Architectural Decision Enforcement: From Architectural Decisions to Automated Conformance Checks

Master Thesis

Raphael Schellander

Supervisor: Stefan Kapferer
External Examiner: Dr. Gerald Reif

Eastern Switzerland University of Applied Sciences

June 26, 2026

Abstract

Architectural decisions play a central role in software architecture and are often documented using Architectural Decision Records (ADRs). Managing such decisions involves identifying architecturally significant choices, documenting them as ADRs, and enforcing them as the codebase evolves. In practice, enforcement is often the weakest phase and is usually left to costly and inconsistent code reviews. Existing architecture conformance tools can automate checks on code, but they are disconnected from the ADRs that motivated their rules. This gap can lead to architectural erosion over time.

This thesis proposes to bridge the gap between decision documentation and automated conformance checking with the Architectural Decision Enforcement (ADE) tool. ADE introduces a tool-agnostic domain-specific language (DSL) in which rule files, located alongside ADRs, express enforceable constraints without knowledge of specific testing frameworks. To decouple rule specification from execution, we developed a compiler architecture consisting of a parser, semantic validation, and an intermediate representation based on Protocol Buffers. This allows the same rule file to be used with multiple enforcement frameworks. We further define a plugin protocol that enables independently developed plugins to target different language ecosystems.

The proposed approach supports architects in enforcing architectural decisions throughout software evolution. We implemented a command-line tool with reference plugins for Go, .NET, and filesystem assertions. Applying ADE to an open-source .NET project containing 17 ADRs showed that 9 decisions could be automatically enforced, resulting in 44 generated checks. As part of our action research, an architect applied ADE to a multi-language project and developed an independent Java plugin. This demonstrated that the plugin protocol can be applied to new ecosystems without changes to the ADE core. The validation activities indicate that a tool-agnostic DSL combined with a plugin-based execution model can effectively connect ADRs with automated conformance checks, minimising architectural erosion.

Contents

1 Introduction

The design of many meaningful software systems is based on a series of architectural decisions relating to decomposition, technology, communication style and structural conventions. These decisions are made at specific times for particular reasons by specific individuals. These decisions primarily convey design rationale. If they are not made explicit, they only exist in the memories of those who made them. If they are not enforced, they are quietly overridden as the system grows and the original context fades. This thesis addresses these two issues by proposing a toolchain that links automated conformance checks directly to the decision records that initiated them.

The following subsections outline the motivation and problem statement behind this thesis, situate it within the context of previous work, and define the research objective and contributions.

1.1 Motivation

Software architecture is shaped by both structural representations and the decisions taken throughout a system's lifetime. Jansen and Bosch [1] characterise an architecture as a "composition of a set of architectural design decisions" rather than a static arrangement of components and connectors, and Kruchten [2] similarly argues that decisions, including their rationale, alternatives, and consequences, are first-class entities of the architecting process. To capture these decisions in a lightweight and developer-friendly way, the community has adopted *Architectural Decision Records* (ADRs)[1]. ADRs explain *why* a system is built the way it is, and they provide the traceability that purely structural views cannot.

Documenting a decision, however, is only the first part of the process. Once an ADR is written, the implementation must continue to adhere to it as the system evolves. In practice, this rarely happens automatically. Implementations gradually diverge from their intended architecture, a phenomenon described in the literature as *architectural drift* and *architectural erosion* [3]. Drift introduces design choices that were not originally planned but do not directly contradict the intended architecture, whereas erosion introduces changes that violate its guiding principles outright. Both are amplified by release pressure, team turnover, and the absence

[1] `https://adr.github.io`

of systematic enforcement [4, 5]. Over time, the architecture documented in the ADRs and the architecture realised in the source code drift apart, and the rationale captured at the time of the decision becomes harder and harder to recover.

1.2 Problem Statement

Architectural decisions can be structured into three phases: *decision identification*, *decision making*, and *decision enforcement* [6]. The first two phases are reasonably well supported in practice. There are templates for documenting decisions, command-line tools for creating, managing, and navigating them, and research prototypes that help architects identify and reason about decisions in a structured way [6, 7, 8]. The third phase, by contrast, is the weakest area. Although a wide range of conformance-checking tools exists, ranging from architecture test frameworks such as ArchUnit[2], ArchUnitNet[3], Arch-Go[4], and NetArchTest[5] to dedicated rule engines such as Sonargraph [9], these tools operate purely at the code level. They are decoupled from the ADRs that describe the decisions they are supposed to enforce, and the rules they execute must be written entirely separately from the documentation.

The result is a gap between decision documentation and technical enforcement that must be addressed manually. For each documented decision that needs to be enforced, an architect or developer must translate the ADR into a tool-specific test or analysis rule, while also keeping the two in sync as the system evolves. This repetitive process is prone to error. Any change to a decision can result in the corresponding rule becoming outdated, and the reason why the rule was created in the first place can easily be lost, as there is no trace linking back to the decision documentation. This makes it easy for developers to change the test without understanding its purpose. As system complexity increases, the gap widens, and despite being formally documented, architectural intent silently erodes.

[2] https://www.archunit.org
[3] https://archunitnet.readthedocs.io/en/stable/
[4] https://github.com/arch-go/arch-go
[5] https://github.com/BenMorris/NetArchTest

1.3 Prior Work

This thesis is the third in a sequence on architectural decision support carried out at the Eastern Switzerland University of Applied Sciences. In the first project [7], recurring architectural decisions from the Clean Architecture literature were analysed, the existing ADR tooling landscape was examined, and a conceptual design for a new command-line tool was proposed. In the second project [8], this concept was realised as the *Architectural Decision Guidance* (ADG) tool[6], a Go-based CLI for managing ADRs through a structured model with support for nesting, linking, tagging, validation, and reuse of decision models. ADG covers the first two phases of the decision lifecycle, namely identification and decision making. The third phase, decision enforcement, was deliberately left out of scope at the time and is the subject of this thesis.

1.4 Research Objective and Contributions

The objective of this thesis is to develop and evaluate a conceptual approach, together with a working toolchain, for the automated enforcement of architectural decisions in software projects. This approach must remain based on ADRs that describe the decisions, has to be independent of any specific programming language or analysis framework, and has to leave room for future extension to additional kinds of rules and target ecosystems.

The work is organised around five concrete deliverables:

- The design of a domain-specific language (DSL) for describing enforceable aspects of architectural decisions in a structured, machine-readable form.
- The definition of a generic intermediate representation that decouples the DSL from individual analysis and enforcement tools.
- The implementation of a standalone command-line tool, ADE (Architectural Decision Enforcement), that parses, validates, and executes rule files, and is additionally packaged as a Go library so that it can be embedded into other tools.
- The integration of ADE into the existing ADG tool, exposing it as the `adg enforce` subcommand and co-locating rule files with the ADRs they enforce.

[6] `https://github.com/adr/ad-guidance-tool`

- Three reference plugins covering Go, .NET, and the filesystem, written as independent executables that communicate with ADE through a standardised protocol.

Together, these deliverables form a continuous workflow that runs from *decision identification* and *decision making* with the existing ADG commands to *decision enforcement* with ADE. In addition to the deliverables, this thesis makes the following contributions:

- A systematic literature review on the state of research in architectural decision documentation and enforcement, which provides the conceptual foundation for the thesis.
- A case study evaluating the toolchain on an open-source .NET repository with seventeen existing ADRs, in which the ADRs are classified by decision type, the enforceable subset is identified, and rules are created and executed using the toolchain directly within the CI pipeline.

The remainder of this thesis is structured as follows. Section 2 introduces the conceptual background on architectural decisions, ADRs, drift and erosion, decision taxonomies, conformance checking, and DSL design. Section 3 positions the work against existing research on reusable decision models, decision-to-constraint mapping, and DSL-based architecture conformance. Section 4 derives the requirements, presents the conceptual approach including the DSL, the intermediate representation, and the plugin model, and describes the implementation and CI integration. Section 5 reports on the case study carried out on the open-source repository, and Section 6 discusses the findings, trade-offs, limitations, and threats to validity of the case study. Section 7 concludes the thesis and outlines directions for future work.

2 Background and Analysis

This section establishes the conceptual terminology and research context necessary to understand the approach developed in this thesis. It is structured to follow the logical progression from the nature of architectural decisions and how they are managed, to the consequences of failing to manage them systematically, and finally to the technical mechanisms that can be used to close that gap.

Section 2.1 introduces the concept of the architectural decision lifecycle and argues why the enforcement phase is the weakest area in current practice. Section 2.2 covers Architectural Decision Records (ADRs) as the documentation artefact that this thesis uses as its starting point. Section 2.3 explains architectural drift and erosion, the phenomena that motivate automated enforcement in the first place. Section 2.4 presents decision taxonomies and analyses which decision categories are suitable for automated checking. Section 2.5 provides an overview of architecture conformance checking tools and techniques. Finally, Section 2.6 discusses the role of domain-specific languages in expressing architectural rules in a readable, tool-agnostic form.

2.1 Architectural Decision Lifecycle

The history of software architecture as a discipline has gradually shifted from a focus on structural artefacts, such as components, connectors, and deployment views, towards a recognition that the decisions underlying those artefacts are what actually constitute the architecture. Jansen and Bosch [1] make this point by proposing that a software architecture should be understood as "a composition of a set of architectural design decisions" rather than as a static diagram, and by arguing that the root cause of common architectural problems, such as erosion, high change costs, and limited reusability, is the *knowledge vaporization* that occurs when decision rationale is lost. Under this view, the architecture of a system is not stored in its diagrams but in the accumulated set of deliberate choices that have been made and justified throughout the system's lifetime.

Kruchten [2] extends this perspective by proposing a formal ontology of architectural design decisions, in which decisions are treated as first-class entities with their own attributes, relationships, and lifecycle states. He categorises decisions into three major kinds:

- *Existence decisions* ("ontocrises") state that some element or artefact will positively exist in the system's design or implementation; they include structural decisions about components, layers, and subsystems, as well as behavioural decisions about how elements interact.
- *Property decisions* ("diacrises") state an enduring, overarching quality of the system, expressed as a design rule, a design constraint, or a guideline.
- *Executive decisions* ("pericrises") concern the environment of development, like technology choices, process decisions, and organisational constraints.

As Kruchten notes, property decisions are particularly prone to being implicit and quickly forgotten because they are cross-cutting concerns that cannot easily be localised to a single artefact. They are therefore also the hardest to preserve under changing teams and evolving requirements. The practical significance of this classification for enforcement will be revisited in Section 2.4.

The Three-Phase Lifecycle

The recognition that architectural decisions are first-class entities raises the question of how they should be managed throughout a project's lifetime. Zimmermann et al. [6] propose a conceptual framework of reusable architectural decision models that structures this management activity into three distinct phases: *decision identification*, *decision making*, and *decision enforcement*.

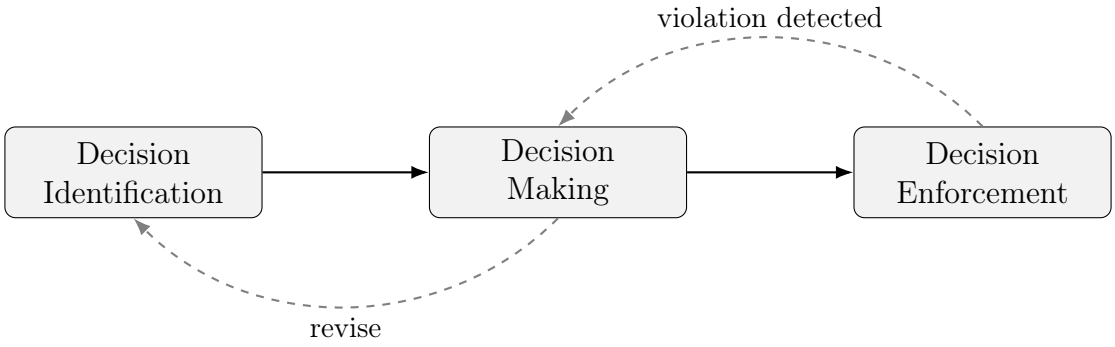

Figure 1: The three-phase architectural decision lifecycle (based on Zimmermann et al. [6]).

Decision identification is the activity of recognising which design choices in a given project are architecturally significant and therefore deserve to be made explicit and documented. Zimmermann et al. propose semi-automatic instantiation of a decision model from requirements artefacts and a reusable catalogue of decision templates ("design spaces"), which pushes a to-do list of decisions to the team instead of relying on ad-hoc identification. Without such support, architects and developers tend to rely on intuition and experience, and many significant decisions remain implicit [10].

Decision making is the activity of selecting an option and recording the rationale, alternatives considered, constraints applied, and expected consequences. A wide variety of support methods have been proposed, including Quality-Option-Criteria (QOC) tables [11], SWOT analysis, Architecture Tradeoff Analysis Method (ATAM) [12], and multi-criteria scoring [6]. In practice, the community has adopted lightweight textual formats such as ADRs to capture this information, which are discussed in more detail in Section 2.2.

Decision enforcement is the activity of ensuring that the outcomes of previously made decisions are actually realised in the system as it is built and evolves. Zimmermann et al. [6] treat enforcement as comprising both human-process mechanisms (coaching, governance, reviews) and technical mechanisms. The latter include what they call *decision injection*: feeding machine-readable decision outcomes into model transformations, code generators, and configuration policies so that compliance is ensured structurally rather than verified manually. Beyond their specific context of model-driven development, the enforcement phase is the least mature and most inconsistently supported of the three.

Figure 1 illustrates the three phases and their feedback relationships. Two feedback arcs extend the otherwise linear flow. The first, labelled *violation detected*, runs from decision enforcement back to decision making. The second, labelled *revise*, runs from decision making back to decision identification. It is important to note that a detected violation does not, by itself, imply that the original decision was wrong. In the common case the decision was correct and the enforcement tool has simply found a legitimate breach in the implementation; the appropriate response is to fix the code, not to revisit the decision. However, a violation can sometimes expose a scenario that was not anticipated when the decision was originally made. When the constraint turns out to be incorrect, incomplete, or impractical in ways that only become apparent through enforcement, the team must re-enter the decision-making phase to revise or supersede the affected decision. In turn, revising an existing decision may reveal that the underlying question itself was

framed too narrowly, or that a related decision that was never explicitly identified is now needed. In that case decision making feeds back further still into decision identification, starting the cycle anew. These feedback loops are particularly important in agile projects, where the system evolves continuously and each iteration may challenge previously made decisions.

The Enforcement Gap in Practice

Empirical evidence consistently shows that the enforcement phase is where current practice falls short the most. LaToza et al. [10] conducted interviews at two software companies and found that while meetings and informal knowledge repositories help communicate architectural decisions within a team, "code reviews are ultimately necessary to ensure conformance". This reliance on manual reviews is costly. Reviews must be carried out at the right moment (after the change and before merge), reviewers must be familiar enough with the architecture to recognise violations, and feedback is neither reproducible nor tracked systematically. The study also found that costly rework occurs when a fundamental limitation of a chosen technology becomes apparent only through actual use, being invisible at the time of decision-making.

The Architectural Decision Guidance (ADG) tool developed in the prior project work [8] focuses on the first two phases. It provides a command-line interface to create, edit, link, tag, validate, and reuse decision models in a structured way. The third phase, enforcement, was deliberately left out of scope at the time. This thesis picks up where that project left off. The goal is to bridge the gap between what is documented in ADRs and what is automatically checked in the build pipeline, using the three-phase lifecycle as the conceptual framework throughout.

2.2 Architectural Decision Records

Architectural Decision Records are lightweight text documents, typically stored as Markdown files alongside the source code, that record the context, outcome, and rationale of a significant architectural decision. The format was popularised by Michael Nygard[7], who proposed a minimal four-field template comprising context, decision, status, and consequences. There are various templates that extend or

[7] https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions

adapt the concept of Nygard's initial proposal[8]. For example, MADR (Markdown Architectural Decision Records)[9] extends this template with fields for listing considered options, their respective pros and cons, and the criteria that led to the chosen option. Another example are Y-statements[10], which offer a compressed single-sentence format aimed at rapid identification and communication. These formats share the same essential purpose: to preserve the *why* behind a design choice at the moment the choice is made, so that future readers can understand and, if necessary, challenge it.

ADRs are typically stored in a designated directory within the repository and managed under version control alongside the code they govern. This co-location is intentional; by living in the same repository, ADRs evolve with the codebase and can be referenced from pull requests, commit messages, and code comments. Tools such as `adr-tools`[11], `adr-log`[12], and `adr-viewer`[13] automate common ADR workflows, such as creating new records from templates, generating a linked table-of-contents log, and rendering the log as a browsable HTML site, respectively.

As an example, Listing 1 shows an ADR verbatim from the ADG tool's own repository[14], which constrains the dependency direction between the four layers of a Clean Architecture: domain, application, adapter, and infrastructure. *AD0001* was created and managed using the ADG tool itself and is used as an example throughout the thesis to illustrate the different contexts discussed, showcasing its form across the full decision lifecycle.

[8] https://adr.github.io/adr-templates/
[9] https://adr.github.io/madr/
[10] https://medium.com/olzzio/y-statements-10eb07b5a177
[11] https://github.com/npryce/adr-tools
[12] https://github.com/adr/adr-log
[13] https://github.com/mrwilson/adr-viewer
[14] https://github.com/adr/ad-guidance-tool/blob/main/docs/adr/AD0001-domain-layer-is-independent-from-upper-layers.md

Listing 1: AD0001 “Domain layer is independent from upper layers”.

```
Question

How should the core domain model relate to the layers above it?

Options

1. Keep domain independent: domain packages may not import application,
    adapter, or infrastructure packages
2. Allow upward imports: domain packages may freely import any layer

Criteria

Domain logic must be reusable independently of any delivery mechanism or
    persistence technology. Coupling domain to upper layers makes it
    impossible to test or reuse in isolation.

Outcome

We decided for Option 1 because: The domain layer encodes business rules
    that must remain stable regardless of how the tool is delivered or how
     data is stored. Keeping domain independent ensures that use-case
    logic and infrastructure can change without affecting the core model.

Comments

1. (2026-05-15 18:40:35) : marked decision as decided
```

2.3 Architectural Drift and Erosion

Two related but distinct forms of architectural degradation have been identified in the literature. Whiting and Andrews [4] characterise *architectural drift* as the accumulation of design choices that were not specified in the intended architecture but do not directly violate it. These choices extend or bypass the design without contradicting it. *Architectural erosion*, by contrast, is the introduction of changes that positively violate the prescribed architecture, breaking its constraints and compromising the properties it was designed to guarantee. The two phenomena are related: a system that has drifted may later suffer erosion as unofficial additions accumulate, making it progressively harder to identify the boundary be-

tween an unspecified extension and a prohibited violation. Whiting and Andrews additionally classify the conformance state of any architectural element into three levels:

- *Convergence.* The implemented element satisfies the intended architecture.
- *Absence.* A planned relationship or constraint was never implemented.
- *Divergence.* A component relationship exists in the implementation that is specifically forbidden by the intended architecture.

This three-way classification provides a useful frame for what conformance checks report.

Li et al. [5] conducted a systematic mapping study of 73 empirical studies on architecture erosion and identified a consistent pattern. Erosion manifests at multiple levels simultaneously. At the structural level, the most commonly reported symptoms are unwanted dependencies between modules (particularly cross-layer calls and cyclic dependencies) and violations of interface contracts; at the quality level, the consequences include reduced testability, increased change cost, and accumulated technical debt. Notably, the study found that the reasons for erosion are distributed roughly equally between technical factors (accumulated quick fixes, missing documentation, tooling gaps) and non-technical factors (schedule pressure, team turnover, and inadequate communication of architectural intent). This finding has a direct implication for enforcement design. Purely mechanical tools can address the technical causes, but addressing the non-technical causes requires the intended architecture to be continuously visible and checkable in the daily development workflow, not only reviewed retroactively.

The consequences of unchecked erosion are not just qualitative. Le et al. [13] analysed 466 versions of ten open-source systems and demonstrated empirically that the density of architectural smells—measurable symptoms of decay such as cyclic dependencies, ambiguous interfaces, and unstable dependencies—is a strong predictor of both future issue-proneness and change-proneness. Systems with higher architectural decay showed significantly more implementation-level defects in subsequent versions and required proportionally more effort per change. This provides quantitative motivation for treating architectural conformance as a first-class engineering concern rather than a periodic audit activity.

Countermeasures identified in the literature fall into three broad categories [4]:

- *Organisational competence* refers to governance processes, architectural mentoring, and ensuring that original architects remain available to advise on evolving decisions.
- *Reactive detection* encompasses static analysis tools that parse the codebase for dependency violations, code smells, or metric thresholds and report findings asynchronously.
- *Proactive enforcement* comprises approaches that make conformance a precondition of integration, preventing violations from being introduced rather than detecting them after the fact. ADE, the enforcement tool developed in this thesis, belongs to the third category: it integrates conformance checks into the build pipeline and fails the build as soon as a rule is broken, before the violation reaches the main branch.

2.4 Decision Taxonomies and Enforceability

Not all architectural decisions are equally suited to automated enforcement. The categories introduced in Section 2.1 differ substantially in how precisely they can be expressed as machine-checkable assertions, and understanding this variation is prerequisite to defining the scope of the domain-specific language (DSL). Keim et al. [14] propose a fine-grained taxonomy, illustrated in Figure 2, which extends Kruchten's ontology [2] with a finer-grained classification designed to support automated consistency analysis.

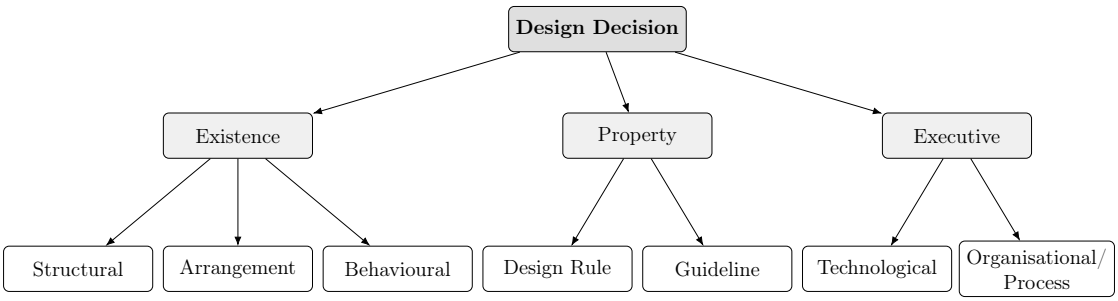

Figure 2: Taxonomy of design decisions proposed by Keim et al. [14].

Within the existence category, *structural* decisions—the presence of specific modules, packages, services, or layers—are very applicable to automated checking because they manifest as named artefacts whose structure can be matched against a pattern, like a package glob, a class name convention, a directory path. *Arrangement* decisions describe how elements are organised according to known architectural styles and patterns, such as a layered architecture or a client-server topology; their structural manifestations—the existence and naming of layers, the dependency directions between them—are partially enforceable through naming and dependency constraints, though the full semantic intent of an architectural style usually exceeds what a static analyser can verify on its own. *Behavioural* existence decisions (how elements interact at runtime) are harder to verify statically; they often require dynamic analysis, integration tests, or contract-based testing that falls outside the scope of source-code conformance checking.

Within the property category, *design-rule* decisions express explicit prohibitions or requirements: "the domain layer must not depend on the infrastructure layer", "all public API types must be documented". These map directly onto dependency queries or structural assertions that static analysis tools can evaluate. AD0001 (Listing 1) is precisely such a design-rule decision: it prescribes a structural constraint on import dependencies between four named layers, allowing for a binary pass/fail evaluation. *Guideline* decisions, by contrast, express preferences whose satisfaction depends on interpretation and context: "prefer small, single-responsibility classes" cannot be reduced to a pass/fail test without additional threshold parameters that are themselves a design choice.

Executive decisions are largely beyond the reach of static structural analysis because they concern the organisational context, not the code itself.

Keim et al.'s taxonomy was derived from mining seventeen open-source software architecture documentation repositories and classifying the design decisions found therein. Their empirical analysis confirms that design-rule decisions constitute a substantial and practically significant fraction of all documented decisions, and identifies them as the most important category for automated consistency checking. Table 1 summarises the enforceability profile of each decision category.

Table 1: Enforceability of architectural decision categories by automated static analysis.

Category	Enforceable?	Notes
Existence: structural	Yes	Structural aspects of the system can be checked statically
Existence: arrangement	Partially	Structural manifestations of a pattern (layer names, dependency directions) are checkable
Existence: behavioural	Partially	Runtime behaviour requires dynamic analysis
Property: design rule	Yes	Maps directly to dependency and structural assertions
Property: guideline	No	Requires semantic interpretation and threshold judgement
Executive: technology	Partially	Not all technology choices leave code footprints
Executive: process	No	Not expressed in the source code

The DSL introduced in this thesis focuses on structural existence decisions and property design-rule decisions as its primary target class. These two categories offer the greatest potential for automated enforcement. They are common in practice, their violation has measurable negative consequences (as the evidence from Section 2.3 demonstrates), and they can be expressed as precise patterns or dependency assertions that automated tools can evaluate reliably without human judgement.

2.5 Architecture Conformance Checking

Architecture conformance checking is the discipline of mechanically verifying that a system's implemented structure corresponds to its intended architecture. The foundational distinction between *intended* and *as-implemented* architecture is due to Perry and Wolf [3], who characterised software architecture as a set of elements, their form, and the rationale behind the choices. The gap between the two has historically been assessed using *reflexion models*; a technique in which a high-level architectural model is defined, inter-element dependencies are extracted from the source code, and the two views are compared to produce a map of convergences,

absences, and divergences [15]. Conformance checking tools perform this comparison at various levels of abstraction and automation. The landscape of available tooling can be organised into three categories.

The first consists of *architecture test frameworks*, which are libraries that allow developers to write conformance rules as executable tests in the same language as the application under test. Examples include ArchUnit (Java), ArchUnitNet (.NET/C#), and Arch-Go (Go). All three frameworks follow the same pattern: a fluent API allows the developer to select sets of types or packages by name or annotation and then assert dependency or inheritance constraints between them. Because the rules are expressed in the same language as the application code, they integrate naturally into the existing test infrastructure and can be executed by any standard test runner. However, this advantage is limited by the fact that the rules are closely linked to a particular programming language, and the intended meaning of each rule is hidden within a method chain that requires knowledge of the programming language to understand. Listing 2 shows how AD0001 would be expressed as an Arch-Go test.

Listing 2: AD0001 expressed as an Arch-Go test.

```
func TestAD0001(t *testing.T) {
    rules := config.Config{
        DependenciesRules: []*config.DependenciesRule{{
            Package: "*.domain..",
            ShouldNotDependsOn: &config.Dependencies{
                Internal: []string{
                    "*.application..",
                    "*.adapter..",
                    "*.infrastructure..",
                },
            },
        }},
    }
    result := archgo.CheckArchitecture(config.Load("github.com/adr/ad-
    guidance-tool"), rules)
    if !result.Pass { t.Fatal(result) }
}
```

The second category consists of *dedicated architecture analysis tools* that introduce their own rule specification format. Sonargraph [9] defines an architecture DSL in which the developer declares named *artifacts* (matched by include patterns) and then specifies which artifacts may communicate with which others. Violations are reported continuously, visualised as dependency maps, and can be configured to fail automated builds. The READ tool [16] for example, uses a DSL implemented in the RASCAL meta-programming language to describe the intended layered architecture of embedded C/C++ systems and checks conformance against the actual CMake build structure. Both tools demonstrate that a dedicated language can express architectural intent more concisely and more readably than a testing framework API, but both are also strongly coupled to a specific technology context.

The third category consists of *declarative approaches* that attempt to decouple rule specification from any particular execution engine. Caracciolo et al. present Dictō [17], a lightweight textual DSL whose rules can be interpreted by a plugin-based framework that delegates the actual checking to registered third-party tools. The companion paper [15] provides a comprehensive evaluation demonstrating that such an approach improves adoption because stakeholders unfamiliar with any specific testing framework can read and validate the rules independently of the execution infrastructure.

Despite the variety of tools available, all three categories share a critical limitation: they are entirely disconnected from the ADRs in which the decisions they enforce were originally documented. A developer encountering a failing architecture test cannot determine, from that test alone, which architectural decision the test enforces or why that decision was made. The rules must be written and maintained separately from the documentation, and as teams change and decisions evolve, the rules tend to lag behind or silently contradict the current intent.

2.6 Domain-Specific Languages for Architecture Rules

A domain-specific language (DSL) is, in Martin Fowler's formulation, "a computer language that's targeted to a particular kind of problem, rather than a general-purpose language that's aimed at any kind of problem" [18]. The architectural rule domain is a compelling candidate for DSL treatment: the vocabulary is small (components, dependencies, constraints, assertions), the statements are highly declarative, and the audience extends beyond software engineers to architects, tech leads, and quality managers who need to read and understand what a rule says without a programming-language background.

Fowler distinguishes two broad approaches to DSL construction:

- An *internal DSL* reuses the syntax of an existing host language, exploiting its grammar and tooling to present rules in a readable but host-language-native form; the fluent APIs of ArchUnit and Arch-Go are internal DSLs of this kind.
- An *external DSL* defines its own grammar, independent of any host language, which is then processed by a dedicated toolchain. External DSLs require more upfront investment—a grammar, a parser, and an intermediate representation—but the investment is justified when the language must be consumed by multiple plugins written in different programming languages, when the intended readership includes non-programmers, or when the language must remain stable across technology generations.

The DSL in this thesis is an external DSL for exactly these reasons. The same file must be processable by plugins written in Go, C#, Java, or any future language, and the rules must remain readable and valid after the target codebase has migrated to a different platform. Parser generators such as ANTLR (ANother Tool for Language Recognition) [19] make external DSL construction practical. A single grammar specification drives code generation for multiple target languages, so the same rule syntax can be parsed independently of the backend ecosystem. ANTLR's context-free grammar formalism combined with its visitor interfaces provides a direct path from parse tree to an application-level semantic model without the need to handwrite a lexer or parser.

Applied to AD0001, an external DSL might express the constraint as follows:

```
Domain must not depend on Application, Adapter, Infrastructure
```

The exact syntax used by ADE is introduced in Section 4.4; what matters here is only that the rule is independent of any target language and can therefore be processed by plugins targeting Go, .NET, Java, or any future ecosystem. An external DSL that targets multiple frameworks also benefits from a stable intermediate representation (IR) that separates the parsed rule semantics from any particular execution engine. When the parsed output is serialised into a typed, schema-validated format before being handed to a plugin, the producer and every consumer share a common contract without needing to re-parse the original source

syntax. Schema-validated serialisation formats such as Protocol Buffers (Protobuf)[15] are well suited to this role because they generate decoder code for multiple programming languages and support backwards-compatible schema evolution, so new rule constructs can be added without breaking existing consumers.

The six topics covered in this section collectively define the problem space addressed by ADE. Architectural decisions, captured in ADRs as first-class artefacts, represent a system's intended architecture; drift and erosion gradually undo this architecture, driven by technical and non-technical forces, and measurably increasing the cost of maintenance and change. The taxonomy analysis establishes which decision categories can be mechanically enforced—primarily structural existence decisions and property design-rule decisions—while the conformance-checking analysis maps the current state of tooling and highlights its shared limitation of operating independently of the ADRs that motivated the rules. The DSL discussion motivates the design choice of an external grammar and a Protobuf IR over building on top of an existing testing-framework API. Together, these six topics lead to the design and implementation outlined in Section 4: a DSL containing architectural rules that link back to ADRs, a parser with semantic validation, and a plugin-based execution layer, packaged as the standalone ADE tool and additionally embeddable as a library so that the existing ADG tool can expose enforcement as a subcommand of its existing ADR workflow.

[15] `https://protobuf.dev`

3 Related Work

This section reviews the most relevant prior work and positions the contribution of this thesis within the existing literature. Whereas Section 2 established the conceptual foundations, such as what architectural decisions are, how they degrade, and what kinds of conformance-checking mechanisms exist, this section focuses on what has been built and where the remaining gaps are. The review is organised into five areas. Section 3.1 discusses the reusable architectural decision model approach that first articulated the enforcement step targeted by this thesis. Section 3.2 covers approaches that define explicit mappings from decisions to checkable constraints. Section 3.3 analyses conformance tools based on domain-specific languages (DSL) that inspire this thesis's DSL, but which lack architectural decision record (ADR) integration. Section 3.4 examines the architecture test frameworks that serve as execution backends for the plugin system. Section 3.5 reviews empirical studies on how enforcement is practised in industry. Finally, Section 3.6 summarises the findings into a gap analysis that motivates the design presented in Section 4.

3.1 Reusable Architectural Decision Models

Zimmermann et al. [6] propose a meta-model for reusable architectural decision models that structures the management of decisions into three phases: identification, making, and enforcement. Their enforcement mechanism is called *decision injection*. The documented decisions, stored as instances of the meta-model, are fed as parameters into model transformations, code generators, and configuration-management tools so that compliance is achieved structurally rather than verified manually. The approach is demonstrated in the context of model-driven enterprise application development, where decisions about middleware choices, transaction policies, and layering rules are propagated into architectural templates and code skeletons. However, the contribution is conceptual rather than tool-oriented; the paper defines the lifecycle and the role of enforcement within it but does not produce a general-purpose enforcement tool. Their decision injection targets model transformations in a Java-centric MDA environment, not a reusable, language-agnostic mechanism. This thesis can be understood as a concrete realisation of the enforcement idea they articulated, generalised to operate on source-code artefacts rather than transformation templates and implemented as a plugin-based tool that delegates the actual checking to existing framework-specific plugins.

Schröder et al. [20] investigated the industrial practice of architecture enforcement through interviews with twelve experienced software architects. Their findings confirm the relevance of the enforcement phase: architects consistently identified dependency rules, design patterns, naming conventions, and technology constraints as their primary enforcement concerns. Crucially, the study also found that enforcement activities remain overwhelmingly manual, like with code reviews, pair programming, and informal discussion, and that architects desire "more advanced tool support for enforcement". This thesis targets precisely this gap by automating the checks that architects currently perform by hand and tying them back to the decisions they serve.

3.2 Decision-to-Constraint Mapping

A recurring theme in the decision management literature is the need for a formal mapping between documented decisions and the artefacts they constrain. Lytra et al. [21] propose a constraint-based approach that formally maps architectural design decisions onto a generic component model. This approach automatically generates OCL (Object Constraint Language) constraints from the mapping and validates the component model whenever decisions change. Their tool regenerates constraints after each decision modification and highlights inconsistencies in the component diagram.

Ton That et al. [22] pursue a similar goal through a different formalism. They treat architectural patterns as first-class representations of architectural decisions. Once a pattern is integrated into an architecture model, a set of views and OCL invariants can be automatically generated that express the structural properties the pattern imposes. Conformance checking then reduces to verifying the invariants against the current architecture model. The approach is demonstrated on service-oriented architectures.

Both approaches share two important characteristics with this thesis. First, they introduce an explicit mapping layer between the world of decisions and the world of checkable constraints; in this thesis, the DSL and the intermediate representation (IR) serve an analogous structural role. Second, they regenerate constraints when the decision model changes, ensuring that enforcement remains synchronised with intent. However, both approaches operate at the model level, they target component diagrams and architecture descriptions rather than source code. ADE extends the idea to code-level checking, where the "constraints" are dependency assertions, structural tests, and file-system expectations evaluated by concrete testing tools.

Capilla et al. [23] examine the evolution dimension more broadly, proposing a model for documenting how decisions change over time and how such changes propagate through the architecture. While their work does not produce an enforcement tool, it reinforces the argument that the link between decisions and their downstream artefacts must be maintained dynamically, not established once and forgotten.

3.3 DSL-Based Architecture Conformance

Several projects have proposed domain-specific languages for expressing architecture rules in a form that is both human-readable and tool-agnostic. The closest predecessor to the DSL in this thesis is Dictō, developed by Caracciolo et al. [17]. Dictō provides a lightweight declarative language in which rules such as

```
org.app.Model cannot-depend org.app.View
```

can be written once and verified by multiple third-party tools through a plugin adapter framework. The companion paper [15] presents an evaluation showing that the unified approach significantly reduces the specification effort compared to maintaining tool-specific configurations independently.

Dictō and this thesis's DSL are based on the idea that architectural rules should be specified once in a tool-agnostic notation and then executed by delegating to specialised plugins, but they differ on three important axes. First, Dictō has no concept of a governing decision: rules exist as free-standing specifications without explicit traceability to the ADR (or any other documentation artefact) that motivated them. This thesis makes the ADR the organisational unit by requiring every file to begin with an `adr` header that binds it to a specific decision. Second, Dictō relies on an internal adapter mechanism within a single monolithic tool, whereas ADE uses an out-of-process plugin protocol (stdin/stdout, Protobuf IR) that allows plugins to be developed, versioned, and distributed independently. Third, Dictō focuses primarily on dependency rules, whereas the rule file DSL distinguishes `code`, `file`, and `custom` rule blocks to cover structural dependencies, filesystem layout, and arbitrary domain-specific assertions.

The READ tool [16] is another DSL-based conformance checker, developed as a master's thesis in cooperation with Philips. It defines a RASCAL-based DSL tailored to the layered architecture of embedded positioning software and checks

conformance against the CMake build model. READ demonstrates that a DSL-driven approach can achieve precise detection of disallowed layer crossings and invalid interface usage in an industrial setting. However, it is strongly coupled to a single technology context (C/C++, CMake) and does not support extension to other languages or ecosystems.

Sonargraph [9] introduces an architecture DSL built around the concept of named *artifacts* matched by include patterns, with explicit "`connect to`" statements defining allowed communication paths. The tool supports Java, C#, C/C++, and Python, and integrates into automated builds to fail when architectural constraints are violated. Sonargraph represents the state of practice in commercial architecture conformance tooling; its limitation in the context of this thesis is that its DSL is proprietary, its execution is not extensible via plugins, and its rules exist independently of any decision documentation.

Bucaioni et al. [24] introduce the concept of *Architecture as Code* (AaC) whereby architectural descriptions are treated as machine-readable, version-controlled artefacts integrated with the implementation. This concept is analogous to Infrastructure as Code in DevOps. Their literature review, which considers multiple perspectives, identifies consistency between description and implementation as the central unsolved challenge. It also argues that AaC approaches should enable automation, continuous validation and seamless integration with CI/CD pipelines. This thesis aligns with the AaC approach, in which rules written in the DSL are treated as code and version-controlled alongside the source. They are also automatically evaluated in the pipeline.

3.4 Architecture Test Frameworks

Architecture testing frameworks, like *ArchUnit* for Java, *NetArchTest* for .NET, and *Arch-Go* for Go, provide the execution environments that the ADE reference plugins target. All three follow the same conceptual pattern: types or packages are selected by naming conventions, annotations, or glob patterns, and dependency or inheritance assertions are expressed through a fluent API. The resulting tests run inside the standard test runner of the respective ecosystem and produce pass/fail results that integrate naturally with CI pipelines.

These frameworks are powerful, yet limited in two ways. Firstly, they are language-specific, meaning that a multi-language project must maintain separate rule sets in different testing frameworks with no shared specification. Secondly, the rules

are expressed as executable code in the same language as the application. This confuses specification with implementation, obscuring the conceptual intent of a rule from anyone unfamiliar with the API. ADE does not treat these frameworks as alternatives, but rather as plugins.

3.5 Empirical Evidence on Enforcement Practice

Minsky [25] argues from a software-engineering principles perspective that architectural constraints should be enforced because developers, working under schedule pressure and with limited global understanding of the system, will inevitably introduce violations unless the constraints are mechanically imposed. The paper proposes a runtime enforcement mechanism for distributed object systems; while the mechanism itself (interceptor-based policy checking) is specific to its target domain, the motivation applies broadly and supports the case for build-time enforcement as pursued in this thesis.

LaToza et al. [10] provide empirical evidence supporting Minsky's argument. Their interview study at two software companies found that code reviews are the primary enforcement mechanism, that reviews fail to catch architectural violations systematically, and that costly rework occurs when a technology decision's limitations are discovered only through production experience. The study identifies a clear need for "automated, reproducible enforcement", which is what this thesis is aiming to provide.

The systematic mapping study by Li et al. [5] further reinforces this motivation by cataloguing the consequences of failing to enforce decisions, such as unwanted dependencies, accumulated technical debt, reduced testability and increased change cost. Their finding that non-technical causes, such as schedule pressure and staff turnover, contribute equally to erosion strengthens the case for proactive, tool-based enforcement that does not depend on individual awareness.

3.6 Gap Analysis

The preceding subsections reviewed related work across five areas: reusable decision models (Section 3.1), decision-to-constraint mapping (Section 3.2), DSL-based conformance (Section 3.3), architecture test frameworks (Section 3.4), and empirical enforcement practice (Section 3.5). The comparison dimensions in Table 2

were derived from the recurring concerns surfaced across these areas. *ADR-linked* captures whether a tool anchors its rules in a documented decision, which both the decision-model literature (Sections 3.1 and 3.2) and the empirical study by Schröder et al. [20] identify as a missing link. *Language-agnostic* captures whether the rule specification is independent of any target programming language, which Dictō (Section 3.3) demonstrates is achievable but not the norm among test frameworks (Section 3.4). *Plugin-extensible* captures whether new plugins can be added without modifying the core tool. *Code-level* captures whether the tool operates on actual source artefacts rather than on abstract models. *CI-integrated* captures whether the tool produces artefacts that integrate into standard build pipelines without wrapper scripts, which the expert study (Section 3.5) identifies as a practical prerequisite for adoption. These five dimensions are not the only ones that could be chosen; a full multi-criteria evaluation would consider additional axes. Three such dimensions are worth naming for transparency:

- *Formal verification support.* Some tools (e.g. model checkers) can prove the absence of violations; static analysers, including ADE, can only detect violations in the artefacts they observe. None of the approaches reviewed offers formal proof.
- *Runtime or behavioural analysis.* Conformance checking at the source-code level cannot detect violations that only manifest at runtime (e.g. service-mesh traffic patterns, dynamic class loading). This limitation applies to all approaches compared here, including ADE.
- *Quantitative metric-based detection.* Sonargraph and similar tools can report on architectural smells as measurable densities (e.g. coupling scores, cycle counts). ADE produces binary pass/fail results per rule rather than continuous measurements, and does not compute aggregate metrics.

The five dimensions in Table 2 represent the subset most directly motivated by the problem addressed in this thesis: bridging ADR documentation and executable conformance checks in a language-agnostic, extensible way.

Table 2: Comparison of related approaches along five dimensions motivated by the reviewed literature.

Approach	***ADR-linked***	***Language-agnostic***	***Plugin-extensible***	***Code-level***	***CI-integrated***
Zimmermann (RADM)	–	✓	–	–	–
Lytra (constraint)	–	✓	–	–	–
Ton That (patterns)	–	–	–	–	–
Dictō	–	✓	✓	✓	✓
READ	–	–	–	✓	✓
Sonargraph	–	✓	–	✓	✓
ArchUnit / Arch-Go / NAT	–	–	–	✓	✓
ADE (this thesis)	✓	✓	✓	✓	✓

The table reveals a consistent pattern along the first dimension. Every approach that operates at the decision or model level (Zimmermann, Lytra, Ton That) is conceptually language-agnostic but does not reach down to source-code checking. Every approach that operates at the code level (Dictō, READ, Sonargraph, the test frameworks) provides actionable enforcement but is disconnected from the architectural decisions that motivated the rules. No prior system ties an ADR-centric documentation workflow to a generic, plugin-based enforcement backend that delegates to existing code-level tools.

ADE sits at the intersection of these two groups and is the only approach in the table that is simultaneously ADR-linked, language-agnostic, plugin-extensible, code-level, and CI-integrated. However, it does not offer formal verification, runtime analysis, or metric-based detection, and the three omitted dimensions above are where future work can extend the approach. The design choices that realise the five checked dimensions are presented in Section 4.

4 Design and Implementation

This section presents the design and implementation of ADE, the Architectural Decision Enforcement tool that translates documented architectural decisions into automated conformance checks. ADE is shipped both as a standalone command-line binary[16] and as a Go library that can be embedded into other tools; the Architectural Decision Guidance (ADG) tool[17] uses this integration path to expose ADE as the "`adg enforce`" subcommand, continuing the workflow established in the preceding project thesis [8]. The design follows from the gaps identified in Section 3. Section 4.1 first introduces the user roles whose needs shape the tool's functional scope and their corresponding use cases. Section 4.2 then states the functional and non-functional requirements that those roles imply. Section 4.3 describes the end-to-end workflow from rule file to conformance report, establishing the conceptual workflow before its parts are described individually. Section 4.4 specifies the domain-specific language (DSL) in which rules are expressed, followed by Section 4.5, which presents the intermediate representation (IR) that decouples the DSL from any particular technology that was used to implement a plugin. Section 4.6 defines the plugin model through which external tools are integrated. Section 4.7 documents the command-line interface. Finally, Section 4.8 covers the integration points between ADE and the ADG tool and the continuous integration setup.

4.1 User Roles and Use Cases

The functional scope of ADE is defined by four representative user roles that are common to teams that adopt architectural decision enforcement. These roles are not mutually exclusive; in a small team the same person may act as both architect and developer, and the plugin author may be either an external contributor or an internal platform engineer. Each role is introduced below with a short description and a list of the use cases it covers. The commands that realise these use cases are described in detail in the subsections that follow.

Software Architect. The architect documents architectural decisions as ADRs and is responsible for keeping the documentation in sync with the actual system. With ADE, the architect extends this responsibility to authoring rule files that

[16] https://github.com/phi42/ad-enforcement-tool
[17] https://github.com/adr/ad-guidance-tool

formally express enforceable decisions and to managing the plugins that execute those rules.

- Author a rule file co-located with the corresponding ADR, declaring selectors and assertions that encode the structural intent of the decision.
- Generate a rule file template from an existing ADR to reduce initial boilerplate.
- Validate a rule file for syntactic and semantic correctness before committing it.
- Install and manage the plugins required for rule compilation and verification.

Developer. The developer writes and modifies the application code that the rule files constrain. When a check fails locally or in CI, the developer reads the violation report to determine which ADR the failing rule belongs to, opens the referenced ADR for context, and either adjusts the code to comply or proposes an exception that is recorded back in the rule file.

- Run conformance checks against the codebase to detect architectural violations.
- Execute generated test artefacts via the project's standard test runner.
- Read violation reports and trace each failing check back to its originating ADR.
- Revise the implementation or record a documented exception in response to failures.

Plugin Author. The plugin author extends ADE to a new language, framework, or check type without modifying the tool itself. The only coupling to ADE is the published intermediate representation schema, which can be copied into the plugin repository and used to generate type-safe bindings in any language for which a protobuf code generator exists.

- Consume the published intermediate representation schema to understand the wire format.

- Implement a plugin binary that reads the IR and either generates test artefacts or evaluates assertions directly.
- Test the plugin in isolation against synthetic IR payloads without running ADE.
- Publish the plugin binary so it can be discovered and installed by architects and pipeline engineers.

DevOps Engineer. The pipeline engineer integrates ADE into the project's CI workflow and is responsible for reproducibility and build configurations. This role decides the enforcement strategy (generated tests versus direct verification), pins tool and plugin versions, and ensures that conformance failures propagate correctly to the build result.

- Configure enforcement defaults to avoid repeating common flags across every invocation.
- Install and pin plugin versions to guarantee reproducible CI runs.
- Integrate rule compilation into the CI pipeline to generate test artefacts as part of the build.
- Integrate direct conformance verification into the CI pipeline and propagate exit codes to fail the build on violations.

These four roles correspond directly to the functional and non-functional requirements that follow: the architect drives the DSL design; the developer drives the traceability and reporting requirements; the plugin author drives the IR and protocol design; the pipeline engineer drives the CLI design, the CI integration, and the configuration model.

4.2 Requirements

The requirements for ADE can be grouped into functional and non-functional categories:

Functional Requirements

1. **Parse and validate rule files.** ADE reads rule files written in the dedicated DSL, detects syntactic and semantic errors (missing selectors, undefined references, type mismatches), and reports them with file and line information.

2. **Emit a typed intermediate representation.** After successful parsing, ADE produces a language-neutral, schema-validated IR that encodes the ADR metadata, selectors, and rules in a structured binary format.

3. **Discover and invoke plugins.** ADE locates installed plugins by name, verifies that a plugin supports the requested invocation mode, serialises the IR to the plugin's standard input, and relays the plugin's output and exit code.

4. **Maintain ADR traceability.** Every rule file begins with a mandatory ADR header. The resulting IR carries the decision identifier and title, so that any conformance violation can be traced back to its originating decision.

5. **Integrate with CI pipelines.** The exit code of ADE reflects the overall conformance result: zero indicates that all rules pass, non-zero indicates one or more violations. This convention enables direct use in continuous integration without additional wrapper scripts.

6. **Manage plugins.** ADE provides commands to install, update, uninstall, and list plugins. Installation supports both local binaries and remote GitHub releases.

7. **Provide persistent configuration.** Default values for frequently used flags (plugin name, output directory) can be stored in a hierarchical configuration that combines a global user-level file with a project-local override.

8. **Embeddability.** ADE is additionally packaged as a Go library so that other tools, including the ADG tool, can register its commands as part of their own command tree without forking the codebase.

Non-Functional Requirements

The non-functional requirements are partially organised along the FURPS+ categories [26], consistent with the quality model used in the preceding project thesis [8].

1. **Usability.** The DSL reads as near-natural-language English. Error messages indicate the exact file, line, and nature of the problem. Every command supports `--help`.
2. **Reliability.** Compilation is deterministic: given the same rule file and plugin version, the output is identical. Plugin failures are isolated and do not corrupt the tool's state.
3. **Performance.** The full pipeline from parsing to plugin invocation completes within seconds for typical project sizes, enabling execution on every commit in a CI environment.
4. **Supportability.** The design is independent of any specific programming language, framework, or decision format. New rule kinds can be added to the grammar and)IR without breaking existing rule files. New plugins can be developed and distributed independently of the ADE tool.

4.3 Conceptual Workflow

The enforcement pipeline of ADE operates as a linear sequence. Figure 3 illustrates the data flow inside a single `ade compile` or `ade verify` invocation, and shows the link to the ADR that motivated the rule file.

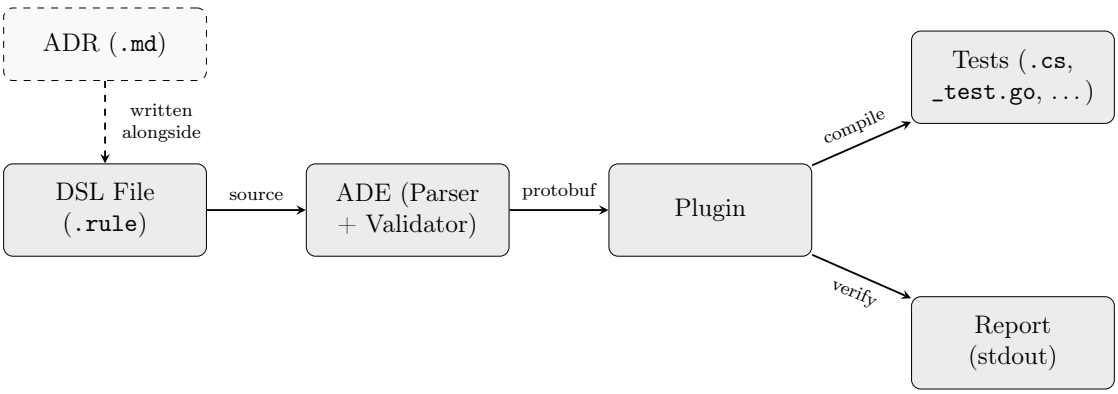

Figure 3: Enforcement pipeline: from rule file to conformance output.

The following paragraphs walk through each stage in turn, tracing AD0001 as a running example from ADR to generated test file. To keep the example in this section short, only one code rule is illustrated. The full rule file and generated test are shown in the following sections describing the stages in more detail. The complete rule file and test are shown in Listing 6 and Listing 9 respectively.

The first stage is the specification of the architectural constraint. The architect writes a rule file (`.rule`) alongside the corresponding ADR markdown (`.md`). The file references the decision by identifier and title, declares named selectors that map logical architecture elements to implementation-level patterns, and expresses one or more assertions using those selectors. Listing 3 shows the decision outcome of AD0001 that motivates this rule file (see Listing 1 for the full ADR). Listing 4 shows the rule file written alongside it; the header references the ADR by identifier and title; the `component` selectors map logical names to package patterns; the `code` block expresses the dependency prohibition.

Listing 3: Decision outcome of AD0001, which motivates the co-located rule file.

```
Outcome

We decided for Option 1 because: The domain layer encodes business rules that
    must remain stable regardless of how the tool is delivered or how data is
    stored. Keeping domain independent ensures that use-case logic and
    infrastructure can change without affecting the core model.
```

Listing 4: Rule file co-located with AD0001.

```
adr "0001" "Domain layer is independent from upper layers"

component "Domain"      = "github.com/adr/ad-guidance-tool/internal/domain.."
component "Application" = "github.com/adr/ad-guidance-tool/internal/application
    .."

code "domain_does_not_import_application" {
  Domain must not depend on Application
  severity error
}
```

In the second stage, ADE parses the rule file. An ANTLR4-generated parser [19] processes the input according to the formal grammar, and a visitor pass performs semantic validation. It checks that every referenced selector has been declared, that file-level assertions appear only in `file` blocks, that code-level assertions appear only in `code` blocks, and that names do not collide with reserved keywords. One implementation challenge arises from `custom` blocks, whose bodies are free-form text that cannot be tokenised by the grammar. A pre-processing step extracts these blocks before lexing, preserving line numbers for error reporting, and appends the raw bodies to the IR after the visitor pass completes.

The third stage transforms the validated abstract syntax tree into a protobuf-encoded IR. The IR is a self-contained message that carries the ADR header, the selector list, and the fully resolved rule list including severity annotations and exclusion clauses. This message constitutes the sole interface between ADE and any plugin.

In the fourth stage, the plugin receives the serialised IR on standard input and performs one of two actions depending on its advertised mode. A *compile-mode* plugin generates test source files in the target language; Listing 5 shows the Go test file that the *archgo* plugin produces from the AD0001 rule file. The generated test shares the same ADR identifier as the rule file and integrates with Go's standard `go test` toolchain without additional configuration. A *verify-mode* plugin evaluates the assertions directly against the target codebase and prints structured pass/fail results to standard output. In both cases, the plugin's exit code indicates whether the process was successful or not. A plugin can support both modes; in this case, the plugin depends on the command used by the user (compile or verify). If a user tries to use an unsupported mode, an error is reported.

Listing 5: Go test file of AD0001 generated by `ade compile` using *archgo* plugin.

```
func TestArchitectureFrom_0001(t *testing.T) {
  configuration := config.Config{
    DependenciesRules: []*config.DependenciesRule{
      {
        Package: "github.com/adr/ad-guidance-tool/internal/domain..",
        ShouldNotDependsOn: &config.Dependencies{
          Internal: []string{
            "github.com/adr/ad-guidance-tool/internal/application..",
          },
        },
      },
    },
  }

  moduleInfo := config.Load("github.com/adr/ad-guidance-tool")

  result := archgo.CheckArchitecture(moduleInfo, configuration)
  if !result.Pass {
    t.Fatalf("Architecture rules from ADR %q (%q) are violated",
      "0001", "Domain layer is independent from upper layers")
  }
}
```

The strict linearity of the pipeline keeps each stage independently testable. The parser can be validated in isolation using synthetic inputs. The IR can be inspected using standard Protobuf tooling. Plugins can be developed and tested against hand-crafted IR payloads, without the need of running ADE.

4.4 Domain-Specific Language

The rule file DSL is the primary user-facing artefact of ADE. Its design is guided by three principles: (1) readability ensures that rule files serve as documentation of the enforced constraint; (2) the ADR link ensures that every rule traces back to a documented decision; (3) declarativity keeps the specification independent of how the check is ultimately executed.

The scope of the DSL and which decision types it can express was derived from the taxonomy of Keim et al. [14] reviewed in Section 2.4. That taxonomy categories decisions into *existence*, *property*, and *executive* types, and further subdivides existence decisions into structural, arrangement, and behavioural categories. The analysis of which categories are practically enforceable through static analysis di-

rectly shaped the two rule block types. Structural existence decisions that govern the internal component structure and inter-component dependencies of the codebase are expressed through `code` blocks, which capture dependency prohibitions, inheritance and interface constraints, visibility rules, and namespace containment assertions. Arrangement existence decisions that concern how components are grouped into layers or modules also map to `code` blocks, since layer ordering and module boundaries both reduce to namespace-level dependency assertions. Structural existence decisions that concern the presence or organisation of artefacts in the project filesystem are expressed through `file` blocks, which assert the existence, absence, or content of files and directories.

Behavioural existence decisions are outside the scope of the DSL because they concern runtime interaction protocols, event sequences, and operational contracts that cannot be verified from static artefacts. Among *property decisions*, design-rule decisions that prescribe structural patterns, naming conventions, or required dependency directions often reduce to the same namespace-level assertions already captured by `code` blocks and are therefore within scope. Property guidelines and quality-attribute decisions, however, require semantic interpretation that goes beyond what static analysis can evaluate; a rule such as "aggregate roots must enforce invariants in their setters" has no structural manifestation a namespace-oriented tool can resolve. *Executive decisions* (technology choices, process conventions) are organisational in nature and lack a verifiable code-level structure. Behavioural existence decisions, property guidelines and quality-attribute decisions, and executive decisions account for the non-enforceable ADRs identified in the case study of Section 5.

The following subsections describe the DSL's file structure, selector declarations, rule blocks, assertion syntax, and grammar implementation. The DSL is written in a `.rule` file.

File Structure

Every rule file begins with a mandatory header that binds the file to a specific ADR:

```
adr "<id>" "<title>"
```

The header is followed by zero or more *selector declarations* and one or more *rule blocks*. Selectors define reusable names for architecture elements; rule blocks contain the assertions that reference those selectors.

Selectors

A selector maps a logical name to an implementation-level pattern:

```
component "Domain"         =  "MyApp.Domain"
component "Application"    =  "MyApp.Application"
class     "AggregateRoot"  =  "*.AggregateRoot"
interface "IRepository"    =  "*.IRepository"
path      "License"        =  "LICENSE"
```

Four selector kinds are supported. A `component` matches a namespace, package, or module. A `class` matches a type name. An `interface` matches an interface name. A `path` matches a filesystem path or glob pattern. Selector names must begin with an uppercase letter and must not collide with any reserved keyword of the grammar.

Rule Blocks

Assertions are grouped into typed blocks that determine their domain. A `code` block contains code-structural assertions: dependency, inheritance, naming, visibility. A `file` block contains filesystem assertions: existence, content. A `custom` block contains free-form text that is forwarded to a plugin without interpretation by the tool.

```
code "domain-isolation" {
  Domain must not depend on Application, Infrastructure
  severity error
}

file "license-exists" {
  License must exist
  severity warning
}

custom "custom-block" {
  This is a custom block with free-form content that ADE does not parse.
    It can be used by plugins to define their own assertion vocabularies
    beyond what the standard grammar supports.
}
```

Each block carries a unique name (used in violation reports), one or more assertion statements, optional exclusion clauses, and an optional severity annotation that defaults to `error` if omitted. A `custom` block provides an escape option for plugin-defined rule vocabularies that go beyond the standard grammar. The parser recognises only the block header and the matching closing brace; the body is stored verbatim in the IR and forwarded to the plugin for interpretation. This mechanism allows plugin authors to extend the assertion vocabulary without requiring changes to the grammar or a new release of ADE.

Listing 6 shows the complete rule file for AD0001 (introduced in Section 2.2). The file declares four `component` selectors (one per Clean Architecture layer) and three `code` rules, each forbidding a dependency from the domain layer into one of the upper layers. For visualization purposes, the selector names are shortened with a placeholder, `<repository>`, which would be `github.com/adr/ad-guidance-tool` in the actual file. Besides that, the file is taken verbatim from the ADG repository[18].

[18] https://github.com/adr/ad-guidance-tool/blob/main/docs/adr/AD0001-domain-layer-is-independent-from-upper-layers.rule

Listing 6: Complete rule file for AD0001.

```
adr "0001" "Domain layer is independent from upper layers"

component "Domain"         = "<repository>/internal/domain.."
component "Application"    = "<repository>/internal/application.."
component "Adapter"        = "<repository>/internal/adapter.."
component "Infrastructure" = "<repository>/internal/infrastructure.."

code "domain_does_not_import_application" {
  Domain must not depend on Application
  severity error
}

code "domain_does_not_import_adapter" {
  Domain must not depend on Adapter
  severity error
}

code "domain_does_not_import_infrastructure" {
  Domain must not depend on Infrastructure
  severity error
}
```

Assertion Syntax

Assertions follow a subject–modality–verb pattern that reads as near-natural English. The subject identifies the element under test (either a named selector or an inline pattern). The modality is one of `must`, `must not`, or `must only`. The verb phrase specifies the constraint type. Table 3 lists the assertion families supported by the grammar.

Table 3: Assertion families in the rule DSL.

Family	**Example**	**Block**
Dependency	`A must not depend on B`	code
Access	`A must only be accessed by B`	code
Acyclicity	`A must be acyclic`	code
Inheritance	`A must extend class B`	code
Interface	`A must implement interface B`	code
Annotation	`A must be annotated with "X"`	code
Naming	`A must match "regex:..."`	code
Location	`A must be in B`	code
Visibility	`A must be public`	code
Type constr.	`A must be abstract`	code
Existence	`A must exist`	file
Content	`A must contain "..."`	file

Subjects support six forms of varying specificity: a bare selector name, an inline literal, an inline regex match, a scoped-all form, a scoped literal, and a scoped regex. The following examples illustrate each form:

```
Domain                              # bare selector name
class "Handler"                     # inline literal
class match "regex:.*Handler"       # inline regex match
class in Domain                     # scoped-all form
class "Handler" in Domain           # scoped literal
class match "regex:..." in Domain   # scoped regex
```

The scoping mechanism uses the `in` keyword to restrict the assertion's subject to elements that reside within a given component.

Grammar Implementation

The grammar is specified as an ANTLR 4 combined grammar that defines both the lexer and parser rules in a single file [19]. ANTLR generates a Go lexer, parser, and visitor interface from the grammar. The tool traverses the parse tree with a visitor that constructs the domain-level semantic model and emits the IR. Listing 7 shows the top-level parser rules of the `.g4` file[19].

Listing 7: Top-level parser rules of the grammar.

```
file
  : adrDecl (selectorDecl | ruleDecl)* EOF
  ;

adrDecl
  : ADR STRING STRING
  ;

selectorDecl
  : (COMPONENT | CLASS | INTERFACE | PATH) STRING EQ STRING
  ;

ruleDecl
  : ruleType STRING LBRACE ruleStmt* RBRACE
  ;

ruleType
  : FILE
  | CODE
  ;
```

The grammar treats certain English words (`on`, `be`, `with`, `by`) as optional filler tokens so that assertions can be written in either their minimal or their expanded form: `must depend Application` and `must depend on Application` are both accepted. Comments begin with `#` and extend to the end of the line.

[19] https://github.com/phi42/ad-enforcement-tool/blob/main/internal/parser/ADE.g4

Syntax Highlighting

To improve the authoring experience, a Visual Studio Code syntax highlighting extension[20] for the rule file DSL is distributed alongside ADE. The extension is packaged as a `.vsix` file and is available from the releases page of the ADE repository[21]. It provides token-based colouring for keywords, selector declarations, assertion verbs, string literals, and comments, as shown in Figure 4.

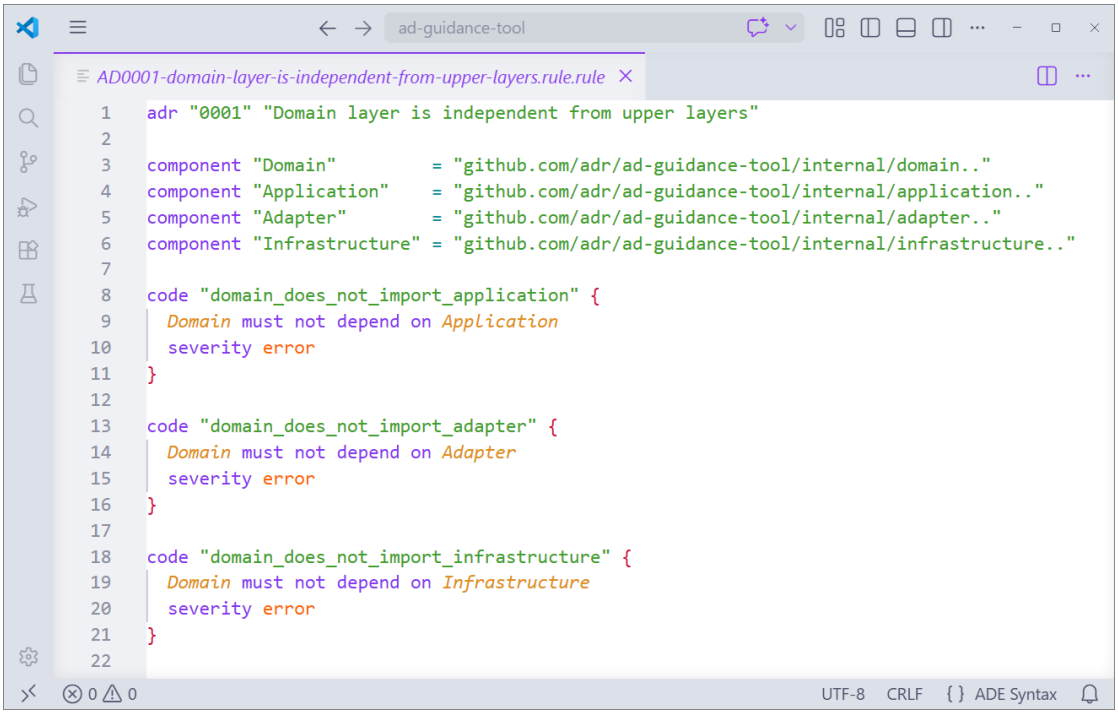

Figure 4: VS Code syntax highlighting for the rule DSL.

[20] https://github.com/phi42/ad-enforcement-tool/tree/main/extras/vscode
[21] https://github.com/phi42/ad-enforcement-tool/releases

4.5 Intermediate Representation

The IR is the single interface between ADE and its plugins. Its design addresses three concerns: (1) decoupling means that plugins do not depend on the parser or the DSL syntax; (2) cross-language compatibility means that plugins may be written in any language for which a protobuf library exists; (3) schema evolution means that the IR can be extended with new fields without breaking existing plugins.

Figure 5 shows the domain model of the IR. The root `Spec` message aggregates one `Adr` header, a list of `Selector` declarations, a list of `Rule` entries, and a plugin configuration map; each `Rule` in turn carries a `from` field identifying the assertion subject, zero or more `targets`, and optional exclusion clauses. The four enumerations (`SelectorKind`, `RuleKind`, `Severity`, `InvocationMode`) capture the closed sets of values that selectors, rules, severities, and invocation modes may take.

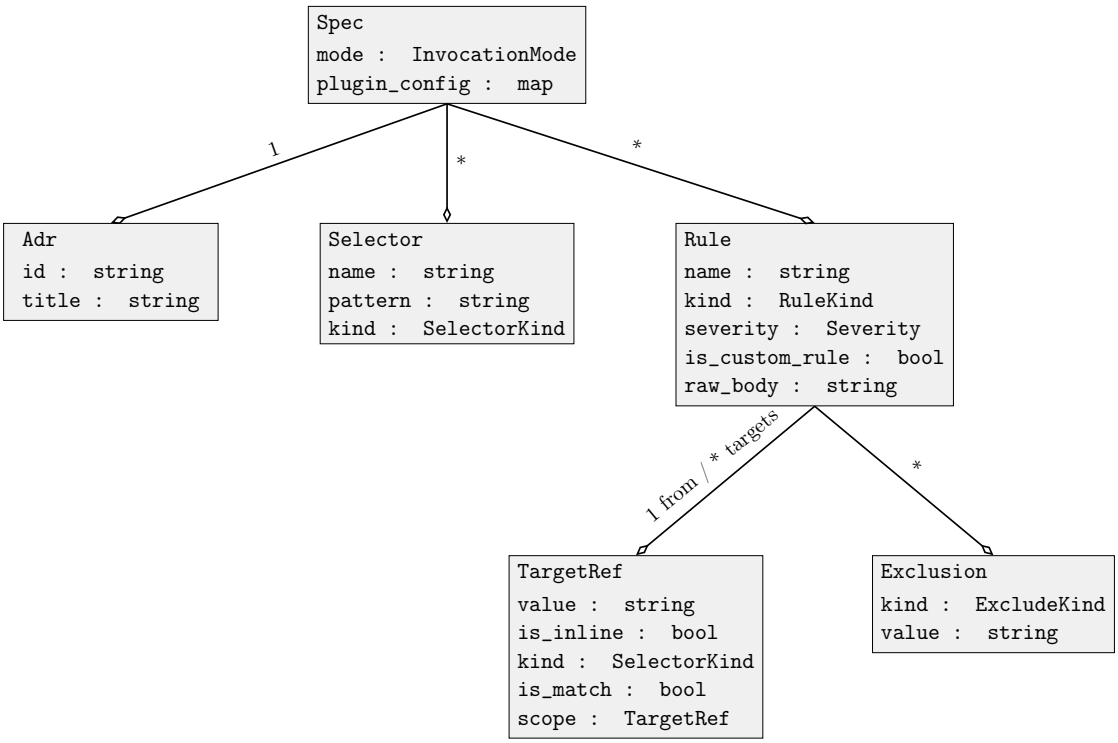

Figure 5: Domain model of the protobuf IR, field names match the proto schema.

Format Choice

Protocol Buffers were chosen over alternatives like JSON for three reasons. First, protobuf provides a schema definition language (`.proto` file) that serves as a machine-readable contract between host and plugin. Second, every field is numbered rather than named, so new fields can be added without changing existing serialised messages, and old plugins simply ignore unknown field numbers. Third, protobuf code generators exist for Go, C#, Java, Python, Rust, and other languages, so plugin authors receive type-safe data access without manual parsing.

JSON was considered as a simpler alternative. While JSON is human-readable and trivially parseable, it provides no schema enforcement at the wire level, no type safety without additional tooling (e.g. JSON Schema), and no forward-compatibility guarantee when fields are added. The performance difference is negligible at the message sizes involved (typically a few kilobytes).

Schema Design

The protobuf schema is defined in a single `.proto` file. The top-level message carries all information a plugin needs to process a single rule file:

```
message Spec {
  Adr                  adr           = 1;
  repeated Selector    selectors     = 2;
  repeated Rule        rules         = 3;
  InvocationMode       mode          = 4;
  map<string, string> plugin_config = 5;
}
```

The `Adr` message contains the decision identifier and title. Each `Selector` carries a name, a pattern string, and a kind enum (`SELECTOR_COMPONENT`, `SELECTOR_CLASS`, `SELECTOR_INTERFACE`, `SELECTOR_PATH`). Each `Rule` carries the rule name, a kind enum that encodes the assertion type (e.g. `RULE_NOT_DEPEND`, `RULE_IMPLEMENT`), a `from` field referencing the assertion subject, a list of `targets`, a severity level, and optional exclusion clauses. The `InvocationMode` enum distinguishes between `MODE_COMPILE` and `MODE_VERIFY`.

Schema Usage by Plugin Authors

Plugin developers use the same `.proto` file to generate type-safe bindings in their implementation language. ADE itself uses the generated types that are produced by the standard `protoc` compiler with the `protoc-gen-go` plugin. Since all the reference plugins are written in Go, they directly reference the generated types by importing them from the ADE Go module. If a plugin is not written in Go, the developer runs the `protoc` compiler with the appropriate language plugin to generate the corresponding types in that language. Each plugin repository then contains its own copy of the generated code, produced from the same `.proto` source as the host tool, so that host and plugins share the same wire format without requiring a shared library dependency. In the host tool itself, no separate domain model exists between the parser output and the IR. The visitor writes directly into the protobuf message structs, eliminating an additional mapping layer. Serialisation uses deterministic marshalling, and the payload is written to the plugin's standard input as a raw byte stream. The plugin reads until EOF and unmarshals the payload. This framing choice is safe because the host spawns a fresh process per invocation and closes the write end of the pipe after marshalling.

Because Protobuf's encoding is inherently forward-compatible, unknown fields are either preserved or silently ignored. This means that the schema can be extended without increasing the version number, provided that new fields are strictly additive. However, a breaking change (removing a field, changing a field's type or redefining an enum value) would require the proto package version to be incremented.

Listing 8 shows the an abbreviated version of the IR produced when the rule file from Listing 6 is parsed. The text format here is the Protobuf debug rendering; the wire format is the binary equivalent. The selector list is fully resolved, every assertion is reduced to a typed enum value (`RULE_NOT_DEPEND`), and the original ADR header is retained so that plugin-generated reports can attribute violations to the originating decision.

Listing 8: Abbreviated IR for AD0001 (Protobuf text format).

```
adr { id: "0001"
      title: "Domain layer is independent from upper layers" }

selectors { name: "Domain"
            kind: SELECTOR_COMPONENT
            pattern: "<repository>/internal/domain.." }

rules { name: "domain_does_not_import_application"
        kind: RULE_NOT_DEPEND
        from { value: "Domain" }
        targets { value: "Application" }
        severity: SEVERITY_ERROR }

mode: MODE_COMPILE
```

4.6 Plugin Model

The plugin model defines how external tools integrate with ADE. A plugin is an executable binary that reads Protobuf from the standard input and writes the results to the standard output or the filesystem. A plugin must satisfy three obligations:

1. **Info endpoint.** When invoked with the `--info` flag, the plugin prints a JSON object to standard output that declares its supported invocation modes: `{"modes":["compile"]}` or `{"modes":["verify"]}` (or both).
2. **Protobuf input.** When invoked without flags, the plugin reads a binary-encoded IR message from standard input.
3. **Output convention.** A compile-mode plugin writes generated source files to the directory indicated by the IR specification. A verify-mode plugin prints structured results to standard output and exits with code 0 if all rules pass, or non-zero otherwise.

The contract is deliberately agnostic to the plugin's implementation language. Any language with a protobuf library and the ability to read from standard input can implement a conforming plugin. The three reference plugins developed alongside

this thesis are all written in Go, but this is an implementation choice, not a constraint of the protocol.

The two supported plugin modes serve complementary use cases:

- *Compile mode* generates test source code artefacts that can either be created locally or directly as part of the CI pipeline and executed by the project's standard test runner.
- *Verify mode* executes assertions immediately and reports the results to the terminal. This approach does not produce persistent artefacts and is suited to filesystem checks, naming conventions, or any assertion that can be evaluated without a full compilation step. However, code assertions are also possible if the plugin invokes the target language's test runner or analysis tools and returns its result immediately.

At runtime, ADE validates the requested mode against the plugin's advertised capabilities before invocation by querying the `--info` endpoint. If the modes do not match, the command fails with an explanatory error message rather than sending an IR payload to a plugin that cannot handle it.

Plugins are managed through a dedicated subcommand group (`ade plugin`, also reachable as `adg enforce plugin` when invoked through the ADG integration). Installation supports two strategies: either *local* registration of a pre-built binary via `--path`, or *remote* download from a GitHub release via `--repo`. In both cases, the binary is copied to a platform-appropriate data directory and registered in the global configuration file. Remote installation uses the GitHub Releases API to identify the correct asset by matching the current operating system and architecture against asset filenames. The `GITHUB_TOKEN` environment variable is forwarded on API requests, enabling access to private repositories. Updates re-download the latest release; uninstallation removes both the binary and the configuration entry.

Reference Plugins

Three plugins are developed as part of this thesis to demonstrate the generality of the protocol. Each is an independent Go module with its own repository and release pipeline, demonstrating that the plugin protocol imposes no coupling to the ADE release cycle. A fourth plugin, written in Java by an independent author,

confirms that the protocol can be implemented outside Go as well; it is presented later in Section 5.2.

The compile plugin *archgo*[22] translates code-structural rules into a Go test file using the Arch-Go library. It supports three rule kinds: `RULE_NOT_DEPEND` maps to `ShouldNotDependsOn`, `RULE_DEPEND_ONLY` maps to `ShouldOnlyDependsOn`, and `RULE_ACYCLIC` maps to `ShouldNotContainCycles`. All other rule kinds (inheritance, annotation, visibility, type constraint) have no Arch-Go equivalent and are emitted as comments in the generated file, making the coverage gap visible to the architect. The generated file follows the naming convention `ADR_<id>_archgo_test.go` and integrates with Go's standard `go test` toolchain without additional configuration. Listing 9 shows the Go test file that the plugin generates from the AD0001 IR.

Listing 9: Go test file generated by the *archgo* plugin for AD0001.

```
// Code generated by ade arch-go plugin; DO NOT EDIT.
package main

import (
  "testing"

  archgo "github.com/arch-go/arch-go/api"
  config "github.com/arch-go/arch-go/api/configuration"
)

func TestArchitectureFrom_0001(t *testing.T) {
  configuration := config.Config{
    DependenciesRules: []*config.DependenciesRule{
      {
        Package: "<repository>/internal/domain..",
        ShouldNotDependsOn: &config.Dependencies{
          Internal: []string{
            "<repository>/internal/application..",
          },
        },
      },
      {
        Package: "<repository>/internal/domain..",
        ShouldNotDependsOn: &config.Dependencies{
          Internal: []string{
            "<repository>/internal/adapter..",
```

[22] https://github.com/phi42/ad-plugin-archgo

```
          },
        },
      },
      {
        Package: "<repository>/internal/domain..",
        ShouldNotDependsOn: &config.Dependencies{
          Internal: []string{
            "<repository>/internal/infrastructure..",
          },
        },
      },
    },
  }

  moduleInfo := config.Load("<repository>")

  result := archgo.CheckArchitecture(moduleInfo, configuration)
  if !result.Pass {
    t.Fatalf("Architecture rules from ADR %q (%q) are violated", "0001",
    "Domain layer is independent from upper layers")
  }
}
```

The compile plugin *netarchtest*[23] generates C# NUnit test classes using the NetArchTest library. It targets a significantly larger subset of the IR than archgo: dependency, inheritance, interface implementation, annotation, namespace containment, naming, visibility, and type constraint assertions are all supported. Two rule kinds have no NetArchTest equivalent: `RULE_ACCESSED_BY` (incoming dependency checks) and `RULE_ACYCLIC` (cycle detection), which are logged as skipped rules. The generated file contains one `[Test]` method per rule, and exclusion clauses are translated into chained predicates. Rules with severity `warning` use `Assert.Warn` instead of `Assert.That`, so that warnings remain visible in the test report without failing the build. The generated file follows the naming convention `ADR_<id>_NetArchTest.g.cs`. An example of the generated C# test class is shown in the case study (Section 5).

The verify plugin *fscheck*[24] evaluates filesystem assertions directly against the target directory tree. It supports existence, absence, content matching, and content absence checks. Path patterns support `*`, `?`, and `**` (recursive descent) wildcards.

[23] https://github.com/phi42/ad-plugin-netarchtest
[24] https://github.com/phi42/ad-plugin-fscheck

The plugin prints structured pass/fail results—and a warning if the rule file contains code rules—to the terminal and exits with code 1 if any error-severity check fails, for example:

```
warning:    2 rule(s) skipped (plugin can only handle file rules)
passed:     0004 - readme_must_exist
failed:     0004 - license_must_exist (expected path "LICENSE" to exist)
passed:     0004 - gomod_must_exist
passed:     0004 - gitignore_must_exist

error: one or more file checks failed
```

4.7 CLI Reference

ADE is primarily a standalone command-line tool, invoked as `ade`. Table 4 lists the enforcement-related commands it exposes.

Table 4: Enforcement commands provided by ADE.

Command	Key Flags	Description
`validate`	`-i` (input)	Parse and validate rule files without invoking a plugin. Reports syntax and semantic errors.
`compile`	`-i`, `-p` (plugin)	Parse rule files and invoke a compile-mode plugin to generate test artefacts.
`verify`	`-i`, `-p`	Parse rule files and invoke a verify-mode plugin to check assertions immediately.
`config set`	key, value, `--global`, `--config`	Set a default value (e.g. default plugin name).
`config get`	key, `--global`, `--config`	Display the effective value of a configuration key.
`config unset`	key, `--global`, `--config`	Remove a default value.
`config list`	`--global`, `--config`	Show all configured defaults.
`plugin install`	name, `--path` or `--repo`	Install a plugin binary globally.
`plugin uninstall`	name	Remove an installed plugin.
`plugin update`	name, `-v` (version), `-a` (all)	Re-download the latest or a pinned release of a remote plugin; `--all` updates all remote plugins at once.
`plugin list`		List all installed plugins with their paths.

The `validate` command accepts multiple input paths via a repeated `-i` flag. Each path is expanded: if it refers to a directory, all `.rule` files within it are collected recursively; if it refers to a file, it is validated directly. This convention allows both single-file validation (`ade validate -i ad0001.rule`) and batch validation (`ade validate -i docs/adr/`). The `compile` and `verify` commands follow the same input expansion logic but additionally require a plugin name via `-p` or through the configured default.

Rule files are intended to live alongside the ADR markdown files they enforce. For example, the ADG tool's own repository contains the following structure:

```
docs/adr/
  AD0001-domain-layer-is-independent.md
  AD0001-domain-layer-is-independent.rule
  AD0002-application-does-not-depend-on-adapters.md
  AD0002-application-does-not-depend-on-adapters.rule
  ...
```

This convention ensures that rule files are version-controlled together with the decisions they enforce, that they appear in the same directory listing as the ADR, and that they can be discovered automatically by globbing for `*.rule` in the ADR directory. However, this is a convention rather than a technical requirement; the `-i` flag allows rule files to be located anywhere in the filesystem, and the ADR header within the rule file ensures that the link between decision and rules remains intact regardless of file location.

4.8 Integration with ADG and CI

ADE is additionally packaged as a Go library so that any other Go program can register its command tree. The ADG tool uses this integration path: it imports the ADE `cmd` package and exposes the entire ADE command tree under the `adg enforce` subcommand, continuing the workflow established in the preceding project thesis [8]. From a user perspective, `ade compile` and `adg enforce compile` are interchangeable; the difference is only which entry binary the user installs.

In addition to the enforcement commands, the ADG tool provides `adg enforce rule`, which generates a `.rule` file template for an existing ADR. The ADR can be identified either by its identifier (`--id`) or by its title (`--title`). By default, the rule file is placed in the same directory as the referenced ADR and shares the same base filename with a `.rule` extension, matching the co-location convention described in Section 4.7; an explicit output path can be provided with `--output`. The generated template pre-fills the mandatory ADR header and includes commented-out skeleton selector and assertion blocks to guide the architect in completing the file.

The ADG tool additionally exposes an MCP (Model Context Protocol[25]) server through `adg mcp run`, enabling AI assistants—such as GitHub Copilot in Visual Studio Code—to assist with rule file authoring. The server operates over standard input and output and is registered in the project's `.vscode/mcp.json` configuration file; running `adg mcp` without a subcommand prints the corresponding configuration snippet. Five tools are exposed to a connected AI assistant:

- `list_adrs` returns the full ADR catalogue with each decision's identifier, title, and status.
- `get_adr` retrieves the markdown content of a specific decision and reports the expected path for its companion rule file.
- `get_dsl_reference` returns the complete ADE rule DSL specification, sourced directly from the ADE library, so the assistant always works from the canonical grammar.
- `list_rule_files` returns all existing rule files with their full content, providing concrete examples the assistant can follow when drafting new ones.
- `validate_rule` invokes the ADE parser against a candidate rule file to check syntax and semantics before the file is written to disk.

This integration closes the feedback loop between the ADR narrative and the rule file: an AI assistant can read the decision rationale through `get_adr`, consult the DSL grammar through `get_dsl_reference`, draft a rule file, and immediately validate it through `validate_rule`—all without leaving the editor.

End-to-End Workflow

Through the ADG integration, the full decision lifecycle is supported in a single command-line interface:

1. *Identify*: `adg add` creates a new ADR from a template.
2. *Make*: `adg decide`, `adg edit`, `adg link` refine the decision and connect it to related decisions.

[25] `https://modelcontextprotocol.io`

3. *Enforce*: `adg enforce rule` creates a template rule file; the architect fills in selectors and assertions; `adg enforce compile` or `adg enforce verify` translates and executes the rules.

The first two phases are implemented and described in the preceding project thesis [8]. The third phase is the focus of this thesis and is implemented by ADE. This three-phase workflow addresses the issue raised in the introduction by ensuring that architectural decisions are documented, linked and enforced within a single toolchain. Furthermore, every conformance violation is traceable back to the originating ADR.

CI Workflow

ADE is also practical for continuous integration (CI). The ADG repository's own GitHub Actions workflow[26] demonstrates the integration. The pipeline performs the following steps on every push to the `main` or `development` branch:

1. Build the `adg` binary from source.
2. Install the archgo and fscheck plugins from their pinned GitHub releases using `adg enforce plugin install --repo`.
3. Compile code-structural rules into architecture tests: `adg enforce compile` (the input directory and target plugin are taken from the repository's `.ade.yaml`).
4. Verify filesystem rules directly: `adg enforce verify`.
5. Run all Go tests (including the generated architecture tests) with `go test ./...`.

Listing 10 shows the corresponding GitHub Actions workflow file that implements these steps.

[26] https://github.com/adr/ad-guidance-tool/blob/main/.github/workflows/test.yml

Listing 10: GitHub Actions workflow for the ADG tool (test.yml).

```
name: Test

on:
  push:
    branches: [main, development]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-go@v5
        with:
          go-version-file: go.mod

      - name: Build adg
        run: go build -o bin/adg ./adg

      - name: Install archgo plugin
        run: ./bin/adg enforce plugin install archgo
             --repo github.com/phi42/ad-plugin-archgo@v0.1.2-dev
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

      - name: Install fscheck plugin
        run: ./bin/adg enforce plugin install fscheck
             --repo github.com/phi42/ad-plugin-fscheck@v0.1.2-dev
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

      - name: Compile ADR rules into architecture tests
        run: ./bin/adg enforce compile

      - name: Verify file-system rules
        run: ./bin/adg enforce verify

      - name: Run all tests
        run: go test ./...
```

This pipeline validates the tool's own architectural decisions on every commit, demonstrating that ADE is practical for continuous use and that generated tests integrate seamlessly with Go's standard test toolchain. Among the decisions enforced by this pipeline is AD0001 (Listing 6): every push therefore re-verifies that the domain layer remains independent of the layers above it. The case study in Section 5 applies the same integration pattern to a .NET repository using the *netarchtest* and *fscheck* plugins.

5 Validation

This section presents the empirical validation of ADE. The main case study applies the toolchain to an open-source .NET modular monolith comprising 17 documented architectural decisions, providing the primary quantitative evidence. Section 5.1 describes the target repository and the study procedure, the ADRs it contains, the authored rule files, the CI pipeline integration, and the results. Section 5.2 summarises an additional validation activity.

5.1 Case Study: Modular Monolith with DDD

The *modular-monolith-with-ddd* repository[27] is an open-source .NET application that implements a meeting-management system using a modular monolith architecture with Domain-Driven Design patterns. The repository was selected because it contains 17 ADRs covering both structural and process-level decisions, its architecture has well-defined bounded-context boundaries that an enforcement tool can check, and it already contains a handwritten architecture test suite using the NetArchTest library, which provides a baseline for comparison with the generated tests. All modifications were made on a public fork of the repository[28].

The case study follows a six-step procedure:

1. Analyse the documented ADRs of the target repository and record their identifiers, titles, and scope.
2. Classify each ADR by decision type using the taxonomy of Keim et al. [14].
3. Assess each ADR's enforceability: can the decision's constraints, as written, be expressed in the rule file DSL and checked by one of the available plugins?
4. Author rule files for the enforceable subset, co-located with the ADRs.
5. Integrate the enforcement commands into the repository's CI pipeline.
6. Execute the pipeline and analyse the results.

[27] https://github.com/kgrzybek/modular-monolith-with-ddd
[28] https://github.com/phi42/ad-casestudy

ADR Overview and Classification

The repository contains 17 ADRs in Markdown format, all of which are located in the `docs/architecture-decision-log/` directory. These decisions cover system decomposition, architectural styles, design patterns, technology choices and development process conventions. Table 5 lists all 17 decisions.

Table 5: ADR overview of the *modular-monolith-with-ddd* repository.

ID	Title
0001	Record Architecture Decisions
0002	Use Modular Monolith System Architecture
0003	Use .NET Core and C# language
0004	Divide the System into 4 Modules
0005	Create One REST API Module
0006	Create Facade Between API and Business Module
0007	Use CQRS Architectural Style
0008	Allow Return Result After Command Processing
0009	Use 2 Layered Architectural Style for Reads
0010	Use Clean Architecture for Writes
0011	Create Rich Domain Models
0012	Use Domain-Driven Design Tactical Patterns
0013	Protect Business Invariants Using Exceptions
0014	Event-Driven Communication Between Modules
0015	Use In-Memory Events Bus
0016	Create an IoC Container Per Module
0017	Implement Architecture Tests

Each ADR was classified using the taxonomy of Keim et al. [14], which distinguishes *existence decisions* (what elements exist in the architecture), *property decisions* (what qualities elements must exhibit), and *executive decisions* (organisational or process-level choices). Based on this classification, each ADR was assessed for enforceability against the current toolchain: the *Enforceable* column in Table 6 answers the question *"can the decision's constraints, as written in the ADR, be expressed in the current rule file DSL and verified by the netarchtest or fscheck plugin available at the time of the study"*. A "No" therefore reports a limitation of the rule-file plus plugin combination for this specific ADR text, not an inherent claim that the decision category can never be enforced.

Table 6: Decision type classification and enforceability assessment.

ID	Decision Type	Enforceable	Justification
0001	Executive (process)	No	Meta-decision (use ADRs)
0002	Existence (arrangement)	No	Meta-level style choice
0003	Executive (technology)	No	Technology selection
0004	Existence (structural)	Yes	Module boundaries checkable
0005	Existence (structural)	Yes	Single API project verifiable
0006	Existence (arrangement)	Yes	Facade boundary enforceable
0007	Existence (arrangement)	Partial	Covered by existing tests
0008	Property (guideline)	No	Not structurally testable
0009	Existence (arrangement)	Yes	Layer structure verifiable
0010	Existence (arrangement)	Yes	Clean arch. layers checkable
0011	Property (design rule)	No	Member-level encapsulation
0012	Existence (arrangement)	No	Pattern-level concepts
0013	Property (behavioural)	No	Exception hierarchies
0014	Existence (behavioural)	Yes	Module isolation checkable
0015	Executive (technology)	Yes	Infrastructure presence
0016	Existence (arrangement)	Yes	IoC structure verifiable
0017	Executive (process)	Yes	Test project presence

Of the 17 ADRs, 9 were assessed as enforceable, 1 as partially enforceable (already covered by existing handwritten tests), and 7 as not enforceable. The non-enforceable decisions fall into three categories. The first comprises meta-decisions about process or technology choices (ADRs 0001–0003). The second comprises design-rule decisions that require class-member-level assertions beyond what the DSL expresses (ADRs 0008, 0011, 0013). The third comprises pattern-level concepts that have no direct structural manifestation checkable by namespace-dependency or type-level rules (ADR 0012).

Several of these "No" entries are not permanent verdicts. A design-rule decision could become enforceable by rewriting the ADR to name a structural artefact rather than a behavioural invariant, by extending the DSL with new rule kinds, or by writing a plugin that performs the missing analysis. The trade-offs of each mitigation are discussed in Section 6.2.

Rules

For each of the 9 enforceable ADRs, a `.rule` file was created and placed alongside the corresponding ADR markdown in the `docs/architecture-decision-log/` directory. Table 7 summarises the rule files and the number of assertions each contains.

Table 7: Authored rule files and assertion counts.

ADR	Code Rules	File Rules	Generated Tests
0004	5	2	7
0005	0	2	2
0006	1	0	1
0009	0	3	3
0010	15	0	15
0014	10	0	10
0015	0	2	2
0016	0	2	2
0017	0	2	2
Total	**31**	**13**	**44**

The 9 rule files together contain 31 code-structural assertions and 13 filesystem assertions, producing 44 generated conformance checks when compiled by the netarchtest plugin and verified by the fscheck plugin. The following paragraphs describe the principal rule patterns encountered.

ADR 0004, 0014: Module Isolation. The most complex rules enforce module boundary isolation. ADR 0004 defines the system's decomposition into five bounded-context modules (Meetings, Administration, Payments, UserAccess, Registrations), and ADR 0014 mandates that cross-module communication occurs only through integration events. The corresponding rules express pairwise `must not depend on` assertions with exclusion clauses for the legitimate integration mechanism, as shown in Listing 11.

Listing 11: Module isolation rule with exclusions (0004.rule).

```
code "meetings_isolated" {
  Meetings must not depend on Administration, Payments,
    UserAccess, Registrations
  exclude class implementing interface
    "MediatR.INotificationHandler<>"
  exclude class "EventsBusStartup"
  severity error
}
```

The `exclude class implementing interface` clause excludes integration event handlers (which subscribe to events published by other modules) from the dependency restriction. This pattern recurs in all module isolation rules and demonstrates the DSL's ability to express nuanced constraints with documented exceptions.

Listing 12 shows the corresponding generated NUnit test. The exclusion clauses are translated into chained `.And().DoNot...` predicates that narrow the type selection before the dependency assertion is applied. Each test method embeds the ADR identifier and title directly in its failure message, preserving traceability from a failing test back to the originating decision.

Listing 12: Generated NUnit test (ADR_0004_NetArchTest.g.cs).

```
[Test]
public void Rule_meetings_isolated()
{
    var result = Types.InCurrentDomain()
        .That()
        .ResideInNamespace("CompanyName.MyMeetings.Modules.Meetings")
        .And().DoNotImplementInterface(typeof(MediatR.
    INotificationHandler<>))
        .And().DoNotHaveName("EventsBusStartup")
        .Should()
        .NotHaveDependencyOnAny(/* other module namespaces */)
        .GetResult();
    Assert.That(result.IsSuccessful, Is.True,
        $"ADR 0004 (Divide the system into 4 modules), " +
        $"rule meetings_isolated violated");
}
```

One deliberate architectural violation was identified during rule authoring: the Registrations module depends directly on UserAccess through `UserAccessGateway` and `UserAccessAutofacModule`. This cross-module call creates a user account synchronously after registration and is an accepted exception documented in the rule file comments. The classes are excluded explicitly rather than treated as test failures.

ADR 0010: Clean Architecture Layering. ADR 0010 mandates a three-layer clean architecture (Domain, Application, Infrastructure) within each module for write operations. The constraint follows the same structural pattern as AD0001 (Listing 1), generalised from a single application across five bounded-context modules. The rule file defines selectors for all 15 layer components and expresses inward-only dependency constraints, as can be seen in Listing 13.

Listing 13: Clean architecture layer rules (0010.rule).

```
code "meetings_domain_is_inner" {
  MeetingsDomain must only depend on MeetingsDomain
  severity error
}

code "meetings_application_only_inward" {
  MeetingsApplication must only depend on
    MeetingsApplication, MeetingsDomain
  severity error
}
```

This pattern repeats for all five modules, producing 15 rules that together ensure no layer references an outer layer. The Registrations module's infrastructure layer includes an additional exclusion for the deliberate UserAccess integration point.

Listing 14 shows the generated test for the first rule. Because the DSL expresses inward-only dependence with `must only depend on`, the plugin inverts the constraint: it enumerates all known namespaces except the permitted set and passes them to `NotHaveDependencyOnAny`. The resulting test is therefore more verbose than the rule, but the intent remains identical.

Listing 14: Generated NUnit test (ADR_0010_NetArchTest.g.cs).

```
[Test]
public void Rule_meetings_domain_is_inner()
{
    var result = Types.InCurrentDomain()
        .That()
        .ResideInNamespace("CompanyName.MyMeetings.Modules.Meetings.
    Domain")
        .Should()
        .NotHaveDependencyOnAny(
            "CompanyName.MyMeetings.API",
            "CompanyName.MyMeetings.Modules.Administration.Application",
            "CompanyName.MyMeetings.Modules.Administration.Domain",
            /* all other module namespaces except MeetingsDomain */)
        .GetResult();
    Assert.That(result.IsSuccessful, Is.True,
        $"ADR 0010 (Use Clean Architecture for writes), " +
        $"rule meetings_domain_is_inner violated");
}
```

ADR 0006: Facade Boundary. ADR 0006 states that the API layer communicates with each module exclusively through its facade interface. The rule expresses this as a prohibition on the API depending directly on module Domain namespaces, as shown in Listing 15.

Listing 15: Facade boundary rule (0006.rule).

```
code "api_does_not_use_domain_directly" {
  API must not depend on MeetingsDomain, AdminDomain,
    PaymentsDomain, UserAccessDomain, RegistrationsDomain
  severity error
}
```

Listing 16 shows the complete generated test. The rule's selector names are resolved to their namespace pattern strings, and the assertion is expressed as a NetArchTest fluent chain. Because ADR 0006 contains only one rule, the generated file contains a single test method.

Listing 16: Generated NUnit test (ADR_0006_NetArchTest.g.cs).

```
[Test]
public void Rule_api_does_not_use_domain_directly()
{
    var result = Types.InCurrentDomain()
        .That()
        .ResideInNamespace("CompanyName.MyMeetings.API")
        .Should()
        .NotHaveDependencyOnAny(
            "CompanyName.MyMeetings.Modules.Administration.Domain",
            "CompanyName.MyMeetings.Modules.Meetings.Domain",
            "CompanyName.MyMeetings.Modules.Payments.Domain",
            "CompanyName.MyMeetings.Modules.Registrations.Domain",
            "CompanyName.MyMeetings.Modules.UserAccess.Domain")
        .GetResult();
    Assert.That(result.IsSuccessful, Is.True,
        $"ADR 0006 (Create facade between API and business module), " +
        $"rule api_does_not_use_domain_directly violated");
}
```

A stricter version forbidding access to Infrastructure namespaces was considered but not implemented because the API legitimately references `Infrastructure.Configuration` per ADR 0016, and NetArchTest's namespace matching cannot distinguish sub-namespaces.

ADR 0005, 0009, 0015, 0016, 0017: Filesystem Assertions. Five rule files use `file` blocks to verify the presence of expected project structure elements. These rules confirm that module directories exist (ADR 0004), that the single API project is present (ADR 0005), that query handlers reside in the correct layer (ADR 0009), that the in-memory event bus implementation exists and no external messaging framework is referenced (ADR 0015), that each module has a composition root (ADR 0016), and that the architecture test project exists with a NetArchTest dependency (ADR 0017).

Listing 17 shows the complete rule file for ADR 0015. The first rule verifies that the in-memory event bus implementation file exists at its expected path. The second uses the `must not contain` assertion with a regular expression pattern to confirm that no application configuration file references an external messaging framework. The `warning` severity means a violation is reported but does not fail the CI build.

Listing 17: Filesystem assertion rule file for ADR 0015 (0015.rule).

```
adr "0015" "Use In-Memory Events Bus"

file "inmemory_eventbus_exists" {
  path "BuildingBlocks/Infrastructure/EventBus/InMemoryEventBus.cs"
    must exist
  severity error
}

file "no_external_messaging" {
  path "**/appsettings*.json"
    must not contain "regex:RabbitMQ|MassTransit|NServiceBus"
  severity warning
}
```

Pipeline Integration

The enforcement commands were integrated into the repository's CI pipeline through a modified GitHub Actions workflow. The pipeline defines two parallel jobs that run on every push and pull request.

The build job installs the `ade` binary via `go install`, downloads the netarchtest plugin from its pinned GitHub release, and generates architecture test files from all rule files in the ADR directory:

```
- name: Install ade
  run: go install github.com/phi42/ad-enforcement-tool/ade@v1.0.0

- name: Install netarch plugin
  run: ade plugin install netarch
    --repo github.com/phi42/ad-plugin-netarchtest@v1.0.0

- name: Generate architecture tests from ADR rules
  run: ade compile
```

The generated `.g.cs` files are written to a `Generated/` subdirectory within the existing ArchTests project. They are regenerated on every pipeline run. The subsequent `dotnet build` and `dotnet test` steps compile and execute both the

existing hand-written architecture tests and the newly generated ones as part of a single test run. Figure 6 illustrates the pipeline structure for the build job.

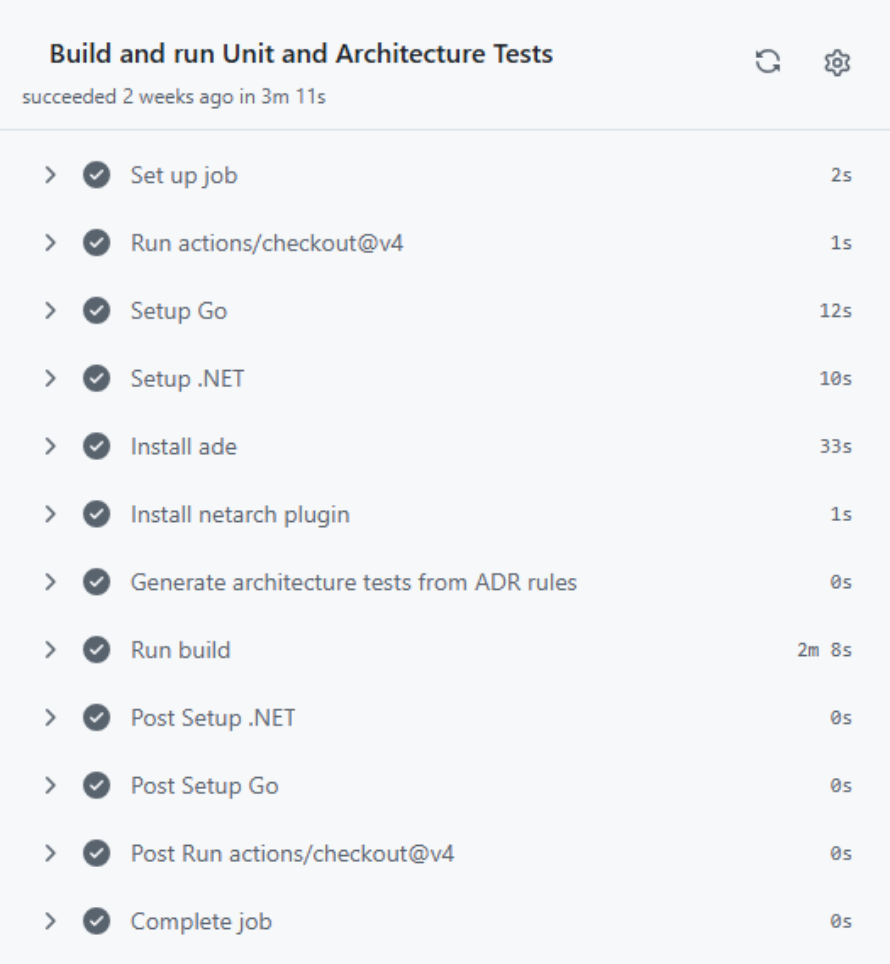

Figure 6: GitHub Actions pipeline for architecture rule enforcement.

A separate rule-check job runs the fscheck plugin in verify mode against all rule files:

```
- name: Install fscheck plugin
  run: ade plugin install fscheck
    --repo github.com/phi42/ad-plugin-fscheck@v1.0.0

- name: Check architecture file rules
  run: ade verify
```

This job evaluates all filesystem assertions (existence, absence, content matching) directly against the `src/` directory and exits with code 1 if any error-severity check fails. Because file rules require no execution step using a test runner, this job provides rapid feedback on structural violations. Figure 7 illustrates the pipeline structure for the rule-check job.

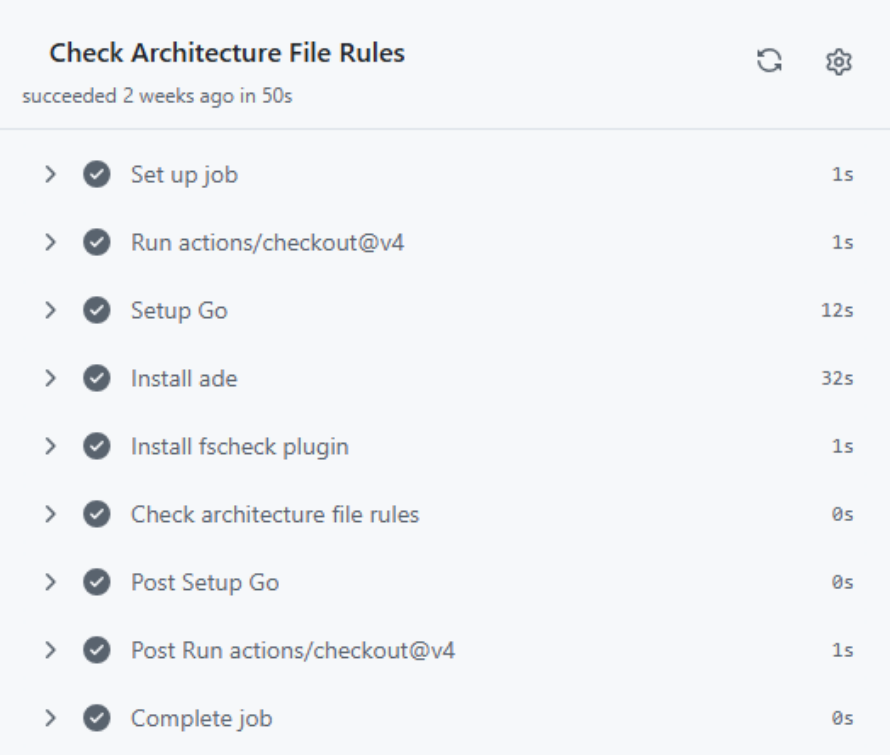

Figure 7: GitHub Actions pipeline for file rule verification.

Results and Findings

The pipeline execution produced all 44 generated conformance checks from the 9 rule files. Combined with the 8 hand-written architecture tests already in the repository, the suite comprises 52 architecture tests in total, all of which pass. The 9 enforceable ADRs translate directly into automated checks; the remaining 8 concern technology choices, coding conventions, or design patterns that require class-member-level or semantic analysis beyond the scope of the DSL.

Two architectural tensions were identified during rule authoring and resolved through documented exclusions rather than test failures:

1. **Registrations → UserAccess dependency.** The Registrations module calls UserAccess directly through `UserAccessGateway` to create a user account synchronously after registration. This violates the module isolation principle (ADRs 0004 module decomposition, 0010 Clean Architecture for writes, 0014 event-driven communication) but is an accepted exception; the relevant classes are excluded from three rule files with explanatory comments.

2. **API → Infrastructure.Configuration dependency.** The API host references each module's `Infrastructure.Configuration` namespace to initialise the per-module IoC container (ADR 0016). Because NetArchTest's prefix-based namespace matching cannot distinguish `Infrastructure.Configuration` from `Infrastructure`, a blanket prohibition on Infrastructure access would produce false positives. The rules enforce the meaningful constraint (no direct Domain access) while accepting the legitimate configuration references.

Both cases are deliberate design trade-offs documented in the ADRs themselves and preserved in the rule file comments. The DSL's exclusion mechanism proved essential for expressing these nuances.

Authoring the 9 rule files required reading each ADR, identifying its structural implications, and iterating on exclusion clauses. The module isolation rules (ADRs 0004 module decomposition and 0014 event-driven communication) required the most effort due to exclusion tuning for the legitimate integration mechanisms; the filesystem rules (ADRs 0005 single REST API, 0015 in-memory event bus, 0016 IoC per module, 0017 architecture tests) required only glob patterns and content checks. The DSL rules are significantly more concise than equivalent hand-written

tests: the 15 clean-architecture layer rules in `0010.rule` occupy 112 lines of DSL versus approximately 300 lines of C# in the existing ArchTests project. More importantly, the rules are co-located with the ADRs they enforce, creating an explicit traceability link that is absent from the hand-written tests.

Two limitations of the NetArchTest framework affected rule expressiveness:

1. *Prefix-based namespace matching.* NetArchTest selects types by namespace prefix, making it impossible to allow `Infrastructure.Configuration` while forbidding `Infrastructure` proper. This forced the relaxation of ADR 0006's facade rule.
2. *No incoming-dependency checks.* NetArchTest can assert that namespace A does not depend on namespace B, but cannot assert that namespace B is only accessed by namespace A.

Both limitations are properties of the chosen plugin. A plugin with more granular type analysis could enforce these constraints without changes to the DSL or IR.

The performance of the file-check and test-build jobs can be seen in Figure 6 and Figure 7. Each job takes approximately one minute end-to-end (besides the .NET test runner), with the dominant cost being the `go install` step that downloads the Go toolchain and compiles the `ade` binary. The actual rule evaluation and test generation each complete in under a second, and the generated tests add negligible overhead to the existing `dotnet test` execution. This setup cost could be eliminated by using a precompiled release binary or a Docker image with ADE pre-installed.

The case study demonstrates that ADE can be applied to a larger .NET repository with meaningful results: 53% of documented decisions become automatically enforceable, the generated tests integrate seamlessly with the existing test infrastructure, and the CI pipeline provides continuous conformance feedback on every commit. Section 6 discusses these findings in the broader context of the thesis goals.

5.2 Action Research

A second validation activity was conducted as action research with an architect. The architect applied the full ADG and ADE toolchain to an open-source event-driven serverless application[29] (referred to as TLA) that comprises three microservices written in .NET, Java, and Python. The activity unfolded in two phases: a half-day workshop focused on ADR management and rule authoring, followed by an independent post-workshop extension that introduced a new plugin for the Java ecosystem.

ADR Management with ADG. Before authoring any rule files, the architect used ADG to generate and manage the architectural decision records for the TLA repository. Seven architectural decision records were produced, each capturing a distinct architectural choice in the system, such as the event-driven communication pattern between services and the selection of an in-memory queue. This ADG-first workflow demonstrates the intended end-to-end usage of the toolchain: ADRs are created and maintained through ADG, and ADE then consumes those records to derive enforcement artefacts.

Rule Authoring and Toolchain Setup. During a workshop session, the architect worked through the complete ADE workflow from rule authoring to live CI feedback. The session covered three activities. First, the architect read the seven ADRs and assessed their enforceability, then authored `.rule` files for the enforceable subset using the DSL. Second, the toolchain was configured: ADE was installed, the relevant plugins were downloaded and registered, and the configuration file was set up to point at the ADR directory. Third, the enforcement commands were wired into the repository's CI pipeline, adding `ade compile` and `ade verify` steps to the existing workflow.

At the time of the workshop, three plugins were available: the *netarchtest* plugin for .NET structural analysis, the *fscheck* plugin for filesystem assertions, and the *archgo* plugin for Go codebases. Six of the seven ADRs proved enforceable using this set. The three .NET microservice constraints were covered by the *netarchtest* plugin; the filesystem assertions about project structure, configuration files, and directory layout were covered by *fscheck*. The Java microservice, however, could

[29] https://github.com/OST-Cloud-Application-Lab/tla-sample-serverless-eda-with-sqs-queue-sample

not yet be enforced: no plugin existed for the ArchUnit testing framework or any other Java analysis tool, leaving one ADR without automated coverage. This limitation was noted during the workshop as a concrete gap for post-workshop follow-up.

The workshop also revealed a few bugs in the toolchain that had not been apparent before, which were subsequently addressed. Most notably, although the toolchain supported custom project-level configuration using the `ade.yaml` file, it did not handle multiple configurations properly. This was crucial for this particular repository, as it required different configurations per microservice. A fix was implemented to enable multiple configurations to be specified and applied correctly per service (`.ade-manager.yaml`, `.ade-resolver.yaml`), allowing the architect to run the enforcement commands successfully for each microservice in the repository.

Post-Workshop Extension: Java Plugin and Multi-Language Pipeline. After the workshop, the architect independently implemented a plugin[30] written in Java for the ArchUnit testing framework. The plugin was developed by consuming the published `rule.proto` schema, with no involvement from the ADE authors and no modification to ADE itself. The plugin reads the intermediate representation emitted by `ade compile` and generates JUnit test classes that express the same structural assertions using ArchUnit's fluent API. Once integrated, the CI pipeline invoked both the *netarchtest* plugin (for the .NET service) and the *archunit* plugin (for the Java service) as parallel steps, with *fscheck* checking shared filesystem constraints across all three services.

The resulting pipeline enforces six of the seven documented decisions continuously across a three-language codebase. The two code-structural plugins are written in different host languages (Go and Java) and target different testing frameworks (NUnit via NetArchTest and JUnit via ArchUnit), yet both consume the same IR and integrate into a single workflow through the uniform ADE plugin protocol. This outcome directly validates the extensibility and generality requirements stated in Section 4.2: a new ecosystem can be supported by an independent author with no changes to the core tool, and the protocol is expressive enough to drive test generation across language boundaries.

[30] `https://github.com/stefan-ka/ad-plugin-archunit`

6 Discussion

This section interprets the results of the case study and design experience and provides a balanced evaluation of what ADE demonstrates, its boundaries, and its implications for the broader issue of bridging the gap between documented and implemented architectural decisions. Section 6.1 interprets the case study results against the taxonomy and design decisions. Section 6.2 reflects on the trade-offs and limitations of the approach and examines how far the findings generalise beyond the single case. Section 6.3 considers the threats to validity.

6.1 Interpretation of Results

The case study shows that a significant proportion of a typical architectural decision record (ADR) set can be enforced automatically. Of the 17 ADRs examined, 9 translated into 44 executable conformance checks without requiring modifications to the application source code. Mapping these results onto the taxonomy of Keim et al. [14] shows that *existence decisions* with a structural or arrangement character (module decomposition, layer organisation, component boundaries, facade interposition) are reliably enforceable because their constraints reduce to namespace-level or filesystem-level assertions. *Executive decisions* that concern infrastructure choices are similarly enforceable through filesystem checks. By contrast, *property decisions* that concern member-level encapsulation, *behavioural decisions* that concern semantic invariants, and *pattern-level* decisions fall outside the reach of the current domain-specific language (DSL).

The DSL rule file proved effective as a specification tool. Three design properties contributed most to this. Firstly, the co-location of the rule file with its ADR eliminates the traceability gap that exists in manually written test code. Secondly, separating selector declarations from rule bodies gives the rules a flat, declarative structure that can be understood by stakeholders outside the development team. Thirdly, the first-class exclude syntax enables authors to express documented exceptions alongside the constraints they qualify, rather than burying them in tool-specific configuration. The case study confirms this: the 15 clean-architecture layer rules comprise 112 DSL lines compared to approximately 300 lines of equivalent hand-written C#, and the readability advantage extends the audience for architectural rules beyond developers familiar with the specific test framework.

The separation between the DSL/IR layer and the plugin layer means that a single rule file can drive different enforcement frameworks without the need for specialisation on the host side. The case study uses two plugins (netarchtest in compile mode and fscheck in verify mode) from the same rule files. The evolution strategy of the protobuf intermediate representation (IR) ensures that new rule types can be added without breaking existing plugins.

6.2 Trade-offs, Generalisability, and Limitations

Rather than expecting authors to write architecture tests directly, the decision to introduce a dedicated DSL and a protobuf-encoded IR carries both benefits and costs. The main advantage is the decoupling confirmed by the case study: a single rule file expresses intent once and is processed by any compliant plugin. The main drawback is the additional layer that must be designed, parsed, validated and maintained over time. However, this cost is offset when rules are shared across multiple repositories, evolve under version control or need to be readable by stakeholders outside the development team. Even at the small end of the scale, with just one repository and nine rule files, the case study shows a favourable line-count comparison against equivalent hand-written code. For an organisation with several repositories sharing the same architectural style, the advantage would grow with each additional project.

The IR introduces a second trade-off. While a preserialised IR requires more effort than passing the parsed AST directly to a plugin, it enables plugin authors to be independent of the host language and documents the contract through the `.proto` schema. However, every new DSL construct must be promoted into the IR before any plugin can act on it, and the schema's evolution rules (additive changes only and numbered fields must be preserved) must be respected throughout the project's lifetime.

A third trade-off is the additional workload for the author, as a rule file is an extra artefact to write and maintain. While the case study demonstrates that this workload is manageable for structural decisions, the rule file cannot replace the ADR itself. The rule file captures the enforceable shape of the decision, while the ADR captures the reasoning. This demonstrates that both are necessary. This separation also has implications for architectural drift. Rule files act as a mechanical safeguard against silent erosion, catching any code change that violates a selector-and-assertion pair in the CI pipeline. However, the rule file remains distinct from the ADR it enforces. If a decision is revised but the corresponding rule

file is not updated, the enforcement pipeline will continue to check the outdated constraint, creating a second-order drift between the documented and enforced intents. Although co-location and shared identifiers make this drift easier to detect than in systems where tests and documentation live in different directories, they do not eliminate it. The rule file must still be maintained as the decision evolves, and currently, the toolchain offers no automated mechanism to flag a stale rule when its ADR is modified. One possibility would have been to integrate the DSL directly into the ADR template. However, this would mean that the tool would only work with a specific ADR template containing this DSL section, making it incompatible with popular, commonly used templates. Furthermore, since not every decision can be enforced, or since the author may decide not to enforce it, this section would often be left empty. Additionally, the tool would need to detect and parse this specific subsection of the file first.

In terms of generalisability, two properties of the target codebase determine how well the approach can be applied. The first is the explicitness of the architectural boundaries: a codebase whose modules are clearly delineated by namespace or directory conventions would be a good fit, whereas a monolith with ad hoc namespace organisation would not offer any stable selectors. The second property is the quality of the documented decisions: an ADR that names layers, modules and prohibited dependencies can be translated directly into a rule file, whereas a vague ADR cannot. Beyond the .NET ecosystem, the protobuf IR was designed specifically for language-agnostic use. The three reference plugins already cover Go, .NET and filesystem checks, and the existence of an independently-developed Java plugin confirms that adding plugins for other ecosystems is a self-contained engineering task that requires no changes to the DSL or to ADE.

Several limitations of the current work should be noted. The DSL targets structural conformance and cannot express decisions requiring semantic analysis, runtime behaviour or human judgement. Seven of the seventeen case study ADRs fall into these categories. The effectiveness of a rule depends on selector correctness. A selector that matches no types is considered valid. Although the tool emits a warning in this case, it cannot determine whether a non-zero match is intended. Each plugin maps IR constructs onto its own specific behaviour, such as prefix-based namespace matching. These properties affect the effective expressiveness of any rule targeting that plugin. The empirical evidence is drawn from a single repository and three reference plugins. While the qualitative claims (non-invasiveness, conciseness and IR decoupling) are expected to generalise, broader, multi-repository studies and third-party plugins would be required to make statistical claims about typical coverage. Finally, three plugins do not constitute a

population from which the IR's adequacy can be inferred in a closed-world sense: the introduction of an additional plugin that exposes an unanticipated requirement could necessitate a non-trivial schema change. The claim that the current IR is sufficient should therefore be read as "sufficient for the rule kinds encountered so far" rather than as a guarantee.

6.3 Threats to Validity

The validity discussion focuses on the case study as an empirical artefact, tool-side limitations have already been addressed in Section 6.2.

Construct validity. The case-study claim rests on the assumption that each authored `.rule` file faithfully encodes the constraint expressed in its source ADR. This is a translation step performed by a single author who was not involved in the original repository, so it carries a risk of misinterpretation. Two specific risks materialised during rule authoring: relaxing ADR 0006 to allow access to the `Infrastructure.Configuration` sub-namespace, and excluding the Registrations → UserAccess gateway from three module-isolation rules. Both deviations are documented in the rule-file comments alongside the assertion they qualify, so a reviewer can audit them, but neither has been independently validated against the intent of the original ADR authors.

Internal validity. A passing test does not prove that the underlying rule is correct; it proves only that no violation was detected for the types selected by the rule. A selector that names a non-existent or mistyped namespace can pass silently even when the architectural property is being violated by code outside the selector's reach. The tool emits a warning when a selector matches zero types but cannot determine whether a non-zero match is the intended set. Mitigation in this study was limited to manual inspection of selector patterns and to comparison against the existing hand-written architecture tests, which serve as an informal cross-check for the modules they happen to cover.

External validity. The principal case study is a single .NET modular monolith with documented ADRs and a pre-existing handwritten architecture test suite. Coverage ratios, authoring effort, and pipeline timings are all properties of this particular combination of architecture, ecosystem, and prior tooling. Generalising the quantitative results (53% enforceable, 44 generated checks, ≈1 minute pipeline overhead) to other architectural styles, other ecosystems, or repositories without an established test culture would require additional case studies of comparable depth.

The qualitative claims (non-invasive integration, DSL conciseness, ADR-to-test traceability) are expected to generalise more readily but remain to be confirmed in further studies.

Taken together, the discussion provides support for the central thesis that a tool-agnostic DSL coupled with a versioned IR and a plugin-based execution model could be a viable option to bridge the gap between ADRs and automated conformance checking for architectural decisions. The next section concludes the thesis and outlines directions for future work.

7 Conclusion

The aim of this thesis was to bridge the gap between documented architectural decisions and their automatic enforcement in source code. The preceding sections developed a conceptual approach and designed and implemented a domain-specific language (DSL) together with a tool-agnostic intermediate representation (IR), a standalone enforcement tool, and three reference plugins. The resulting tool, ADE, is shipped both as a standalone command-line binary and as a Go library, and is additionally embedded into the existing Architectural Decision Guidance (ADG) tool as the "`adg enforce`" subcommand. The entire pipeline was validated on a case study and additional action research covering .NET, Go, and a multi-language application. Section 7.1 summarises the results and revisits the contributions. Section 7.2 outlines directions for future research that the current implementation does not yet fully explore, and Section 7.3 provides closing remarks.

7.1 Summary

The problem motivating this thesis was the systematic disconnect between architectural decision documentation and architectural conformance checking. ADRs explain *why* a system is built the way it is, but the tools that verify whether the system actually conforms to those decisions operate at the code level and are decoupled from the documentation. Every team wanting both has to bridge the gap manually, translating text into tool-specific tests and keeping the two in sync as the codebase evolves. The result is that documented intent silently erodes despite being formally recorded, and the third phase of the decision lifecycle, decision enforcement, remains the weakest phase in practice.

The approach developed in this thesis closes this gap through three coordinated artefacts. A small, declarative DSL lets architects express the enforceable shape of a decision next to the ADR it complements, sharing the same identifier and the same directory. A protobuf-encoded IR serialises the parsed rule file into a self-contained message that any plugin can consume, decoupling the rule specification from the enforcement tool. A plugin model discovers, executes, and reports from arbitrary frameworks, demonstrated through three reference plugins covering Go (archgo), .NET (netarchtest), and direct filesystem checks (fscheck), and validated by an independently-developed Java plugin (archunit). ADE is shipped both as a standalone "`ade`" binary and as a Go library; the latter is the integration path used by the ADG tool, which exposes ADE as the "`adg enforce`" subcommand

so that the same workflow already used to manage ADRs now also manages and executes the rule files that enforce them.

The case study on an open-source repository provides end-to-end evidence that the approach works in practice. Of 17 documented architectural decisions, 9 translated into rule files that together produced 44 generated tests, all passing on the development branch alongside the 8 hand-written architecture tests already present. The rule files were integrated into a GitHub Actions Pipeline workflow with negligible performance overhead and no modifications to the application source code. Two architectural tensions that had been silently present in the codebase (the deliberate Registrations-to-UserAccess synchronous call and the API-to-Infrastructure.Configuration dependency) became visible and reviewable through documented exclusions in the rule files. The qualitative comparison with the equivalent hand-written tests showed a substantial reduction in lines of code and, more importantly, the establishment of an explicit traceability link between each rule and the ADR it enforces. Taken together, these results show that the structural subset of architectural decisions, which the taxonomy of Keim et al. [14] identifies as the most consistently checkable category, is well served by the proposed approach.

Concretely, the five contributions outlined in the introduction have been delivered as follows. The *systematic literature review* on decision documentation, drift and erosion, decision-to-constraint mapping, and conformance checking forms the foundation of Sections 2 and 3. The *rule file DSL*, *ADE itself* (DSL parser, semantic validation, intermediate representation, and plugin-based execution model, packaged as both a standalone tool and an embeddable library), and the *three reference plugins* (*archgo* for Go, *netarchtest* for .NET, and *fscheck* for filesystem assertions) are specified and implemented in Section 4. The *case study* and additional validation are reported in Section 5 and analysed in Section 6.

7.2 Future Work

The current implementation opens several lines of work that this thesis intentionally leaves to future iterations.

Broader Rule Kinds

The DSL currently emphasises structural conformance. Extending it to behavioural rules, configuration rules (e.g. production-only secrets in configuration files), and naming-convention rules (e.g. suffix and prefix policies) could be a next step.

Tighter ADG Integration

The current integration registers the "`adg enforce`" command tree and co-locates the rule files with their ADRs, but it does not yet surface enforcement state in the ADR lifecycle itself. A natural next step is to promote "enforced" to a first-class ADR status that is visible in "`adg list`" and "`adg view`", to generate status updates that indicate whether the latest CI run for an ADR's rule file passed, and to produce traceability reports that link each ADR to the tests it produces and the violations it has detected over time.

Additional Plugins for Other Ecosystems

The plugin contract is language-agnostic by design. The Java plugin for ArchUnit, developed independently of this thesis, already demonstrates that the contract generalises to the JVM. The remaining mainstream ecosystems where additional plugins would broaden the practical reach include Python (e.g. via an import-graph analyser), TypeScript (e.g. via an ESLint-based plugin or the TypeScript compiler API), and infrastructure-as-code policy engines (Terraform/OpenTofu via Open Policy Agent or Conftest). Each new plugin requires only an implementation of the small protocol described in Section 4.6 and does not affect any other part of the toolchain.

Semi-Automated Rule Extraction

Today, an architect writes a rule file by hand after writing the corresponding ADR. Keim et al. [14] demonstrate that machine-learning classifiers can categorise ADRs by decision type with reasonable accuracy. A natural extension is to use such a classifier as a first step in a recommendation pipeline that proposes rule-file skeletons for newly written ADRs: identify the decision type, suggest plausible

selectors based on the entities mentioned in the ADR text, and offer a starter set of assertions for the architect to refine. The Model Context Protocol server already shipped with ADG is a concrete enabler: it already exposes the ADR set and the DSL reference to AI assistants and validates the resulting rule files against the grammar, so the missing piece is the assistant prompt and a verdict step rather than new tooling.

The logical endpoint of this direction is to eliminate the rule file as a manually authored artefact altogether, making the ADR the single source of truth from which conformance checks are derived automatically. A generative AI component could read the ADR text, infer the structural constraints it implies, and emit a complete rule file without human intervention. Because such components are non-deterministic and can hallucinate selectors or assertions, their output would need to be treated as a proposed draft subject to review, not as a trusted specification. A practical architecture for this scenario would therefore separate roles: an AI component authors or updates the rule file, while the existing deterministic toolchain (parser, IR, plugin) continues to execute the checks. This preserves the repeatability and auditability of enforcement while reducing the manual effort of keeping rules in sync with evolving decisions.

Validity in the AI Era

As AI-assisted code generation becomes more widespread, the role of automated enforcement is likely to evolve. Generated code is unfamiliar to its authors and makes architectural intent harder to discern from the artefact alone. It also produces large diffs that are difficult to review manually. In this context, machine-checked architectural rules based on human-written ADRs could prove to be valuable. They provide an automated barrier that an AI assistant must pass before its contribution is accepted. They also encode design intent in a form that the assistant itself can consult when drafting changes. One particularly promising area for future research is investigating how the rule file DSL can be made available to AI code assistants, both as a constraint that limits their output and as a context document that informs it.

7.3 Closing Remarks

Although architectural decisions are made once, they must be adhered to continuously. The literature has long recognised the discrepancy between decision documentation and enforcement as a significant factor contributing to architectural erosion. A wide range of tools exists to address either aspect of the gap in isolation. This thesis contributes to this body of work by explicitly bridging the gap. It does this by introducing a small, readable language for expressing the enforceable shape of a decision, carrying that intent through a tool-agnostic intermediate representation, and dispatching it to plugins that speak the language of the target ecosystem. All of this can be done from within the same tool that already manages the ADRs themselves. The case study demonstrates the feasibility and usefulness of this approach for the structural subset of architectural decisions. The work that lies ahead involves the remaining categories, additional ecosystems, and integration with emerging AI workflows. The current implementation is a starting point, and the toolchain architecture is open to extensions by design.

References

[1] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," vol. 2005, 01 2005, pp. 109–120.

[2] P. Kruchten, "An Ontology of Architectural Design Decisions in Software-Intensive Systems," *2nd Groningen Workshop on Software Variability*, 01 2004.

[3] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, 09 2000.

[4] E. Whiting and S. Andrews, "Drift and Erosion in Software Architecture: Summary and Prevention Strategies," 02 2020.

[5] R. Li, P. Liang, M. Soliman, and P. Avgeriou, "Understanding Software Architecture Erosion: A Systematic Mapping Study," *CoRR*, vol. abs/2112.10934, 2021. [Online]. Available: https://arxiv.org/abs/2112.10934

[6] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, "N.: Reusable Architectural Decision Models for Enterprise Application Development," 07 2007.

[7] R. Schellander, "Concept Alternatives for the Management of Architectural Decisions in Clean Architectures," https://eprints.ost.ch/id/eprint/1280/, 2024, Project Thesis 1, Eastern Switzerland University of Applied Sciences.

[8] ——, "A Command-Line Tool for Managing Recurring Architectural Decisions: Design, Implementation, and Empirical Evaluation," https://eprints.ost.ch/id/eprint/1287/, 2025, Project Thesis 2, Eastern Switzerland University of Applied Sciences.

[9] A. von Zitzewitz, "Mitigating Technical and Architectural Debt with Sonargraph," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019, pp. 66–67.

[10] T. LaToza, E. Shabani, and A. van der Hoek, "A study of architectural decision practices," 05 2013, pp. 77–80.

[11] A. MacLean, R. Young, V. Bellotti, and T. Moran, "Questions, Options, and Criteria: Elements of Design Space Analysis," *Human-Computer Interaction*, vol. 6, pp. 201–250, 09 1991.

[12] R. Kazman, M. Klein, and P. Clements, "ATAM: Method for Architecture Evaluation," 10 2000.

[13] D. Le, S. Karthik, M. Schmitt Laser, and N. Medvidovic, "Architectural Decay as Predictor of Issue- and Change-Proneness," 03 2021, pp. 92–103.

[14] J. Keim, T. Hey, B. Sauer, and A. Koziolek, *A Taxonomy for Design Decisions in Software Architecture Documentation*, 07 2023, pp. 439–454.

[15] A. Caracciolo, M. Lungu, and O. Nierstrasz, "A Unified Approach to Architecture Conformance Checking," *Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015*, pp. 41–50, 07 2015.

[16] R. Hu, "An Automated Approach to Check Software Architecture Erosion," Master's thesis, Eindhoven University of Technology, 09 2023.

[17] A. Caracciolo, M. Lungu, and O. Nierstrasz, "Dictō: A Unified DSL for Testing Architectural Rules," *ACM International Conference Proceeding Series*, 08 2014.

[18] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2010.

[19] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[20] S. Schröder, M. Soliman, and M. Riebisch, "Architecture Enforcement Concerns and Activities - An Expert Study," *Journal of Systems and Software*, vol. 145, 08 2018.

[21] I. Lytra, H. Tran, and U. Zdun, "Constraint-Based Consistency Checking between Design Decisions and Component Models for Supporting Software Architecture Evolution," 03 2012.

[22] T. M. Ton That, S. Sadou, and F. Oquendo, "Using Architectural Patterns to Define Architectural Decisions," in *Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA)*. IEEE, 2012, pp. 196–200.

[23] R. Capilla, F. Nava, and J. Dueñas, "Modeling and Documenting the Evolution of Architectural Design Decisions," 06 2007, pp. 9–9.

[24] A. Bucaioni, A. Di Salle, L. Iovino, P. Pelliccione, and F. Raimondi, "Architecture as Code," 03 2025, pp. 187–198.

[25] N. Minsky, "Should architectural principles be enforced?" in *Proceedings Computer Security, Dependability, and Assurance: From Needs to Solutions (Cat. No.98EX358)*, 1998, pp. 89–102.

[26] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.

Declaration of Authorship

I, Raphael Schellander, declare that this master thesis titled *Architectural Decision Enforcement: From Architectural Decisions to Automated Conformance Checks* and the work presented in it are my own. I confirm that:

- This thesis was completed as part of the master's program at Eastern Switzerland University of Applied Sciences.
- All sources consulted and quotations from the work of others are clearly attributed throughout. With the exception of such quotations and the tool assistance listed below, this thesis is entirely my own work.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.
- All AI-generated output was reviewed and adapted to my own needs before inclusion.

Rapperswil, 26.06.2026

R. Schellander

Raphael Schellander

Tools Used

- *VS Code*[31] was used as the main editor for coding in Go as well as writing the thesis using LaTeX.
- *GitHub Copilot*[32] integrated in VS Code has been used for coding purposes, improving text and documentation, receiving feedback on the report, and assisting with diagram generation in LaTeX.
- *DeepL*[33] has been used for translations and to improve the text quality of this report by rephrasing sentences and correcting grammar.

[31] `https://code.visualstudio.com/`
[32] `https://github.com/features/copilot`
[33] `https://www.deepl.com/`