenable to automated analysis. Constraints captures whether a language allows the specification of restrictions on how a component or connector may be used, composed, or configured, such as restrictions on the number or type of components a connector may attach. Evolution captures whether a language provides explicit support for architectures, or component and connector types within them, changing over time, whether through versioning mechanisms, dynamic reconfiguration facilities, or both. Non-functional properties captures whether a language allows the association of quality attributes, such as performance or reliability characteristics, with components and connectors, and whether those associations can be used to support architecture-level analysis of those attributes.

For configurations, Medvidovic and Taylor (2000) proposed nine comparison dimensions, several of which extend the component and connector dimensions to the level of overall architectural structure. Understandability captures whether the language produces specifications comprehensible to human readers, an important practical concern given that highly formal notations often trade readability for precision. Compositionality captures whether a configuration described in the language can itself be treated as a composite component, nested within a larger configuration, supporting hierarchical architectural description. Refinement and traceability captures whether the language supports a systematic, ideally verifiable, mapping from an abstract architectural specification down to a more detailed design or implementation, preserving the ability to trace implementation elements back to the architectural decisions that motivated them. Heterogeneity captures whether the language can accommodate components and connectors originally specified in other languages or formalisms, which matters because real systems are rarely built using a single uniform description technique throughout. Scalability captures whether the language's specification and analysis techniques remain tractable as the size and complexity of the described architecture grows. Evolvability captures, at the configuration level, whether the overall architecture can be modified without a disproportionate specification or re-verification cost. Dynamism captures whether the language supports architectures whose structure, that is, whose set of components, connectors, or their interconnections, may change at runtime, as opposed to architectures that are fixed once deployed. Constraints and non-functional properties, at the configuration level, mirror the corresponding component and connector dimensions but apply to properties and restrictions that only make sense at the level of the overall architectural structure, such as global topological constraints or system-wide quality attribute budgets.

Taken together, these fifteen dimensions constitute a demanding standard against which to measure any single language, and Medvidovic and Taylor's (2000) survey found, unsurprisingly, that no classical ADL satisfied all of them equally well. The value of the framework lies less in any single language's score against it than in making explicit the tradeoffs that language designers were, whether consciously or not, making when they chose to emphasize some dimensions over others. That observation is worth holding onto, because Section 6 argues that the dimensions along which the classical languages differed from one another turn out to be far less consequential for their subsequent fate than a set of assumptions all of them shared.

### 4.3 Survey of Classical ADLs Against the Framework

Medvidovic and Taylor (2000) applied their framework to a representative population of ADLs developed by different research groups during the preceding decade, and the resulting comparison illustrates how differently these languages approached the same underlying representation problem. Table 1 consolidates the survey's characterization of the ten languages discussed below, together with the contemporary analogue this review identifies for each, developed in detail in Section 6.3.

Aesop, developed at Carnegie Mellon University, was distinguished in the survey as the only language among those examined that provided at least some support across all six component and connector modeling categories, and it explicitly supported architectural evolution through its style-based approach to generating domain-specific design environments (Garlan et al., 1994; Medvidovic & Taylor, 2000). Aesop's underlying premise, that architectural styles such as pipe-and-filter or client-server could be formalized and then used to automatically generate a tailored design environment enforcing that style's rules, made it unusually well-rounded relative to languages that focused narrowly on one or two modeling concerns.

C2, developed at the University of California, Irvine, is organized around a strict architectural style in which components communicate exclusively through connectors arranged in a layered, message-passing topology, and the survey noted that C2 did not model non-functional properties directly but was notable for its practical, implementation-oriented tooling, supporting realization in C++, Java, and Ada (Medvidovic et al., 1996; Medvidovic & Taylor, 2000). This combination of strong structural discipline with weak treatment of quality attributes is characteristic of ADLs that grew out of a specific architectural style rather than out of an attempt to model architecture generically.

Rapide, developed at Stanford University, took an event-based approach to architectural specification, in which components communicate via posted events and architectures can be executed as discrete-event simulations to analyze their dynamic behavior. The survey found that Rapide, like C2, did not directly support the modeling of non-functional properties, notwithstanding its otherwise sophisticated behavioral and event-causality semantics (Luckham et al., 1995; Medvidovic & Taylor, 2000).

MetaH, developed for the domain-specific problem of real-time avionics guidance, navigation, and control software, was found not to support architectural evolution and to be tightly bound to the Ada language for component implementation, reflecting its origin as a domain-specific language for a safety-critical industry with different priorities than general-purpose architectural flexibility (Binns et al., 1996; Medvidovic & Taylor, 2000). MetaH's later evolution into the SAE Architecture Analysis and Design Language (AADL) is discussed in Section 5.4.

UniCon, developed at Carnegie Mellon University, offered a fixed catalog of predefined connector types, such as pipes, procedure calls, and data access, each with associated compilation and analysis support, but the survey found that UniCon, like MetaH, did not support architectural evolution once a system had been specified (Shaw et al., 1995; Medvidovic & Taylor, 2000). UniCon's strength lay instead in the depth of its treatment of a comparatively small, fixed set of connector types, each backed by working compiler and code-generation support, at the cost of the flexibility to define new connector types.

Weaves, developed at the University of California, Irvine and the Aerospace Corporation for large-scale tool-integration and data-processing applications, also lacked support for architectural evolution in the survey's assessment, reflecting its origin as a pragmatic tool-bus infrastructure for connecting existing analysis tools rather than as a general architecture specification notation intended to support long-lived, evolving systems (Gorlick & Razouk, 1991; Medvidovic & Taylor, 2000).

Several languages, including Darwin, MetaH, and Rapide, were found by the survey not to support explicit connector types as first-class, separately definable entities, instead modeling connections in-line as part of a component's specification, with the consequence that these languages offered no independent mechanism for connector evolution distinct from component evolution (Magee & Kramer, 1996; Medvidovic & Taylor, 2000). Wright, developed at Carnegie Mellon University, stood in contrast to this group by treating connectors as fully first-class, independently specifiable entities with their own formal behavioral protocols expressed in the CSP process algebra, supporting sophisticated static checking of whether a component's behavior is compatible with the protocol required by a connector to which it is attached, and thereby supporting flexible connector arrangement in a way that in-line connector languages could not (Allen & Garlan, 1997; Medvidovic & Taylor, 2000).

On the configuration-level dimension of refinement and traceability, the survey singled out SADL and Rapide as offering the strongest support among the languages examined for tracing an architectural specification through successive levels of refinement toward an implementation, with SADL in particular built around a formal theory of architectural refinement developed at SRI International specifically to support provably correct transformation from an abstract architecture to a more concrete one (Moriconi & Riemenschneider, 1997; Medvidovic & Taylor, 2000).

The comparative weaknesses the survey identified in individual languages' treatment of component and connector interfaces are not merely academic. Garlan et al. (1995) had already documented, through a series of concrete integration case studies, how subtle mismatches between the assumptions a component makes about its interaction context and the assumptions a connector or its neighboring components actually satisfy can cause substantial and unanticipated integration cost even when each individual element is well specified in isolation. This finding underscores why Medvidovic and Taylor (2000) treated interface and semantics as separate comparison dimensions rather than collapsing them into a single notion of compatibility: a component can expose a syntactically well-defined interface while still harboring behavioral or architectural assumptions, about control flow, data representation, or protocol ordering, that only a connector-level semantic specification of the kind Wright provided can surface before integration rather than after it.

Finally, the survey discussed ACME, developed jointly at Carnegie Mellon University and elsewhere, as occupying a distinct role from the other languages surveyed. Rather than competing with the other ADLs as an alternative primary specification language, ACME was explicitly designed as an architecture description interchange language, providing a common, relatively unopinionated core vocabulary of components, connectors, ports, roles, and configurations that could serve as a lowest common denominator for translating architectural descriptions between the more idiosyncratic, tool-specific notations used by languages such as Aesop, Wright, and Rapide (Garlan et al., 1997; Medvidovic & Taylor, 2000). This interoperability role made ACME widely referenced as a bridge across an otherwise fragmented and largely non-interoperable

population of restrictive, tool-specific ADLs, a role that, as discussed in Section 6, foreshadows the interoperability problems that would resurface, in a very different form, in the heterogeneous tooling ecosystems of contemporary cloud-native architecture.

Medvidovic and Taylor (2000) also discussed tool support as a cross-cutting concern distinct from the language-level comparison itself, observing that ADL tooling of the era tended to fall into several recognizable categories: tools supporting proactive specification, in which the architecture is specified before implementation and used to generate or constrain code, as opposed to reactive specification, in which an architectural description is derived after the fact from an existing implementation; tools supporting various forms of architectural analysis, such as consistency checking or performance simulation; tools supporting the generation of different views of an architecture tailored to different stakeholders, echoing the multi-view logic of Kruchten (1995) from a tooling perspective; tools supporting dynamism, that is, runtime reconfiguration; tools supporting refinement from architecture to detailed design; and tools supporting the generation of implementation skeletons or complete implementations directly from an architectural specification. The survey's finding that few if any languages had mature tool support across all of these categories anticipated a theme that would recur in later assessments of ADL adoption, discussed in Section 5.2 below, namely that the practical value of an ADL to a working engineering organization depends heavily on tool maturity and integration with the rest of the development toolchain, not on expressive power alone. The proactive-versus-reactive distinction introduced in that discussion is also, as Section 6.2 argues, the single classification in the entire framework whose contemporary balance has most decisively inverted.

**Table 1**. The Ten Classical Architecture Description Languages, Their Characterization in Medvidovic and Taylor's (2000) Framework, and the Contemporary Analogues Identified in This Review

| ADL (origin) | Primary modeling emphasis | Characterization in the classification framework | Principal limitation noted | Contemporary analogue (Section 6.3) |
|---|---|---|---|---|
| Aesop (Carnegie Mellon) | Architectural styles used to generate tailored design environments | Only surveyed language with at least some support across all six component and connector categories; explicit evolution support | Style formalism dependent on a dedicated design environment | Style enforcement without style formalism: opinionated service templates, scaffolding, and pipeline policy checks; no living descendant as a notation |
| C2 (UC Irvine) | Strict layered, message-passing style with implementation-oriented tooling | Strong structural discipline; realization supported in C++, Java, and Ada | Non-functional properties not modeled | Opinionated frameworks and service templates; the same silence on quality attributes recurs in C4 and arc42 |
| Rapide (Stanford) | Event-based specification with executable discrete-event simulation | Sophisticated behavioral and event-causality semantics; strong refinement and traceability support | Non-functional properties not modeled; connectors not first-class | Event-driven architecture and distributed tracing, with the direction inverted: histories are observed rather than simulated |
| MetaH (avionics GNC domain) | Domain-specific specification of real-time, safety-critical software | Tightly bound to Ada for component implementation | No support for architectural evolution; connectors not first-class | AADL, the one unambiguously living descendant, sustained by high defect cost and low volatility |
| UniCon (Carnegie Mellon) | Fixed catalog of predefined connector types with compilation and analysis support | Deep treatment of a small, fixed connector vocabulary backed by working tools | No support for architectural evolution; no user-defined connector types | Service mesh infrastructure, which makes the same fixed-catalog bet with a runtime rather than compile-time substrate |
| Weaves (UC Irvine and Aerospace Corp.) | Tool-bus composition for large-scale data-processing and tool integration | Pragmatic infrastructure for connecting existing analysis tools | No support for architectural evolution | Data and machine-learning pipeline orchestrators, in which graph evolution is likewise handled by editing and redeploying |
| Darwin (Imperial College London) | Declarative structural specification with dynamic structure | Designed with runtime structural change in mind | Connectors specified in-line, not as first-class entities, so connector evolution is not independent of component evolution | Service manifests and deployment descriptors, which inherited Darwin's in-line ontology and with it the absence of a place to attach connector properties |

| ADL (origin) | Primary modeling emphasis | Characterization in the classification framework | Principal limitation noted | Contemporary analogue (Section 6.3) |
|---|---|---|---|---|
| Wright (Carnegie Mellon) | First-class connectors with formal behavioral protocols in CSP | Static checking of component-to-connector protocol compatibility; flexible connector arrangement | High specification cost; dependent on a dedicated analysis environment | No adequate analogue: schema registries, interface definition languages, and contract testing check a syntactic subset only. Identified here as the clearest unmet demand |
| SADL (SRI International) | Formal theory of provably correct architectural refinement | Strongest refinement and traceability support among the surveyed languages | Narrow evidentiary base; refinement presupposes a human-directed abstraction ladder | Abandoned as refinement; survives degraded as non-formal traceability from decision record to commit to deployment manifest |
| ACME (Carnegie Mellon and collaborators) | Interchange vocabulary of components, connectors, ports, roles, and configurations | Deliberately unopinionated common core for translating between tool-specific ADLs | Value contingent on other tools choosing to emit it | Architecture-as-code proposals, which restate ACME's problem with the delivery pipeline as the point of integration, and inherit its failure mode |

Note. Column 3 and column 4 entries follow Medvidovic and Taylor's (2000) published assessment together with the primary source for each language cited in Section 4.3.

Column 5 entries are interpretive claims advanced by this review and defended in Section 6.3; they assert functional correspondence, not historical influence.

## 5. CONTEMPORARY DEVELOPMENTS

### 5.1 ISO/IEC/IEEE 42010 and the Standardization of Architecture Description

The most direct institutional descendant of the multi-view logic underlying both Kruchten (1995) and the broader ADL research tradition is the ISO/IEC/IEEE 42010 family of standards. Its lineage begins with IEEE Std 1471-2000, the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, which formalized a conceptual model in which an architectural description is organized around one or more stakeholders, each of whom has one or more concerns, and each concern is addressed by a viewpoint, which in turn governs the construction of one or more views (IEEE, 2000). This conceptual model is a direct generalization of the intuition underlying the 4+1 view model: rather than prescribing a fixed set of views, such as Kruchten's (1995) logical, process, development, and physical views, IEEE 1471 and its successors define a meta-model specifying what a view, a viewpoint, a stakeholder, and a concern are, and how they relate to one another, without committing to any particular fixed set of views as universally applicable.

IEEE 1471-2000 was subsequently adopted and revised jointly by ISO, IEC, and IEEE as ISO/IEC/IEEE 42010:2011, Systems and Software Engineering: Architecture Description, which extended the standard's scope from software-intensive systems to systems generally, reflecting the growing recognition that the architecture description problem is not unique to software (International Organization for Standardization et al., 2011). A further revision, ISO/IEC/IEEE 42010:2022, Software, Systems and Enterprise: Architecture Description, extended the standard's scope again to explicitly cover enterprise architecture, formally recognizing the convergence between software architecture description practice and the enterprise architecture tradition exemplified by frameworks such as ArchiMate (International Organization for Standardization et al., 2022; The Open Group, 2024). The standard's insistence on architecture viewpoints as reusable, named conventions for constructing a view, tied explicitly to the concerns of a class of stakeholders, gives principled standing to exactly the kind of stakeholder-oriented multi-view reasoning Kruchten (1995) had argued for on pragmatic grounds a decade and a half earlier, while remaining agnostic about which specific viewpoints, whether Kruchten's original four or some other set tailored to a different class of systems, an organization should actually adopt. Cross-domain practitioner frameworks that integrate information technology systems engineering with broader organizational supply-chain operations provide one contemporary example of how enterprise architecture has begun to absorb these cross-cutting concerns at the boundary between software and physical systems (Okonkwo et al., 2023).

The standard's conceptual generality, however, has not translated into corresponding uptake, and the gap is instructive. A meta-model that declines to prescribe viewpoints offloads onto each adopting organization precisely the work that Kruchten (1995) had already done for a particular class of systems, namely selecting a small set of viewpoints that are worth the cost of maintaining. Migliorini et al. (2024), mining roughly fifteen thousand architectural views from open-source repositories, found that most projects maintain a single view rather than a coordinated set, that views are rarely updated after creation, that they are typically authored by an individual

contributor near a project's beginning or end, and that the notation used is most often informal colored graphics without an accompanying legend. The multi-view principle, in other words, is widely endorsed and narrowly practiced, and the standard that generalized it supplies no mechanism to close that gap.

### 5.2 Lightweight and Agile Documentation

Notwithstanding the standardization effort, the practitioner literature and empirical studies of the last fifteen years converge on a consistent finding: heavyweight, formally specified architectural description of the kind the classical ADL research program aimed to support is rarely adopted in industrial practice, and where architecture is documented at all, it is documented using lighter-weight, less formal notations (Ernst & Robillard, 2023; Malavolta et al., 2013). Three developments illustrate this shift concretely.

The C4 model, proposed by Brown (2018), organizes architectural diagramming around four progressively zoomed-in levels of abstraction: system context, showing how the system in question fits among the people and other systems around it; containers, showing the high-level, independently deployable or runnable units, such as applications, data stores, and microservices, that make up the system; components, showing the major structural building blocks inside a single container and their interactions; and code, showing how a given component is implemented, typically via existing UML or equivalent detailed design notations rather than a C4-specific notation. Brown (2018) explicitly designed C4 as a reaction against both undisciplined, inconsistent ad hoc box-and-line diagramming and what he characterized as the excessive notational overhead of UML for architecture-level communication, aiming instead for a small, easily learned notation with just enough convention to be consistently interpretable across a diagramming toolchain, while deliberately avoiding the formal semantics and tool-enforced consistency-checking that characterized classical ADLs. Evidence on how the model fares once adopted at scale is beginning to appear: Jongeling et al. (2026), reporting on an embedded software organization that adopted C4 with diagrams authored in a text-based notation and stored alongside source code, found in a survey of forty-eight engineers conducted two years after adoption that the approach was regarded as successful, with the co-location of diagrams and code cited as a principal benefit, while the anticipated difficulties concerned maintenance and the communication of changes across team boundaries rather than the notation itself.

arc42 is a template rather than a notation, providing a standardized outline of twelve sections that an architecture documentation effort should typically address, including introduction and goals, constraints, context and scope, solution strategy, building block view, runtime view, deployment view, crosscutting concepts, architecture decisions, quality requirements, risks and technical debt, and a glossary (arc42, n.d.). The template is explicitly agnostic about which notation is used to populate its sections, and its building block, runtime, and deployment views map closely, though not identically, onto Kruchten's (1995) development, process, and physical views respectively, illustrating a recurring pattern in which the underlying separation-of-concerns logic of the 4+1 view model persists even where the classical terminology and notational apparatus have been replaced.

Architecture decision records (ADRs), first proposed by Nygard (2011), address a different aspect of architecture documentation than either C4 or arc42: not the structural description of the system as it currently stands, but the rationale for significant architectural decisions as they are made, recorded as short, individually version-controlled text documents typically capturing the context of a decision, the decision itself, and its consequences. Empirical work on ADR adoption in practice has found that development teams value ADRs primarily for onboarding new team members and for preventing the re-litigation of previously settled decisions, though sustaining consistent ADR practice over time, and integrating ADRs meaningfully with other architectural artifacts, remains an active challenge (Ahmeti et al., 2024). Repository-mining evidence tempers the enthusiasm: Buchgeher et al. (2023), examining ADR usage across GitHub, found that adoption remains low in absolute terms although the number of repositories using ADRs grows year over year, and that roughly half of the repositories that use ADRs at all contain only between one and five records, a distribution more consistent with exploratory trial than with sustained practice; where usage was sustained, it was characteristically a team activity carried out by two or more contributors over an extended period, and Nygard's original template dominated. The rise of ADRs as a distinct genre of architectural documentation nonetheless reflects a broader shift, discussed further in Section 6, from architecture as a structural artifact toward architecture as an accumulated, evolving record of decisions, a shift with clear affinities to the design-decision-centric view of software architecture articulated by Bosch (2000), Jansen and Bosch (2005), and Tyree and Akerman (2005).

### 5.3 Architecture Description for Microservices and Cloud-Native Systems

Microservice architecture, popularized in Fowler and Lewis's (2014) widely cited definitional article and elaborated at length in subsequent practitioner literature (Richardson, 2018; Newman, 2021), organizes a system as a suite of small, independently deployable services, each owned by a small team, each communicating with the others through lightweight mechanisms such as HTTP APIs or asynchronous messaging, and each free to choose its own internal implementation technology. This organizational pattern, together with the broader cloud-native computing movement with which it is closely

associated, characterized by containerization, dynamic orchestration, and service meshes that route and secure inter-service traffic, poses distinctive architecture description challenges that the classical ADL literature did not anticipate. It is worth noting that the dominant inter-service connector style in this environment, the Representational State Transfer (REST) style formalized in Fielding's (2000) dissertation on network-based software architectures, was itself originally presented using the same architectural-style vocabulary of components, connectors, and configurations subject to a coherent set of constraints that Perry and Wolf (1992) and Garlan and Shaw (1993) had established for the classical ADL tradition, even though REST itself is described informally in natural language and associated conventions rather than through any dedicated ADL. The contemporary applied literature on artificial-intelligence-augmented secure software engineering illustrates how such techniques are increasingly woven into the architectural-lifecycle concerns of continuous architecture in microservice and cloud-native settings, providing automated threat-detection and mitigation primitives that complement, rather than displace, the human architectural-review activities that the 4+1 model originally envisioned as part of the development view (Odejobi et al., 2025).

Empirical studies of microservice architecture practice document specific documentation challenges arising from this shift: the number of independently evolving services in a nontrivial microservice system can be large enough that no single up-to-date structural diagram remains accurate for long, service ownership is distributed across teams that may document their own services inconsistently, and the actual, physically realized topology of inter-service dependencies at any given moment may differ from any static diagram because of dynamic service discovery, feature-flagged routing, or canary deployment strategies (Söylemez et al., 2024). This dynamism sits in direct tension with the largely static configuration model assumed by classical ADLs, discussed further in Section 6. Reference architecture design efforts for microservice systems have accordingly emphasized domain-driven decomposition guided by bounded contexts and iterative validation against real deployment cases, rather than the kind of formal, tool-checked structural specification that characterized the classical ADL tradition (Söylemez et al., 2024). Ford et al. (2021) argue, in a widely read practitioner treatment of this shift, that the genuinely hard architectural problems in a distributed system are decisions about how to draw and manage service boundaries under evolving requirements, decisions that require ongoing trade-off analysis among competing quality attributes rather than a single, front-loaded structural specification, reinforcing the view that the classical ADL tradition's emphasis on getting an architectural specification right once, early, is a poor fit for systems whose service boundaries are expected to shift over the system's operational lifetime.

Quantitative characterizations of production microservice deployments give this argument a sharper edge than practitioner observation alone can supply. Luo et al. (2021), analyzing seven days of production traces from Alibaba clusters covering on the order of twenty thousand microservices and more than ten billion call traces, report that the runtime call graph is not a single stable topology but a large population of request-scoped graphs; that graphs produced by the same online service differ substantially from one another in topology; that graph size is heavy-tailed, with the largest graphs spanning hundreds to thousands of microservices; that mean call-graph depth is small, on the order of four, but with substantial variance; and that a small minority of services act as hot spots appearing in the overwhelming majority of graphs. Repository-level evidence points the same way over longer timescales: Assunção et al. (2023), analyzing changes across open-source microservice repositories, characterize microservice systems as subject to frequent and distributed change rather than to occasional coordinated redesign.

Two lines of work suggest that the response to this situation need not be the abandonment of structured models altogether. Ntentos et al. (2020) show that conformance of a microservice system to coupling-related patterns and practices can be assessed automatically using technology-independent metrics computed over models derived from a system's own artifacts, which is to say that architecture-level checking survives when the model is derived and the property checked is narrow. Genfer et al. (2025), in a controlled experiment with seventy participants drawn from two student populations and from industry, found that supplying security-annotated architecture decomposition diagrams alongside informal documentation significantly improved the correctness of participants' answers about a microservice system's security properties without increasing the time they spent, with the largest benefit accruing to less experienced participants. Structured architectural models, in other words, retain measurable value for comprehension even in the microservice setting where their maintenance is hardest; what has become untenable is not the model but the assumption that a human will keep it current by hand.

### 5.4 Model-Driven and Architecture-as-Code Approaches

A distinct contemporary thread continues the classical ADL project of treating architectural specification as a formal, potentially executable artifact, rather than abandoning formality in favor of lightweight documentation. The Architecture Analysis and Design Language (AADL), standardized by SAE International and directly descended from the MetaH language discussed in Section 4.3, provides a formally defined notation for specifying the software and hardware architecture of embedded, real-time, and safety-critical systems, supporting automated analysis of properties such as schedulability, reliability, and safety, and remains

in active use in domains such as avionics and automotive systems where the cost of formal specification is justified by regulatory and safety requirements (Feiler et al., 2006; SAE International, 2017; Feiler & Gluch, 2012). The Systems Modeling Language (SysML), standardized by the Object Management Group as a systems engineering extension and profile of UML, serves an analogous role in broader systems engineering contexts, supporting formal specification of structure, behavior, requirements, and parametric constraints across a system that may span hardware, software, and other engineering disciplines (Object Management Group, 2019).

A more recent development, loosely grouped under the label architecture as code, extends the broader infrastructure-as-code movement to architectural specification itself, representing architectural structure, constraints, and decisions in machine-readable, version-controlled files that can be validated, diffed, and used to drive automated tooling such as diagram generation or architectural conformance checking, integrated directly into the same CI/CD pipelines used for application code and infrastructure provisioning. Bucaioni et al. (2025) give the idea its most explicit recent formulation, characterizing architecture as code as an approach in which architecture is continuously defined, managed, and evolved through a machine-readable, version-controlled codebase, and identifying three constitutive characteristics: a code-centric representation that makes architecture versionable and traceable, automation-driven evolution in which diagrams, documentation, and quality assessments are regenerated by the delivery pipeline, and an emphasis on approachability so that stakeholders beyond the architect can participate. This approach shares the classical ADL research program's commitment to machine-checkable architectural specification but differs sharply in its design center of gravity: rather than aiming for the comprehensive, formally grounded expressiveness the Medvidovic and Taylor (2000) framework used as its yardstick, architecture-as-code tooling typically aims for a minimal, pragmatic specification sufficient to drive a specific automated check or diagram, integrated tightly with existing developer tooling and workflows rather than requiring a dedicated architecture description environment.

Early industrial evidence suggests the approach is being adopted unevenly and for organizational as much as technical reasons. Pontillo et al. (2026), in a multiple-case study of three engagements at large European financial and insurance organizations, observed the paradigm in use at differing levels of maturity and distinguished two operational profiles: a pipeline-integrated profile in which generation, versioning, and publication of architecture descriptions are automated within CI/CD, and an enterprise-portfolio-integrated profile in which a registry drives standardization and links architecture descriptions to enterprise architecture and governance processes. Their analysis makes explicit a set of prerequisites, including metadata schemas, reverse-checking capability, and repository auditing, that the original formulation had left implicit, and those prerequisites are worth noting because they are organizational rather than notational, which is precisely the pattern this review's contingency framework predicts.

### 5.5 Language-Model Support for Architecture Documentation and Recovery

The most recently emerging thread in this area concerns the application of large language models to architecture description tasks, and it deserves treatment here because it bears directly on the central tension of the review: if the binding constraint on formal architecture description has been authoring and maintenance cost rather than expressive power, then a technology that lowers authoring cost could in principle reopen options that the field closed for economic rather than intellectual reasons. Schmid et al. (2025), in a systematic literature review of work at this intersection, group existing approaches into four categories, namely reference architectures, classification and detection, extraction and generation, and assistant systems, and report that most published approaches rely on decoder-only models and comparatively basic prompting strategies, with conformance checking and cloud-native architecture among the areas least addressed. The state of the thread, in other words, is early, and the gaps the review identifies are precisely the capabilities the classical ADL program cared most about.

Two lines of evaluation are relevant to architecture description specifically. The first concerns generation of decision content. Dhar et al. (2024), in an exploratory study using a corpus of architecture decision records drawn from open-source repositories, found that current models can produce decision text that scores well on automated similarity measures while falling short of human quality on review, and that smaller fine-tuned models can approach the performance of much larger general-purpose ones, which matters for organizations unwilling to send architectural material to external services. Zhou et al. (2025), evaluating language-model generation of design rationale for architecture decisions against expert-provided ground truth across one hundred architecture problems sourced from developer forums and repository discussions, report a characteristic asymmetry between comparatively high recall and low precision, meaning that models surface a large share of the considerations experts identified while also generating many arguments experts did not; qualitative analysis of those additional arguments found the majority to be helpful and only a small minority potentially misleading. Read together, these results position language models as recall-oriented aids to human rationale capture rather than as substitutes for it, a positioning that fits the ADR genre well and the formal specification genre poorly.

The second line concerns recovery, that is, the derivation of architectural description from existing artifacts. Amalfitano et al. (2026) evaluated several contemporary models on recovering class diagrams, design and

architectural patterns, and architectural styles from source code, and found reasonable performance on structural and stylistic recognition alongside clear limitations on relational and pattern-level abstractions. Model-driven recovery approaches developed independently of language models report stronger results within a narrower scope: Alshuqayran et al. (2025) define and evaluate MiSAR, a model-driven approach for semi-automatically recovering architectural models of microservice systems from their implementation and deployment artifacts, evaluated across twelve systems and assessed against both the actual architecture and developer-documented architecture using precision, recall, and F-measure. Alongside these, tooling for measuring the divergence between design artifacts and code has begun to make drift observable rather than merely assumed: Romeo et al. (2024), analyzing hundreds of repositories that maintain UML artifacts in a text-based, version-controllable form, distinguish spatial drift, the share of implementation entities that appear in diagrams at all, from temporal drift, the lag between changes to code and corresponding changes to diagrams, and report that adoption of a version-control-friendly diagramming format did not by itself produce a corresponding increase in documentation effort or coverage. That last finding is a caution against the assumption, implicit in some architecture-as-code advocacy, that making a description machine-readable is sufficient to keep it current.

A related question, distinct from the use of language models as documentation tools, is how the architecture of systems that themselves embed machine-learned components should be described. Nazir et al. (2024), combining a systematic review of forty-one primary studies with interviews of twelve practitioners and researchers, report that absent documentation is among the challenges practitioners associate with reduced reusability and harder maintenance in machine-learning-enabled systems, and that adequate documentation is correspondingly identified as a best practice. The finding is not itself surprising, but it locates such systems awkwardly within the contingency framework proposed in Section 6.4: they frequently combine a high cost of undetected defect with high volatility and with structure that is only partially recoverable, since the behaviour of a learned component is not derivable from the artifacts that describe its wiring. That combination is precisely the one for which neither classical formal specification nor lightweight structural documentation is well suited, and it is a reasonable candidate for the next architecture description problem the field will have to solve rather than inherit.

## 6. CRITICAL ANALYSIS AND DISCUSSION

This section presents the review's central contribution: a systematic assessment of which elements of the classical 4+1 view model and ADL classification framework transfer to contemporary practice, which have broken down and by what mechanism, how each of the ten classical ADLs looks when revisited individually, and what a synthesized, contingency-based framework for architecture description might look like given all of the above.

### 6.1 What Transfers

The most robust finding of this review is that the separation-of-concerns logic underlying Kruchten's (1995) four views has proven considerably more durable than either the specific terminology or the specific notations Kruchten used to express it. The C4 model's context, container, component, and code levels, though motivated by a different organizing principle, namely progressive zoom rather than stakeholder concern, nonetheless recognizably echo the logical-development split in the 4+1 model: the container and component levels address structural organization for broadly the reasons Kruchten's development view did, while the context level addresses the system's relationship to its environment in a manner not unlike a simplified logical view (Brown, 2018). Similarly, arc42's building block, runtime, and deployment views correspond closely, though again not identically, to Kruchten's development, process, and physical views (arc42, n.d.). The persistence of this structure across notations with otherwise very different design philosophies suggests that Kruchten (1995) identified something close to an invariant property of how architectural concerns naturally decompose, rather than an artifact specific to the object-oriented, Rational Unified Process context in which the 4+1 model was originally proposed. The formal standardization of this insight in ISO/IEC/IEEE 42010's viewpoint-and-view meta-model further supports this reading: the standard did not enshrine Kruchten's specific four views as canonical, but it did enshrine the more general principle that architecture description should be organized around stakeholder concerns addressed through named, reusable viewpoints, which is precisely the intellectual move Kruchten had made a decade and a half earlier (International Organization for Standardization et al., 2011). Table 2 sets out the correspondences view by view, and, as importantly, the point at which each correspondence fails.

A second element that transfers, in modified form, is the recognition that architecture description benefits from an integrating, validating mechanism analogous to Kruchten's (1995) scenarios. Contemporary practice does not typically use the term scenario in Kruchten's sense, but functionally equivalent mechanisms recur throughout modern documentation practice: user journey narratives in C4-style documentation, the runtime view in arc42, which is explicitly intended to illustrate how the building blocks identified elsewhere in the documentation collaborate to realize specific important scenarios, and architecture decision records, which typically record the concrete scenario or requirement that motivated a decision as part of its context section (arc42, n.d.; Nygard, 2011).

**Table 2**. Kruchten's (1995) Views and Their Contemporary Analogues, With the Point at Which Each Correspondence Fails

| 4+1 view | Primary concern | Closest contemporary analogue | Nature of the correspondence | Where the correspondence breaks down |
|---|---|---|---|---|
| Logical | Functional requirements; key abstractions and their relationships | C4 system context and, partially, container level; arc42 context and scope | Both isolate what the system does and how it relates to its environment from how it is built or run | C4 organizes by progressive zoom rather than by stakeholder concern, so the logical and development perspectives are not separable at a single level |
| Process | Concurrency, distribution, performance, availability, and integrity | arc42 runtime view; distributed tracing and service-level objectives | Both describe runtime collaboration among independently executing elements | Kruchten's tasks are enumerable and statically assigned; contemporary runtime structure is a population of request-scoped call graphs (Luo et al., 2021) that no single diagram represents |
| Development | Module, library, and subsystem organization for build and team ownership | C4 container and component levels; arc42 building block view | Both describe static decomposition into units suitable for allocation to teams | Layering discipline presupposes one build and one dependency graph; in a multi-repository, multi-team estate no single team owns or can enforce the decomposition |
| Physical | Mapping of processes to nodes; reliability, scalability, and fault tolerance | arc42 deployment view; infrastructure-as-code templates and deployment manifests | Both bind software elements to the infrastructure that executes them | Nodes are no longer enumerable or stable, and the authoritative record of placement is generated by orchestration rather than authored by an architect |
| Scenarios ("+1") | Integration and validation of the other four views against required behavior | arc42 runtime scenarios; user journeys; production distributed traces | Both trace a concrete end-to-end path through the architecture to show that the views cohere | A scenario catalog is authored once and validated against a stable architecture; traces are emitted continuously by a changing system, so validation ceases to be an event and becomes monitoring |
| No classical counterpart | Rationale for architecturally significant decisions | Architecture decision records (Nygard, 2011) | Supplies the intent, alternatives, and consequences that no structural view records | Not a breakdown but a gap in the original model: the 4+1 model has no place for rationale, which is precisely the content that recovery from artifacts cannot reconstruct |

Note. Correspondences in columns 3 and 4 are interpretive claims advanced by this review; supporting sources are cited in Sections 5.2 and 6.1. The final row records an element of contemporary practice for which the 4+1 model offers no counterpart, discussed in Section 6.2.

What has changed is not the underlying need for a concrete, end-to-end illustration that ties structural views together, but the expectation that this illustration be captured once, formally, and treated as a stable artifact, as opposed to a living, frequently revisited narrative embedded throughout ongoing documentation. Distributed tracing deserves mention here as an unacknowledged descendant: a production trace is, formally, exactly what Kruchten's scenario was, namely a concrete path of execution through the architecture used to demonstrate that the views cohere, with the decisive difference that it is emitted by the running system rather than authored by an architect.

Third, at the level of ADL research specifically, the treatment of connectors as entities meriting independent attention, distinct from the components they connect, persists conceptually in contemporary distributed systems practice, even though the vocabulary of connectors as formally typed, independently specifiable objects has largely disappeared from mainstream documentation. Service mesh infrastructure, API gateways, and message brokers are, in effect, connectors in the Medvidovic and Taylor (2000) sense: they mediate interaction among components according to explicit rules, and modern practitioners reason carefully about their properties, such as retry policy, circuit-breaking behavior, and message delivery guarantees, in terms that

would be recognizable to a Wright or ACME designer, even though this reasoning today happens through configuration files, service mesh policies, and informal documentation rather than through a dedicated connector-typing formalism (Allen & Garlan, 1997; Garlan et al., 1997).

Fourth, and least remarked upon, the classical claim that structured architectural models improve reasoning has survived empirical test in the very setting where such models are hardest to maintain. Genfer et al. (2025) found that annotated decomposition models measurably improved the correctness of practitioners' reasoning about microservice security properties without costing additional time. The classical program's premise that an explicit architectural model has cognitive value was therefore not refuted by the industrial turn away from ADLs; what was refuted was the assumption that the value would be realized by having humans author and maintain those models by hand.

### 6.2 What Breaks Down

Set against this continuity, several core assumptions embedded in the classical ADL research program do not survive contact with contemporary distributed system practice. It is worth being precise about the mechanism, because the usual formulation, that modern systems are simply too dynamic for static models, understates the problem and misidentifies its cause. Three distinct assumptions fail, and they fail for different reasons.

The first is an assumption about cardinality. A classical ADL configuration is a single connected graph of components and connectors, and the analyses that gave the classical languages their value, whether style conformance in Aesop, protocol compatibility in Wright, or refinement in SADL, are analyses over that one graph. In a large cloud-native deployment there is no single graph to analyze. The Alibaba production traces reported by Luo et al. (2021) describe a population of request-scoped call graphs, differing substantially in topology even among invocations of the same online service, with a heavy-tailed size distribution in which the largest graphs span hundreds to thousands of services. A description adequate to such a system would need to characterize a distribution over graphs, including which topologies actually occur and with what frequency, and no classical ADL provides a construct for that. The nearest classical analogue is a style in the Aesop sense or a constraint set in the Wright and ACME sense, that is, a specification of which graphs are legal. But a style constrains rather than describes, and constraint gives no purchase on the question a contemporary architect actually asks, which is not whether a topology is permitted but which of the permitted topologies the system is in fact producing, and how often.

The second is an assumption about agency. In the classical model, reconfiguration is an act: an architect revises a specification, and the system is rebuilt or reconfigured to match, or, in the dynamic case exemplified by Darwin, a reconfiguration is invoked from within a space of changes the specification itself enumerates (Magee & Kramer, 1996). Contemporary topology change has no such author. It is the aggregate side effect of autoscaling controllers, deployment pipelines executing canary or blue-green strategies, feature-flag services altering routing, and service-discovery mechanisms resolving endpoints, each operating on timescales of seconds and each operated by a different team (Erder et al., 2021; Söylemez et al., 2024). The architectural specification in such a system is not violated; it is simply not in the causal loop. This is why the dynamism dimension of the Medvidovic and Taylor (2000) framework, which anticipated the theoretical need for runtime reconfiguration support, does not rescue the classical languages: their dynamism support presupposed that reconfiguration would be governed by a specification of the legal reconfigurations, an assumption that does not hold when topology changes are the emergent product of independent decisions made by dozens of independently operating teams and by control loops with no representation of architecture at all.

The third is an assumption about the direction of derivation. Medvidovic and Taylor (2000) distinguished proactive tool support, in which architecture is specified first and used to generate or constrain code, from reactive support, in which a description is derived from an existing implementation, and the classical program's centre of gravity was decisively proactive: SADL's refinement theory and Rapide's traceability support both presuppose that the abstract specification is the source and the implementation the derivative. In contemporary cloud-native systems the authoritative record of structure is the deployment configuration and the runtime telemetry, and the description is derived from them, whether by model-driven recovery (Alshuqayran et al., 2025), by metric computation over derived models for conformance assessment (Ntentos et al., 2020), or, increasingly, by language-model-assisted extraction (Amalfitano et al., 2026; Schmid et al., 2025). The field has moved to the reactive pole, and the consequential research question has changed with it. It is no longer how to specify an architecture precisely, but how to make a derived description trustworthy, and where to inject the prescriptive content, namely intent, rationale, and prohibition, that no recovery procedure can supply because it was never present in the artifacts being recovered from. Architecture decision records occupy exactly that gap, which is a better explanation of their rise than the usual observation that they are cheap to write.

Independent evidence for the scale of the resulting problem comes from empirical research on architectural change and erosion in deployed systems. Le et al. (2015), in a large-scale empirical study of architectural change in open-source software, found that meaningful structural change, comprising additions, removals, and rewiring of architecturally significant elements, occurs continuously and substantially throughout a system's operational life rather than being confined to occasional, discrete redesign episodes, directly undermining the premise that an architectural specification produced at one point in

time will remain an accurate description for long without active, continuous maintenance. Complementing this finding, de Silva and Balasubramaniam's (2012) survey of software architecture erosion documents an extensive body of work showing that, absent deliberate conformance-checking mechanisms, an implemented system's actual structure reliably drifts away from its documented or intended architecture over time. Li et al. (2022), in a systematic mapping study of the erosion literature, and Li et al. (2021), in an interview-based study of how practitioners perceive erosion, add two observations that sharpen the argument here: practitioners identify erosion through symptoms rather than through conformance tooling, which is largely absent from their environments, and they attribute erosion substantially to non-technical causes such as schedule pressure and team turnover rather than to technical drift alone. Romeo et al. (2024) make the same phenomenon measurable, distinguishing the share of implementation entities represented in design artifacts from the lag between code changes and diagram changes, and reporting that adopting a version-controllable diagram format did not by itself increase documentation coverage. The cumulative implication is that erosion, which the classical literature framed as a pathology of undisciplined projects, is under continuous delivery the default steady state of a hand-maintained description, and that the discipline required to prevent it is organizational rather than notational.

A fourth breakdown, closely related to the first, concerns the assumption implicit throughout the classical ADL literature and in Kruchten's (1995) physical view of a single deployable system with an enumerable, if potentially large, set of physical nodes. Microservice and cloud-native architectures instead comprise a shifting population of independently deployable, independently versioned services, frequently owned by different teams with different release cadences, such that there may be no moment at which the system's architecture is well-defined in the sense the classical literature assumed (Assunção et al., 2023; Newman, 2021; Wan et al., 2023). Empirical research on architecture documentation confirms that this organizational fragmentation, and not merely technical dynamism, is a primary driver of the documentation challenges practitioners report, since keeping a structural description current requires coordination across organizational boundaries that a single architecture team may not have the authority or visibility to enforce (Ernst & Robillard, 2023; Wan et al., 2023). Notably, the same organizational boundary appears as the principal anticipated obstacle even in an organization that adopted lightweight C4 documentation successfully on its own terms (Jongeling et al., 2026), which suggests the boundary problem is independent of notational weight rather than a consequence of formality.

Fifth, the scenario-driven validation logic at the heart of Kruchten's (1995) "+1" view assumes that architectural correctness can be meaningfully validated against a bounded, enumerable, and largely stable set of representative scenarios established relatively early in a project's lifecycle. Continuous architecture and DevOps practice instead treat architecture as an ongoing, incremental activity in which significant structural change may occur many times over a system's operational life, driven by evolving business requirements, incident postmortems, and incremental refactoring, such that the relevant validating scenarios themselves change faster than a formally maintained scenario catalog can track (Erder et al., 2021; Bass et al., 2015). The practical response to this mismatch, evident in the empirical ADR literature, has been to shift the unit of architectural validation from a scenario catalog validated against a relatively fixed architecture toward a continuously accumulating decision log validated, if at all, against the specific concern that motivated each individual decision (Ahmeti et al., 2024; Buchgeher et al., 2023; Jansen & Bosch, 2005).

Sixth, and most directly explaining the empirical non-adoption finding reported throughout Section 5.2, is a tooling and organizational-cost mismatch that Medvidovic and Taylor's (2000) own discussion of tool support anticipated without fully resolving. The classical ADLs' most sophisticated analytical capabilities, namely formal consistency checking, behavioral compatibility analysis, and simulation, depend on investment in a dedicated architecture description environment, separate from the general-purpose programming and deployment tools a development team already uses. Malavolta et al.'s (2013) large industrial survey found that practitioners consistently prioritized integration with existing tools and low learning overhead over the kind of deep formal analysis capability that had motivated the classical ADL research program, and more recent empirical work confirms that documentation effort is one of the recurring pain points architects report in day-to-day practice, suggesting that any additional formalism imposing further authoring overhead faces a steep adoption barrier regardless of its analytical benefits (Ernst & Robillard, 2023; Wan et al., 2023). Where formal, machine-checkable architectural specification has persisted in something like its classical form, as in AADL's continued use in avionics and other safety-critical domains, it has done so specifically in contexts where regulatory requirements or catastrophic failure costs justify the tooling investment that general commercial software development has largely judged not worthwhile (Feiler et al., 2006; SAE International, 2017).

One qualification is owed to the argument of this subsection, because the evidence does not support a clean narrative in which formal description failed and lightweight description succeeded. Migliorini et al.'s (2024) mining of open-source architectural views found that most projects maintain a single, rarely updated, informally notated view authored by one contributor, which is not the multi-view practice that C4 or arc42 prescribe either. Buchgeher et al.'s (2023) finding that half of ADR-using repositories contain fewer than six records points the same way. The industrial turn away from ADLs was therefore not a turn toward a lighter but comparably disciplined alternative; in a substantial share

of projects it was a turn toward very little documentation at all. Any argument that lightweight approaches are adequate must reckon with the possibility that their apparent adequacy is partly an artifact of the low bar against which they are measured.

### 6.3 Revisiting the Ten Classical ADLs

The preceding analysis operates at the level of assumptions shared across the classical program. It is equally instructive to revisit each of the ten languages surveyed in Section 4.3 individually, because several of them anticipated mechanisms that reappeared in contemporary infrastructure without attribution, and because the pattern of which languages have living descendants is itself evidence for the contingency framework proposed in Section 6.4. Table 1 summarizes the correspondences developed below.

Three of the languages have recognizable contemporary descendants in the form of infrastructure rather than notation. UniCon's design bet, a small fixed catalog of connector types each backed by real compilation and analysis support rather than an open-ended connector definition facility, was treated in the survey as a limitation, since UniCon supported neither new connector types nor architectural evolution (Medvidovic & Taylor, 2000; Shaw et al., 1995). Contemporary service mesh infrastructure makes precisely the same bet and is generally regarded as making it correctly: a mesh offers a small, fixed vocabulary of connector behaviors, including retry, timeout, circuit breaking, load-balancing policy, and mutual authentication, with genuine runtime implementations behind each, and users do not define new ones. What UniCon lacked was not the right catalog but a deployment substrate in which a catalog of connectors could be enforced at runtime rather than at compile time. Weaves, similarly, was a pragmatic tool-bus for composing data-processing and analysis tools, without evolution support (Gorlick & Razouk, 1991; Medvidovic & Taylor, 2000); its structural descendant is the contemporary data and machine-learning pipeline orchestrator, in which a directed graph of processing stages is declared and executed, and in which, as in Weaves, evolution of the graph is handled by editing and redeploying it rather than by any architectural evolution mechanism. Rapide's event-based semantics and executable discrete-event simulation of posted events (Luckham et al., 1995) reappear as event-driven architecture and distributed tracing, with one decisive inversion: Rapide simulated an event history from a model, whereas contemporary practice collects an event history from production and reasons backward. The classical capability was prediction; the contemporary capability is observation.

Two of the languages illuminate the contingency framework by the domains in which they did or did not survive. MetaH was assessed as lacking evolution support and as tightly bound to Ada (Binns et al., 1996; Medvidovic & Taylor, 2000), which would be disqualifying in a general commercial setting; it is nonetheless the one classical ADL with an unambiguously living descendant, AADL, precisely because avionics and comparable safety-critical domains combine a very high cost of undetected architectural defect with low architectural volatility and a single certifying organizational authority (Feiler et al., 2006; Feiler & Gluch, 2012; SAE International, 2017). MetaH's weakness on evolution was never binding in its domain. Aesop, by contrast, was the best-rounded language in the survey, offering some support across all six component and connector categories and explicit evolution support through style-based generation of design environments (Garlan et al., 1994; Medvidovic & Taylor, 2000), and it has no living descendant at all. The contrast is the clearest available evidence for a claim that runs throughout this review: coverage against the classification framework's dimensions was never the binding constraint on adoption. What survived of Aesop's idea is style enforcement without style formalism, in the shape of opinionated service templates, scaffolding tools, and pipeline policy checks that constrain what a team can build without ever representing the style as an analyzable object.

Two of the languages define what was lost. Wright's treatment of connectors as first-class entities with formal behavioral protocols in CSP, supporting static checking of whether a component's behavior is compatible with the protocol a connector requires (Allen & Garlan, 1997; Medvidovic & Taylor, 2000), addresses a problem whose contemporary form is more acute than its 1990s form and whose contemporary tooling is weaker. The problem is that of Garlan et al.'s (1995) architectural mismatch, which in a microservice estate recurs every time an independently deployed service changes the ordering, retry semantics, idempotency assumptions, or failure behavior expected across a boundary its consumers do not control. Contemporary practice addresses a syntactic subset of this through schema registries, interface definition languages, and consumer-driven contract testing, all of which check message shape and, at best, sampled interaction sequences. None checks protocol compatibility in Wright's semantic sense. This is, in the assessment of this review, the single classical ADL capability with the clearest unmet contemporary demand. SADL, by contrast, defines a genuine abandonment: its formal theory of provably correct refinement from abstract to concrete architecture (Moriconi & Riemenschneider, 1997) presupposed that the gap between an architectural specification and an implementation would be closed by successive human-directed refinement steps. That gap is now closed by frameworks, platforms, and code generation, so there is no refinement ladder left to verify. What survived is not refinement but traceability in a degraded and non-formal form, namely the chain from a decision record to a commit to a deployment manifest, which is auditable but not verifiable.

Three of the languages bear on the ontology and interoperability of contemporary description. Darwin modelled dynamic structure but did not treat connectors as first-class, separately definable entities, expressing

connections in-line within component specifications (Magee & Kramer, 1996; Medvidovic & Taylor, 2000). Contemporary practice inherited Darwin's ontology rather than Wright's: a service manifest or deployment descriptor declares a service's dependencies in-line, and there is no separately named, separately typed connector object to which connector-level properties could be attached. The consequence identified in the survey, that in-line connector languages provide no mechanism for connector evolution independent of component evolution, is precisely the situation of a modern estate in which changing a retry policy means editing the configuration of every consumer or, alternatively, changing a mesh policy that no service description mentions. C2 combined strict structural discipline with implementation-oriented tooling and no treatment of non-functional properties (Medvidovic et al., 1996; Medvidovic & Taylor, 2000); that combination is reproduced almost exactly by contemporary lightweight documentation, since neither C4 nor arc42 provides a place to attach a quality-attribute budget to a structural element, and the quality attributes accordingly migrate into service-level objectives maintained by an entirely separate toolchain with no link back to the architecture description. ACME's role as an interchange language across a fragmented population of tool-specific notations (Garlan et al., 1997; Medvidovic & Taylor, 2000) has the most direct contemporary echo of all, because the fragmentation has returned in a new form: a single organization may now express architecturally significant information in C4 or a diagram-as-code notation, in ArchiMate for enterprise governance, in infrastructure-as-code templates, in interface definition files, in service catalog metadata, and in decision records, with no common model across them. Architecture-as-code proposals are, viewed from this angle, ACME's problem restated with the delivery pipeline rather than the design environment as the point of integration (Bucaioni et al., 2025; Pontillo et al., 2026), and they face ACME's failure mode as well, since an interchange representation is only as useful as the set of tools willing to emit it.

### 6.4 A Contingency Framework

The pattern identified above, in which formal ADL-style architecture description persists in safety-critical and regulated domains while lightweight documentation dominates elsewhere, suggests that the appropriate response to the tension this review opened with is neither to declare the classical approach vindicated nor to declare it superseded, but to state the conditions under which each is justified in a form specific enough to be acted on and to be wrong. Drawing on the analysis above, this review proposes five factors, summarized in Table 3. The first three restate and sharpen distinctions already present in the literature; the fourth and fifth are advanced by this review and are the ones that make the framework prescriptive rather than merely descriptive.

The first factor is the cost of an undetected architectural defect. Where an architectural inconsistency can result in loss of life, regulatory non-compliance, or catastrophic financial loss, as in avionics, automotive, or certain financial-infrastructure systems, the fixed cost of formal ADL-style specification and its associated tool-supported analysis is justified by the marginal reduction in defect risk it provides, consistent with the continued use of AADL and comparable formalisms in these domains (Feiler et al., 2006). Where the cost of an architectural defect is instead measured in incremental engineering rework or a degraded user experience correctable in a subsequent release, the calculus favors lighter-weight documentation that can be produced and revised quickly enough to keep pace with delivery. The same defect-cost reasoning explains why relatively heavyweight architectural evaluation methods, such as Kazman et al.'s (2000) Architecture Tradeoff Analysis Method, which convenes stakeholders to systematically probe an architecture's quality attribute tradeoffs before significant implementation investment is committed, remain justified as an occasional, targeted exercise applied to a system's most consequential structural decisions even in organizations that otherwise document their architecture lightly. The method's cost is worth paying selectively, at the specific decision points where getting the tradeoff wrong would be expensive, without implying that the same rigor must be applied uniformly across an entire system's documentation.

The second factor is architectural volatility, that is, how frequently the structure being described actually changes. Systems with comparatively stable topology, such as a single-deployment enterprise application with an established release cadence, can sustain a heavier, less frequently updated architectural specification without that specification drifting far from reality between updates. Systems with high architectural volatility, including most microservice and cloud-native systems, instead require documentation approaches whose maintenance cost scales with the rate of change rather than requiring wholesale respecification each time the topology shifts (Ahmeti et al., 2024; Assunção et al., 2023; Söylemez et al., 2024;).

The third factor is organizational scope, that is, how many independent teams and ownership boundaries the architecture spans. A single team's architecture, however complex internally, can be documented and kept current by that team using whatever notation it finds most useful, including a fully formal one if the team judges the investment worthwhile. An architecture that spans many independently operating teams, as is typical in large microservice deployments, benefits disproportionately from lightweight, low-barrier-to-entry documentation conventions, such as a shared C4 or arc42 template applied consistently across teams, precisely because the coordination cost of imposing and sustaining a more demanding formalism across organizational boundaries tends to exceed its analytical benefit (Malavolta et al., 2013; Wan et al., 2023). The qualification established in Section 6.2 applies here with force: organizational scope

is the factor whose effect appears to be independent of notational weight, since even successfully adopted lightweight documentation encounters it (Jongeling et al., 2026), and it is therefore the factor least amenable to a purely notational remedy.

The fourth factor, and the first of the two this review adds, is structural recoverability: whether the system's actual structure can be derived automatically from machine-readable artifacts the organization already maintains, such as deployment manifests, infrastructure-as-code templates, interface definitions, service catalog metadata, and distributed traces. Where recoverability is high, hand-authored structural description is dominated as a strategy, because a generated description is cheaper, is current by construction, and describes the system that exists rather than the system that was intended; the appropriate investment is in recovery and conformance tooling of the kind demonstrated by Alshuqayran et al. (2025) and Ntentos et al. (2020), together with a thin prescriptive layer capturing the constraints and intent that recovery cannot supply. Where recoverability is low, as in embedded systems whose structure is expressed in build configuration and hardware allocation rather than in interrogable runtime artifacts, or in legacy systems whose structure is implicit in code, hand-authored structural description retains its classical value because there is no cheaper substitute. This factor is the practically decisive one for most contemporary commercial systems, and it is invisible in the classical framing because in the 1990s recoverability was uniformly low, which is why Medvidovic and Taylor (2000) could treat reactive tool support as a secondary category rather than as the default case.

The fifth factor is the relative half-life of structure and rationale. Where the structure of a system changes faster than the reasoning that produced it, as is typical when a stable set of business and regulatory commitments is realized through frequently re-partitioned services, documentation effort should be concentrated on decisions rather than on structure, because a decision record retains its value across many structural revisions while a structural diagram does not. Where the reasoning is volatile but the structure is stable, as in long-lived embedded controllers subject to changing requirements but fixed topology, the reverse holds and structural specification is the durable artifact. This factor answers a question the first three leave open. Factors one through three indicate how much formality a context can justify; the fifth indicates which artifact should absorb the effort, and it explains why architecture decision records have spread fastest in exactly those environments where structural documentation has decayed most, an association usually attributed to their low authoring cost but better explained by the fact that in those environments rationale simply outlives structure.

**Table 3**. A Five-Factor Contingency Framework for Selecting an Architecture Description Approach

| Factor | Diagnostic question | A high value indicates | Implication for the description approach | Predicted failure if mismatched |
|---|---|---|---|---|
| Cost of an undetected architectural defect | What is the worst consequence of an architectural inconsistency reaching production? | Loss of life, regulatory non-compliance, or catastrophic financial loss | Formal, tool-analyzable specification (AADL or comparable) for the affected subsystem; targeted ATAM-style evaluation at the most consequential decisions | Under-specified interfaces surfacing as integration defects of the architectural-mismatch kind |
| Architectural volatility | How often does the structure being described actually change? | Continuous, distributed structural change across releases | Approaches whose maintenance cost scales with the rate of change rather than requiring wholesale respecification | Measurable spatial and temporal drift of the specification within a small number of release cycles |
| Organizational scope | How many independently operating teams and ownership boundaries does the architecture span? | Many teams with independent release cadences and no single enforcing authority | Low-barrier shared conventions (C4, arc42) applied consistently; formality ceiling set by what crosses team boundaries | Documentation currency does not improve in response to notational interventions, because the binding constraint is organizational |
| Structural recoverability | Can the system's actual structure be derived automatically from artifacts the organization already maintains? | Deployment manifests, infrastructure-as-code, interface definitions, service catalogs, and traces already determine structure | Generate structural views; invest in recovery and narrow conformance checking; add a thin prescriptive layer for intent and prohibitions | Effort spent hand-authoring structure is dominated by a cheaper description that is current by construction |

| Factor | Diagnostic question | A high value indicates | Implication for the description approach | Predicted failure if mismatched |
|---|---|---|---|---|
| Relative half-life of structure and rationale | Which outlives which: the reasoning, or the structure it produced? | Rationale is stable while services are frequently re-partitioned | Concentrate authored effort on decision records rather than on structural diagrams; reverse the emphasis where structure is the durable artifact | The surviving artifact answers the wrong question: structure without intent, or intent without a current structural referent |

Note. Factors 1 through 3 restate and sharpen distinctions present in the reviewed literature; factors 4 and 5 are advanced by this review. Factors 1 and 2 dominate; among contexts outside the high-defect-cost, low-volatility cell, factor 4 determines whether structural description is authored or generated, factor 5 determines which artifact absorbs the effort, and factor 3 sets a ceiling on sustainable formality. The framework is a synthesized hypothesis, not a validated decision procedure; see Section 7.

The factors are not independent, and the framework is more useful when their interaction is made explicit. Factors one and two dominate: a context combining high defect cost with low volatility justifies formal specification more or less regardless of the remaining factors, and this is the AADL cell of the framework. Outside that cell, factor four determines whether structural description should be authored or generated, factor five determines whether effort should go to structure or to decisions, and factor three sets a ceiling on the formality that can realistically be sustained across organizational boundaries whatever the other factors recommend. A concrete application illustrates the difference between this and a generic appeal to context. For a payments platform of forty services owned by eight teams, with high defect cost, high volatility, wide organizational scope, high recoverability, and rationale that outlives structure, the framework does not recommend a compromise between formal and lightweight description. It recommends generated structural views derived from deployment artifacts rather than hand-drawn diagrams; a disciplined decision record practice as the primary authored artifact; automated conformance checks on a small number of narrowly specified invariants, such as which services may reach the ledger, rather than a general architectural specification; targeted formal or semi-formal analysis, including an evaluation exercise in the ATAM tradition, applied only to the ledger subsystem where defect cost concentrates; and no organization-wide formal notation at all, because factor three predicts it will not survive.

Stated this way, the framework generates predictions that could be wrong, which is the property a contingency claim needs in order to be worth more than the observation that circumstances differ. It predicts that a formal, hand-maintained architectural specification adopted in a high-volatility, wide-scope, high-recoverability context will exhibit measurable staleness, in the spatial and temporal drift senses operationalized by Romeo et al. (2024), within a small number of release cycles, and that this will occur regardless of the specification language chosen. It predicts, conversely, that decision-record-only practice adopted in a high-defect-cost, low-volatility, low-recoverability context will be associated with under-specified interfaces surfacing as integration defects of the kind Garlan et al. (1995) characterized. And it predicts that interventions targeting notation in a context whose binding constraint is factor three will not improve documentation currency, because the constraint is organizational. Each prediction is testable with the kinds of repository-mining and multiple-case designs already established in this literature (Le et al., 2015; Migliorini et al., 2024; Pontillo et al., 2026).

None of these five factors is a strict decision rule, and most real systems combine subsystems sitting at different points along each dimension, such that a mature architecture description practice today plausibly combines lightweight, team-level documentation using C4 or arc42 for the majority of a system's components with targeted, more formal specification, whether AADL, a domain-specific analysis tool, or rigorous architecture decision records, reserved for the specific subsystems or decisions where the cost of an undetected defect, the rate of change, or the organizational coordination burden makes that additional investment worthwhile. This contingency view is, in a sense, a return to the spirit rather than the letter of both classical works examined in this review. Kruchten (1995) never claimed that his four views were the only possible decomposition of architectural concerns, only that some such decomposition, tailored to a project's actual stakeholders, was necessary; and Medvidovic and Taylor (2000) never claimed that any single classical ADL was uniformly superior, only that the choice among them should be made deliberately, with the tradeoffs their framework makes explicit clearly in view. Applying that same deliberateness to the choice between classical formalism and contemporary lightweight documentation is the most direct continuation of both projects available to a contemporary architect.

### 6.5 Implications for Research and Practice

Two implications follow directly. For research, if derived description is now the default, the quality attributes that matter for an architecture description have changed, and the field needs evaluation criteria for derived descriptions addressing fidelity to the running system, stability under irrelevant infrastructure change, and the adequacy of the prescriptive layer that recovery cannot produce. The viewpoint conventions of ISO/IEC/IEEE 42010 offer a plausible frame for such criteria but have not, in the literature reviewed here, been applied to recovered

descriptions. The language-model thread should likewise be evaluated against this reframing rather than the classical one, since the gaps Schmid et al. (2025) identify, particularly conformance checking and cloud-native architecture, coincide with the capabilities the contingency framework identifies as most valuable and least supplied. For practice, the framework's most actionable consequence is a reordering of default investment: in most contemporary commercial contexts the first question is not which notation to adopt but whether the organization's existing machine-readable artifacts already determine the system's structure. Where they do, effort spent hand-drawing structure is largely wasted, and is better spent on generation, on a small number of narrowly specified automated invariants, and on the decision record practice that supplies the intent no generator can recover. Where they do not, the classical case for authored structural description remains intact and should be made without apology.

## 7. LIMITATIONS

This review has several limitations that should be considered when interpreting its conclusions. First, and most fundamentally, it is a narrative critical synthesis conducted by a single reviewer, not a database-executed systematic review with independent dual screening, a registered protocol, or a reproducible, exhaustively logged search process; the search approach disclosed in Section 3 was iterative and judgment-driven, and a different reviewer following a similarly structured but independently conducted search might identify a somewhat different, though likely substantially overlapping, set of sources. Second, the classical ADL literature this review draws on is itself comparatively thin and, in several cases, decades old, such that primary sources for some of the languages discussed, particularly Weaves and SADL, consist of a small number of publications with limited subsequent replication or extension, meaning this review's characterization of those languages rests on a narrower evidentiary base than its characterization of more extensively studied languages such as Wright or C2.

Third, the contemporary practice literature synthesized in Section 5 and drawn on in Section 6 is subject to a plausible recency and visibility bias: the practices, tools, and empirical studies most readily discoverable through the search strategy described in Section 3 are disproportionately those that have achieved significant industry visibility or academic citation, which may underrepresent architecture description practices that are effective within a particular organization or domain but have not been written up, published, or widely publicized. Fourth, the correspondences drawn in Section 6.3 between classical ADLs and contemporary infrastructure are interpretive claims made by this review, not historical claims of influence; no evidence is offered, and none should be inferred, that the designers of contemporary service meshes or pipeline orchestrators were aware of UniCon or Weaves, and the argument would be unaffected if they were not, since the point is that similar problems produced similar solutions.

Fifth, the three summary tables are analytical constructions rather than extracted data, as disclosed in Section 3.3, and their cells necessarily compress assessments that the underlying sources state with more qualification. In particular, Table 1's characterization of each classical language follows Medvidovic and Taylor's (2000) published assessment as reported in Section 4.3, and readers requiring a dimension-by-dimension account should consult that source directly rather than relying on the compressed summary offered here. Sixth, the literature on language-model support for architecture work surveyed in Section 5.5 is moving rapidly and is partly available only as preprints; the characterizations offered are therefore provisional, are used to establish the direction of a research thread rather than to support load-bearing empirical claims, and may be superseded quickly.

Seventh, this review's contingency framework in Section 6.4, while grounded in patterns identified across the reviewed literature, has not itself been empirically validated against a structured sample of real architecture decisions, and should be read as a synthesized hypothesis for future empirical testing rather than as an already-validated decision procedure; the predictions stated at the end of that subsection are offered precisely to make such testing possible. Finally, the review's scope was deliberately bounded to software and closely related systems architecture description. It does not address the broader enterprise architecture literature in depth beyond its point of convergence with ISO/IEC/IEEE 42010 and ArchiMate, and a reviewer more centrally interested in enterprise architecture as a management discipline would likely draw on a different and more extensive body of literature than the one assembled here.

## 8. CONCLUSION AND FUTURE WORK

This review set out to assess how well the classical software architecture description literature of the 1990s, specifically Kruchten's (1995) 4+1 view model and Medvidovic and Taylor's (2000) ADL classification framework, holds up against contemporary software architecture practice, and to identify what has transferred, what has not, and why. The analysis finds that the underlying intellectual logic of both works has proven considerably more durable than their specific notational apparatus: the separation-of-concerns reasoning behind Kruchten's four views recurs, in modified form, throughout contemporary documentation approaches such as the C4 model and arc42, and the conceptual distinction between components and connectors as independently significant architectural entities persists in how practitioners reason about service meshes and API gateways, even though the formal connector-typing languages Medvidovic and Taylor (2000) surveyed have themselves fallen out of mainstream use.

What has not survived is a set of three shared assumptions that the classical program never had reason to question: that a system has one configuration rather than a population of request-scoped topologies; that reconfiguration is an act performed against a specification by an identifiable agent; and that an architecture description is the source from which an implementation is derived rather than an artifact recovered from one. Each fails for a distinct reason, and together they explain the industrial retreat from formal architecture description more precisely than the usual appeal to agility or to tooling cost. Revisiting the ten classical languages individually reinforces the point: the language with the broadest coverage against the classification framework, Aesop, has no living descendant, while the one with the narrowest and most domain-bound design, MetaH, does, because its domain supplies exactly the combination of high defect cost and low volatility that formal description requires. Coverage was never the binding constraint.

The review's proposed contingency framework, distinguishing contexts by defect cost, architectural volatility, organizational scope, structural recoverability, and the relative half-lives of structure and rationale, offers a starting point for architects choosing among the now much wider menu of description approaches available, from AADL's formal, tool-supported rigor at one end to lightweight, continuously updated architecture decision records at the other. It is offered as a hypothesis rather than a validated finding, and it is stated in a form that admits refutation. Future work should test it empirically, for example through comparative case studies examining whether teams that select an architecture description approach consistent with the framework's factors report better documentation currency, lower onboarding cost, or fewer architecture-related incidents than teams whose chosen approach is mismatched to their context, and through repository-mining designs that operationalize documentation staleness in the manner of Romeo et al. (2024) across a population of systems classified by the framework's factors.

Three further directions follow from the analysis. Future work would benefit from updated, industrial-scale surveys in the tradition of Malavolta et al. (2013), specifically targeting contemporary cloud-native and microservice organizations, to determine whether the documentation challenges identified in more recent, smaller-scale studies generalize across a broader population of organizations and system types (Ernst & Robillard, 2023; Wan et al., 2023). Given the strong conceptual continuity this review identifies between classical multi-view architecture description and the viewpoint-and-view meta-model formalized in ISO/IEC/IEEE 42010, research might productively examine why that standard has not achieved the practical adoption its conceptual generality would seem to warrant, and whether the evidence that most projects maintain a single informal view rather than a coordinated set indicates a failure of tooling, of incentives, or of the multi-view premise itself in low-stakes settings (Migliorini et al., 2024). A second open problem is representational rather than empirical: the field lacks any notation for describing a population of topologies with associated frequencies, and the closest available formalisms, architectural styles and constraint languages, answer the different question of which topologies are permitted (Luo et al., 2021). Finally, the most consequential open problem this review identifies is the absence of a semantic connector-compatibility check for independently deployed services, the capability Wright provided within a single specified system and that no contemporary mechanism supplies across organizational boundaries (Allen & Garlan, 1997; Garlan et al., 1995). Whether that capability can be reconstructed in a form compatible with continuous delivery, whether through contract formalisms richer than current schema registries, through conformance metrics computed over recovered models (Ntentos et al., 2020; Alshuqayran et al., 2025), or through the language-model-assisted extraction whose current limitations Schmid et al. (2025) and Amalfitano et al. (2026) document, is in this review's assessment the question whose answer would do most to reconnect the classical architecture description program with the systems practitioners actually build.

**Accknowledgment**

**Ethics approval:** Not applicable.
**Consent for publication:** Not applicable.
**Data availability:** Not applicable. This is a literature-based critical review; no new datasets were generated or analyzed.
**Funding:** The author received no specific funding for this work.
**Competing interests: The** author declares no competing interests.

## References:

Ahmeti, B., Linder, M., Groner, R., & Wohlrab, R. (2024). Architecture decision records in practice: An action research study. In Proceedings of the 18th European Conference on Software Architecture (ECSA 2024) (pp. 1-16). Springer. DOI: 10.1007/978-3-031-70797-1_22

Allen, R., & Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology, 6*(3), 213-249. DOI: 10.1145/258077.258078

Alshuqayran, N., Ali, N., & Evans, R. (2025). A model-driven architecture approach for recovering microservice architectures: Defining and evaluating MiSAR. *Information and Software Technology, 186*, Article 107808. DOI: 10.1016/j.infsof.2025.107808

Amalfitano, D., De Luca, M., Santilli, T., Pelliccione, P., & Fasolino, A. R. (2026). Automated software architecture design recovery from source code using LLMs. In Software architecture: 19th European Conference on Software Architecture, ECSA 2025 (Lecture Notes in Computer Science, Vol. 15929, pp. 73-89). Springer. DOI: 10.1007/978-3-032-02138-0_5

arc42. (n.d.). arc42: Template for software architecture documentation. Retrieved August 29, 2026, from https://arc42.org

Assunção, W. K. G., Krüger, J., Mosser, S., & Selaoui, S. (2023). How do microservices evolve? An empirical analysis of changes in open-source microservice repositories. *Journal of Systems and Software, 204*, Article 111788. DOI: 10.1016/j.jss.2023.111788

Bass, L., Clements, P., & Kazman, R. (2021). Software architecture in practice (4th ed.). Addison-Wesley.

Bass, L., Weber, I., & Zhu, L. (2015). DevOps: A software architect's perspective. Addison-Wesley.

Binns, P., Englehart, M., Jackson, M., & Vestal, S. (1996). Domain-specific software architectures for guidance, navigation, and control. International Journal of Software Engineering and Knowledge Engineering, 6(2), 201-227.

Bosch, J. (2000). Design and use of software architectures: Adopting and evolving a product-line approach. Addison-Wesley.

Brown, S. (2018, October). The C4 model for software architecture. InfoQ. https://www.infoq.com/articles/C4-architecture-model/

Bucaioni, A., Di Salle, A., Iovino, L., Pelliccione, P., & Raimondi, F. (2025). Architecture as code. In 2025 IEEE 22nd International Conference on Software Architecture (ICSA). IEEE.

Buchgeher, G., Schöberl, S., Geist, V., Dorninger, B., Haindl, P., & Weinreich, R. (2023). Using architecture decision records in open source projects: An MSR study on GitHub. *IEEE Access, 11,* 63725-63740. DOI: 10.1109/ACCESS.2023.3287654

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010). Documenting software architectures: Views and beyond (2nd ed.). Addison-Wesley.

de Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software, 85*(1), 132-151. DOI: 10.1016/j.jss.2011.07.036

Dhar, R., Vaidhyanathan, K., & Varma, V. (2024). Can LLMs generate architectural design decisions? An exploratory empirical study [Preprint]. arXiv. DOI: 10.48550/arXiv.2403.01709

Erder, M., Pureur, P., & Woods, E. (2021). Continuous architecture in practice: Software architecture in the age of agility and DevOps. Addison-Wesley.

Ernst, N. A., & Robillard, M. P. (2023). A study of documentation for software architecture. *Empirical Software Engineering, 28*(4), Article 96. DOI: 10.1007/s10664-023-10347-2

Feiler, P. H., & Gluch, D. P. (2012). Model-based engineering with AADL: An introduction to the SAE architecture analysis and design language. Addison-Wesley.

Feiler, P. H., Gluch, D. P., & Hudak, J. J. (2006). The architecture analysis & design language (AADL): An introduction (CMU/SEI-2006-TN-011). Software Engineering Institute, Carnegie Mellon University.

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures (Publication No. 9980887) [Doctoral dissertation, University of California, Irvine]. ProQuest Dissertations Publishing.

Ford, N., Richards, M., Sadalage, P., & Dehghani, Z. (2021). Software architecture: The hard parts: Modern trade-off analyses for distributed architectures. O'Reilly Media.

Fowler, M., & Lewis, J. (2014, March 25). Microservices: A definition of this new architectural term. martinfowler.com. https://martinfowler.com/articles/microservices.html

Garlan, D., Allen, R., & Ockerbloom, J. (1994). Exploiting style in architectural design environments. In Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (pp. 175-188). ACM. DOI: 10.1145/193173.195579

Garlan, D., Allen, R., & Ockerbloom, J. (1995). Architectural mismatch: Why reuse is so hard. *IEEE Software, 12*(6), 17-26. DOI: 10.1109/52.469757

Garlan, D., Monroe, R. T., & Wile, D. (1997). Acme: An architecture description interchange language. In Proceedings of CASCON '97 (pp. 169-183). IBM.

Garlan, D., & Shaw, M. (1993). An introduction to software architecture. In V. Ambriola & G. Tortora (Eds.), Advances in software engineering and knowledge engineering (Vol. 2, pp. 1-39). World Scientific. DOI: 10.1142/9789812798039_0001

Genfer, P., Serbout, S., Simhandl, G., Zdun, U., & Pautasso, C. (2025). Understanding security tactics in microservice APIs using annotated software architecture decomposition models: A controlled experiment. *Empirical Software Engineering, 30*(3), Article 66. DOI: 10.1007/s10664-024-10601-1

Gorlick, M. M., & Razouk, R. R. (1991). Using Weaves for software construction and analysis. In Proceedings of the 13th International Conference on Software Engineering (pp. 23-34). IEEE. DOI: 10.1109/ICSE.1991.130630

Hilliard, R. (1999). Using the UML for architectural description. In R. France & B. Rumpe (Eds.), UML'99: The Unified Modeling Language: Beyond the standard (Lecture Notes in Computer Science, Vol. 1723, pp. 32-48). Springer. DOI: 10.1007/3-540-46852-8_4

IEEE. (2000). IEEE Std 1471-2000: IEEE recommended practice for architectural description of software-intensive systems. Institute of Electrical and Electronics Engineers.

International Organization for Standardization, International Electrotechnical Commission, & Institute of Electrical and Electronics Engineers. (2011). ISO/IEC/IEEE 42010:2011: Systems and software engineering: Architecture description. ISO/IEC/IEEE.

International Organization for Standardization, International Electrotechnical Commission, & Institute of Electrical and Electronics Engineers. (2022). ISO/IEC/IEEE 42010:2022: Software, systems and enterprise: Architecture description. ISO/IEC/IEEE.

Jansen, A., & Bosch, J. (2005). Software architecture as a set of architectural design decisions. In Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (pp. 109-120). IEEE. DOI: 10.1109/WICSA.2005.61

Jongeling, R., Strøm, N. J., Nissen, L. P. T., Kitchen, M., & Carlson, J. (2026). Adopting the C4 model for lightweight architecture modeling: An experience report. In Software engineering and advanced applications: SEAA 2025 (Lecture Notes in Computer Science, Vol. 16081, pp. 393-409). Springer. DOI: 10.1007/978-3-032-04190-6_24

Kazman, R., Klein, M., & Clements, P. (2000). ATAM: Method for architecture evaluation (CMU/SEI-2000-TR-004). Software Engineering Institute, Carnegie Mellon University.

Kruchten, P. (1995). Architectural blueprints: The "4+1" view model of software architecture. *IEEE Software, 12*(6), 42-50. DOI: 10.1109/52.469759

Kruchten, P., Obbink, H., & Stafford, J. (2006). The past, present, and future for software architecture. *IEEE Software, 23*(2), 22-30. DOI: 10.1109/MS.2006.59

Le, D. M., Behnamghader, P., Garcia, J., Link, D., Shahbazian, A., & Medvidovic, N. (2015). An empirical study of architectural change in open-source software systems. In Proceedings of the 12th Working Conference on Mining Software Repositories (pp. 235-245). IEEE. DOI: 10.1109/MSR.2015.29

Li, R., Liang, P., Soliman, M., & Avgeriou, P. (2021). Understanding architecture erosion: The practitioners' perceptive. In Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC). IEEE. DOI: 10.48550/arXiv.2103.11392

Li, R., Liang, P., Soliman, M., & Avgeriou, P. (2022). Understanding software architecture erosion: A systematic mapping study. *Journal of Software: Evolution and Process, 34*(3), Article e2423. DOI: 10.1002/smr.2423

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., & Mann, W. (1995). Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, 21*(4), 336-354. DOI: 10.1109/32.385971

Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., & Xu, C. (2021). Characterizing microservice dependency and performance: Alibaba trace analysis. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '21) (pp. 412-426). ACM. DOI: 10.1145/3472883.3487003

Magee, J., & Kramer, J. (1996). Dynamic structure in software architectures. In Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (pp. 3-14). ACM. DOI: 10.1145/239098.239104

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., & Tang, A. (2013). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering, 39*(6), 869-891. DOI: 10.1109/TSE.2012.74

Medvidovic, N., Oreizy, P., Robbins, J. E., & Taylor, R. N. (1996). Using object-oriented typing to support architectural design in the C2 style. In Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (pp. 24-32). ACM. DOI: 10.1145/239098.239106

Medvidovic, N., Rosenblum, D. S., & Taylor, R. N. (2002). Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology, 11*(1), 2-57. DOI: 10.1145/504087.504088

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering, 26*(1), 70-93. DOI: 10.1109/32.825767

Migliorini, S., Verdecchia, R., Malavolta, I., Lago, P., & Vicario, E. (2024). Architectural views: The state of practice in open-source software projects. In Software architecture: 18th European Conference on Software Architecture, ECSA 2024 (Lecture Notes in Computer Science, Vol. 14889, pp. 396-415). Springer. DOI: 10.1007/978-3-031-70797-1_27

Moriconi, M., & Riemenschneider, R. A. (1997). Introduction to SADL 1.0: A language for specifying software architecture hierarchies (Technical Report SRI-CSL-97-01). SRI International Computer Science Laboratory.

Nazir, R., Bucaioni, A., & Pelliccione, P. (2024). Architecting ML-enabled systems: Challenges, best practices, and design decisions. *Journal of Systems and Software, 207,* Article 111860. DOI: 10.1016/j.jss.2023.111860

Newman, S. (2021). Building microservices: Designing fine-grained systems (2nd ed.). O'Reilly Media.

Ntentos, E., Zdun, U., Plakidas, K., Meixner, S., & Geiger, S. (2020). Assessing architecture conformance to coupling-related patterns and practices in microservices. In Software architecture: 14th European Conference on Software Architecture, ECSA 2020 (Lecture Notes in Computer Science, Vol. 12292, pp. 3-20). Springer. DOI: 10.1007/978-3-030-58923-3_1

Nygard, M. (2011, November 15). Documenting architecture decisions. Cognitect Blog. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions

Object Management Group. (2019). OMG Systems Modeling Language (SysML), version 1.6. Object Management Group.

Odejobi, O. D., Okonkwo, C. S., Patrick, M. C. A., Okeke, O. T., & Mayo, W. (2025). AI-augmented secure software engineering: Leveraging deep learning for autonomous threat detection and mitigation. *International Journal of Engineering and Modern Technology, 11*(12), 101-121. DOI: 10.56201/ijemt.vol.11.no12.2025.pg101.121

Okonkwo, C. S., Patrick, M. C. A., Okeke, O. T., & Abolaji, T. O. (2023). A conceptual framework for integrating IT systems engineering with organizational supply-chain operations. *International Journal of Multidisciplinary Research and Growth Evaluation, 4*(6), 712-735. DOI: 10.62225/2583049X.2023.4.6.5500

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, 17(4), 40-52. DOI: 10.1145/141874.141884

Pontillo, V., Bucaioni, A., Carnevale, I., Di Salle, A., Manzoni, L., Pelliccione, P., & Sprotetto, F. M. (2026). Architecture as code in industry: A multiple-case empirical study. In Proceedings of the IEEE International Conference on Software Architecture (ICSA 2026). IEEE.

Richardson, C. (2018). Microservices patterns: With examples in Java. Manning Publications.

Romeo, J., Raglianti, M., Nagy, C., & Lanza, M. (2024). Capturing and understanding the drift between design, implementation, and documentation. In Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24). ACM. DOI: 10.1145/3643916.3644399

Rozanski, N., & Woods, E. (2005). Software systems architecture: Working with stakeholders using viewpoints and perspectives. Addison-Wesley.

SAE International. (2017). Architecture analysis & design language (AADL), AS5506C. SAE International.

Schmid, L., Hey, T., Armbruster, M., Corallo, S., Fuchß, D., Keim, J., Liu, H., & Koziolek, A. (2025). Software architecture meets LLMs: A systematic literature review [Preprint]. arXiv. DOI: 10.48550/arXiv.2505.16697

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., & Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, 21*(4), 314-335. DOI: 10.1109/32.385970

Shaw, M., & Garlan, D. (1996). Software architecture: Perspectives on an emerging discipline. Prentice Hall.

Söylemez, M., Tekinerdogan, B., & Tarhan, A. K. (2024). Microservice reference architecture design: A multi-case study. Software: *Practice and Experience, 54*(1), 58-84. DOI: 10.1002/spe.3241

Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2010). Software architecture: Foundations, theory, and practice. Wiley.

The Open Group. (2024). ArchiMate 3.2 specification. The Open Group. https://pubs.opengroup.org/architecture/archimate3-doc/

Tyree, J., & Akerman, A. (2005). Architecture decisions: Demystifying architecture. IEEE Software, 22(2), 19-27. DOI: 10.1109/MS.2005.27

Wan, Z., Zhang, Y., Xia, X., Jiang, Y., & Lo, D. (2023). Software architecture in practice: Challenges and opportunities. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23) (pp. 1457-1469). ACM. DOI: 10.1145/3611643.3616367

Zhou, X., Li, R., Liang, P., Zhang, B., Shahin, M., Li, Z., & Yang, C. (2025). Using LLMs in generating design rationale for software architecture decisions. ACM Transactions on Software Engineering and Methodology. Advance online publication. DOI: 10.1145/3785010

**Kingsley Chinazaekpere Ndupu**
Kennesaw State University, USA
Kingsleynd34@gmail.com
**ORCID:** 0009-0004-1981-479X

**Serif Oyindamola Oyesiji**
Harrisburg University of Science and Technology, USA
oyesijiserif@gmail.com
**ORCID:** 0009-0000-1203-8063

**Chukwudera Obumneke Anunagba**
Barclays, USA
chiokemuo@gmail.com
**ORCID:** 0009- 0003-3949-5290