A Command-Line Tool for Managing Recurring Architectural Decisions: Design, Implementation, and Empirical Evaluation

Raphael Schellander
Supervised by Olaf Zimmermann

Eastern Switzerland University of Applied Sciences

June 23, 2025

Abstract — This thesis presents the design, implementation, and evaluation of ADG (Architectural Decision Guidance), a command-line tool for managing recurring architectural decisions. Based on previous conceptual work and a Clean Architecture decision handbook, ADG offers a structured, metadata-driven approach to managing the lifecycle of Architectural Decision Records (ADRs). Implemented in Go and aligned with Clean Architecture principles, ADG supports modular decision models using editable Markdown files, nested links, and guidance templates. ADG facilitates the creation, tagging, linking, validation, and reuse of ADRs through 18 command-line interface (CLI) commands, promoting model-driven knowledge reuse across projects. One of its key features is support for guidance modeling, which allows curated decision templates to be instantiated and merged across contexts. An empirical evaluation involving a software architect and a graduate student demonstrated the usability and practical relevance of the tool in realistic scenarios. ADG distinguishes between three roles: Knowledge Engineers, Model Tailors, and Architects/Developers. These roles enable collaboration through structured workflows. The result is an extensible CLI prototype that advances the practical tooling landscape for decision-centric architecture modeling.

Contents

1 Introduction **1**

2 Background and Related Work **3**

2.1 Architectural Decision Records 3

2.2 Existing Tools 5

2.3 Related Research 6

3 Tool Requirements, Design, and Implementation **10**

3.1 User Roles and Use Cases 10

3.2 Non-Functional Requirements 13

3.3 Decision Content, Metadata, and Indexing 14

3.4 Architecture Design and Implementation 19

3.5 Available Commands 24

4 Usage Scenario **26**

4.1 Setup 26

4.2 Workflow Example 27

5 Empirical Evaluation **30**

5.1 Setup and Task Description 30

5.2 Results and Feedback 32

6 Clean Architecture Guidance Model **34**

6.1 Defining Use Case Interfaces 35

6.2 Defining an Error Handling Strategy 37

6.3 Managing Cross-Cutting Concerns 39

6.4 Defining a Testing Strategy Across Layers 41

6.5 Defining a Dependency Injection Strategy 43

6.6 Structuring Use Case Modules 45

6.7 Defining Input/Output Boundaries 47

7 Conclusion **49**

A Task Description of Empirical Evaluation **53**

1 Introduction

Architectural Decision Records (ADRs) [1] have emerged as an effective and lightweight way to document the architectural decisions made during the development of software. By capturing key questions, alternatives and rationale, ADRs provide traceability, support knowledge transfer and encourage deliberate architectural design. However, as the number and interconnection of decisions increases in larger projects, it becomes increasingly important to find more structured and navigable ways to manage and explore these records.

In an initial project [2], an analysis of recurring architectural decisions from the Clean Architecture literature led to the proposal of a CLI tool concept for managing ADRs, alongside a curated decision handbook. The earlier study identified shortcomings in existing tools, including limited support for model reuse, poor traceability between decisions and an absence of metadata structures to support modular or nested ADRs. The project concluded with a conceptual design for a new CLI tool that addresses these requirements. Building on this foundation, this thesis presents the design, implementation and empirical evaluation of *Architectural Decision Guidance (ADG)*, a command-line tool that manages architectural decision records using a structured, indexable metadata model. The tool supports key operations such as adding, editing, deciding on, revising, linking, tagging and validating ADRs. It also enables users to reuse and adapt existing decision models by copying, importing or merging them, thereby facilitating the handling of recurring architectural concerns. An experiment was conducted to evaluate the usability and functionality of the tool. Two participants, an experienced software architect and a graduate computer science student, used the ADG tool to complete realistic tasks and simulate common workflows. Afterwards, they provided qualitative feedback.

The project was formally carried out at the Eastern Switzerland University of Applied Sciences. The task definition included the conceptual design of a metadata model for nested and linked ADRs; the implementation of the ADG command-line software; and the population of reusable decisions derived from prior work on Clean Architecture. Tool development was carried out iteratively in Go guided by Clean Architecture principles, aligning with the structure and semantics of the decision metadata model. Empirical evaluation was conducted via a structured, role-based user tutorial to validate the conceptual model and practical CLI implementation.

This thesis is structured as follows: Section 2 reviews the relevant background and existing ADR tools, building on the prior project work. Section 3 presents

the design and implementation of the ADG tool. Section 4 illustrates the usage scenario of the tool. Section 5 reports on the experimental setup and feedback. Section 6 expands on the Clean Handbook of the first thesis with seven additional decisions using the ADG tool. The thesis concludes in Section 7 discussing lessons learned and potential future directions.

2 Background and Related Work

Since the late 1990s, documenting software architecture decisions has been a well-established practice that aims to capture the rationale behind key design choices. [3] As systems evolve, teams grow and architectural complexity increases, the need for structured, accessible records of these decisions becomes more apparent. Architectural Decision Records (ADRs) are a lightweight, text-based format for consistently and maintainably documenting such decisions.

This section provides an overview of ADRs and their role in software engineering. It is followed by a discussion of the tools currently available in the ADR ecosystem. It also explains why a new command-line-based solution in Go was developed, based on limitations identified in current tooling. Much of the content in this section summarises findings from the previous thesis [2], in which these aspects were examined in greater detail.

2.1 Architectural Decision Records

"An Architectural Decision (AD) is a justified design choice that addresses a functional or non-functional requirement that is architecturally significant." [1]. To ensure transparency, knowledge transfer and long-term traceability, it is essential to document such decisions. One established method of doing so is the Architectural Decision Record. Michael Nygard popularised the concept of ADRs in a blog post, where he introduced the idea of capturing architectural decisions in a lightweight and accessible format. [3] Several more templates are available, as shown in Figure 1.

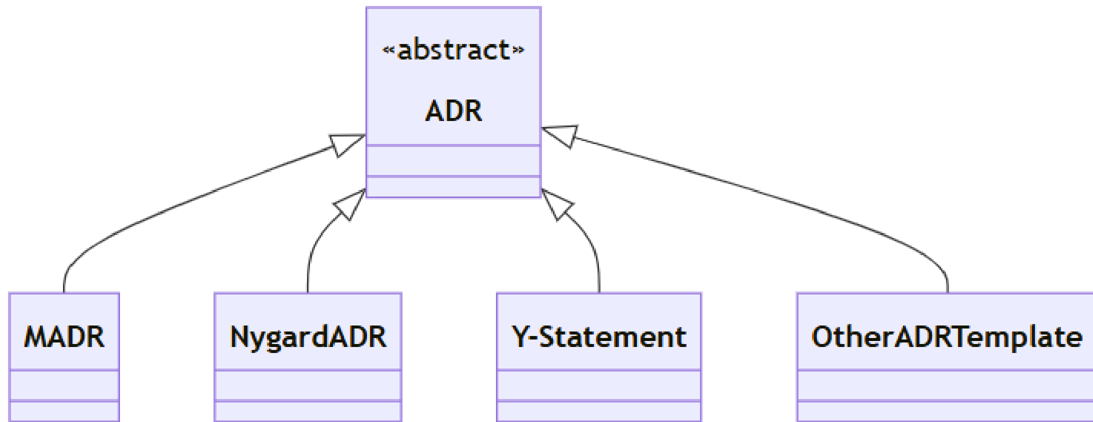

Figure 1: Templates for documenting Architectural Decision Records (source: ADR GitHub Organization [4])

- **Nygard Format**: This format emphasizes simplicity and minimalism. It includes essential elements such as *Context*, *Decision*, *Status*, and *Consequences*. It is well-suited for teams seeking low-overhead documentation but lacks structured sections for alternatives, rationale, or evaluation criteria, which can limit traceability and decision transparency. [5]

- **Markdown Architectural Decision Records (MADR)**: MADR formalizes ADR documentation using Markdown with both short and extended templates. MADR introduces sections such as *Status*, *Context and Problem Statement*, *Decision Drivers*, *Considered Options*, *Consequences*, and *Pros and Cons*, which enhance expressiveness. MADR is especially suited for integration into version-controlled repositories. [6]

- **Y-Statement**: This format condenses architectural decisions into one structured sentence. It follows this pattern: *"In the context of <project or situation>, facing <architectural challenge or need>, we decided for <chosen option> (and against <rejected alternatives>) to achieve <goal or desired outcome>, accepting that <consequences or trade-offs>."* This format enforces clarity, aligns rationale and decision in a compact way, and is especially useful in fast-paced or collaborative documentation contexts. [7]

There are trade-offs in terms of expressiveness, structure and usability for each template. Teams can select the format that best matches their documentation culture and tooling environment. However, what they all have in common is a focus on consistent, structured documentation that evolves alongside the system and supports both human understanding and team alignment.

2.2 Existing Tools

Several tools and libraries have been developed to support the creation and management of Architectural Decision Records. These tools vary in scope, user interface, and integration with software development workflows. Most operate on plain-text files and aim to simplify the process of maintaining consistent ADRs. Some representative tools are summarized in Table 1. This is a condensed selection based on the first thesis, which also included additional tools such as IDE plugins, language-embedded libraries, and web-based platforms.

Table 1: Overview of selected ADR tools

Tool	Description
adr-tools[1]	A command-line toolset implemented in Bash, designed for Unix-like systems. It provides basic functionality for initializing an ADR directory, creating new decisions using the Nygard format, and linking decisions through simple file naming and text references. Suitable for developers who want lightweight, version-control-friendly ADR management, but lacks support for structured metadata, advanced validation, or reusable decision models.
adr-log[2]	A Node.js/Javascript CLI utility that scans a directory of Markdown ADRs and generates a consolidated log or table of contents, either displayed in the terminal or injected into a file. Enhances traceability by automating ADR indexing, but does not support editing, viewing, or visual representation of decisions.
adr-viewer[3]	A Python-based tool (installable via PyPI or Homebrew) that transforms a repository of Markdown ADRs into a navigable static website or local web server. It parses ADR metadata, supports Mermaid diagrams, and offers an accessible UI for browsing decisions, without offering editing or management features.

[1] `https://github.com/npryce/adr-tools`
[2] `https://github.com/adr/adr-log`
[3] `https://github.com/mrwilson/adr-viewer`

Most existing tools prioritize simplicity and integrate well with text-based workflows and version control systems. However, they usually only provide limited support for structured metadata, tagging, validation, and the reuse and composition of decision models. Consequently, navigating and maintaining large or evolving sets of ADRs, especially across multiple teams or projects, can be challenging. These limitations prompted the creation of a new, CLI-based tool written in Go. The goal was to maintain the lightweight, file-based nature of ADR workflows while introducing a structured metadata model, enhanced validation, and support for the modular reuse of decision models. Go was chosen mainly as an opportunity to learn a new language, coming from a background of Java and C# development, but also for its performance, portability, and ease of deployment across platforms.

2.3 Related Research

In addition to praxis-oriented tools, the concept of capturing and structuring architectural decisions is well-established in academic research. Several frameworks and methodologies have been proposed to formalize decision modeling and support documentation of architectural rationale.

Jansen and Bosch [8] propose a foundational shift in architectural thinking by redefining software architecture as an explicit composition of design decisions rather than structural artifacts, such as components and connectors. They argue that architectural problems such as erosion, high change costs, and limited reusability stem from the phenomenon of "knowledge vaporization", whereby rationale and decision intent are lost over time due to implicit embedding in system structures. To counter this phenomenon, Jansen and Bosch advocate treating design decisions as first-class citizens, complete with rationale, alternatives, constraints, and consequences. This approach views architectural evolution as a series of deliberate, traceable transformations where decisions, not just code, drive architectural change. It also emphasizes the need for decision models that relate decisions to architectural elements, enabling structured reasoning, versioning, and analysis. The conceptual model presented in their Archium approach illustrates how decision modeling can be integrated throughout the architecture lifecycle with constructs like deltas, design fragments, and architectural modifications.

The ADG tool adopts a similar design philosophy, conceptualizing architecture as an evolving graph of traceable, interconnected decisions. The tool is designed to provide lightweight, file-based support for capturing explicit rationales, composing modular decisions, and maintaining lifecycle awareness in decision-centric model-

ing. This approach aligns with the broader aim of reducing architectural drift by making design intent explicit and navigable throughout a system's evolution.

Kruchten [9] presents a comprehensive ontology for architectural design decisions, arguing that such decisions should be treated as first-class entities in the architectural process. His taxonomy categorizes decisions into *existence decisions* (ontocrises), *property decisions* (diacrises), and *executive decisions* (pericrises), capturing both the presence and absence of architectural elements, as well as the constraints and strategic context shaping them. The ontology defines relationships between decisions (e.g., enables, forbids, overrides), their lifecycle states (e.g., tentative, approved, obsolesced), and metadata such as rationale, scope, and risk exposure. This structured approach supports traceability, evolution analysis, and architectural consistency by forming interlinked decision graphs.

The ADG tool's conceptual direction aligns with this vision by modeling architectural decisions as interconnected artifacts rich in context and metadata, rather than as isolated entries. ADG is designed to enable traceability of decisions across roles and over time by capturing rationale, references, and interdependencies in a lightweight, versionable format. While it does not fully implement Kruchten's ontology, ADG adopts its spirit by facilitating the structured capture of decisions and supporting awareness of the scope, constraints, and impact of decisions as part of a broader, decision-centric architectural mindset.

Van Heesch et al. [10, 11, 12] expand the notion of architectural decision modeling by focusing on the explicit documentation of *decision forces*, i.e., the various competing influences, such as requirements, stakeholder concerns, or contextual constraints, that shape architectural outcomes. They introduce multiple decision-oriented viewpoints, including the decision forces viewpoint, which models the traceability of stakeholder concerns and contextual forces to chosen alternatives. A key insight is the classification and balancing of these forces to make architectural rationale transparent and reproducible. Furthermore, they propose a reusable, cross-project decision model framework that allows teams to create architecture guidance tailored to recurring situations.

This notion of reusable viewpoints closely aligns with the intentions behind ADG, which aims to enable the systematic reuse and adaptation of architectural knowledge through templates and guidance models. However, rather than adopting a full standards-based meta-modeling approach, ADG targets a lightweight, file-based implementation that maintains conceptual compatibility with these ideas while offering accessibility through Markdown and Git-based tooling.

Building on the work of Jansen and Bosch, Van der Ven and his colleagues have expanded the concept of decision-centric architecture modeling, placing a greater emphasis on practical tools, the needs of stakeholders, and data-driven reasoning. They emphasize that architectural decisions should be understood as actionable, traceable entities, not just abstract units of rationale, and that these entities should be grounded in the day-to-day realities of software development. In their 2006 study [13], Van der Ven et al. propose integrating decision models with architectural documentation to support real-world software teams. Through industrial use cases, they demonstrate how different stakeholders, including architects, developers, and project leads, require different decision views and benefit from modeling support for tasks such as validation and reuse. Subsequent work introduces formalized tools to address these needs, including mechanisms for lifecycle tracking, dependency analysis, and stakeholder-specific views. [14].

The conceptual trajectory of this research significantly impacts the design intent of the ADG tool. Rather than implementing formal modeling languages, the ADG tool draws inspiration from this work by offering pragmatic, stakeholder-aligned mechanisms for capturing rationale, linking decisions, and enabling reuse. Its Markdown-based, GitHub-integrated format reflects the desire to incorporate decision modeling into the environments in which developers work, thereby bridging structured knowledge and everyday tooling.

Zimmermann et al. [15, 16] introduce a proactive framework for architectural decision modeling that goes far beyond the use of simple templates. Originally developed in the context of enterprise application and SOA design, their approach distinguishes between decisions to be made (issues) and those already resolved (outcomes). These form what they call a Reusable Architectural Decision Model (RADM). Central to this framework is the concept of a structured decision network model where decisions are represented in layered hierarchies (conceptual, technological, and asset-level) and connected through logical and temporal dependencies. Rather than treating architectural decisions as static, retrospective records, the framework promotes the semi-automated, method-driven identification, instantiation, and enforcement of decisions. To facilitate reuse and collaboration, Zimmermann introduces decision scoping, model filtering, and decision injection mechanisms that integrate decision outcomes into model-driven development processes. The framework is also supported by tools such as the Architectural Decision Knowledge Wiki (ADkwik) [17], which enable collaborative modeling and sharing of decision knowledge across projects. A concrete application of the framework is a structured SOA instance model that demonstrates how reusable decisions and architectural patterns can guide project-specific design efforts.

Together, these academic contributions establish a comprehensive theoretical foundation for decision-centric architecture modeling. Several recurring themes emerge: the need for traceability across decisions and artifacts; the explicit modeling of rationale and influencing forces; the reuse of structured decision knowledge; and the accommodation of diverse stakeholder concerns through dedicated viewpoints or roles. The design of ADG draws directly from these principles. ADG aims to operationalize these principles in a lightweight, developer-oriented environment by offering a modular, guidance-driven approach to decision modeling grounded in real-world workflows. Rather than replicating formal ontologies or standards-based frameworks, ADG positions itself as a pragmatic bridge between theory and practice. Its Markdown-based format and seamless integration with GitHub enable collaborative, version-controlled decision management that remains accessible to practitioners while respecting the lifecycle-oriented principles advocated in research.

3 Tool Requirements, Design, and Implementation

This section outlines the conceptual and technical foundations of the Architectural Decision Guidance (ADG) tool. It describes the intended use cases, target user roles, and quality requirements that guided its design, as well as the resulting implementation architecture. ADG is a command-line program designed to support the structured creation, editing, and reuse of Architectural Decision Records (ADRs) through lightweight, file-based workflows. It makes use of Markdown and YAML and centers around a metadata-driven model for organizing decisions into modular, reusable guidance structures.

The section begins by introducing the core user roles and their associated use cases, which define the tool's functional scope and provide the foundation for its interface design. Next, it presents a set of non-functional requirements based on the *FURPS* quality model (excluding functionality, already covered) to specify the desired qualities related to usability, reliability, performance, and supportability. Together, these requirements informed the development and evaluation of the ADG tool. Then, it explains the metadata structure and indexing mechanism that allows for managing linked and nested decisions in the form of decision models. Afterwards, it describes the underlying software architecture and technology stack used in the implementation. Finally, it presents the available commands of the current implementation.

3.1 User Roles and Use Cases

The functional design of the ADG tool was shaped by a set of representative user roles and their associated use cases. These roles represent common stakeholder responsibilities observed in architectural decision-making processes, and they were defined using a user-centered design approach. The purpose of these roles is to ensure that the tool delivers role-specific functionality aligned with realistic usage scenarios. This subsection introduces the three primary roles and illustrates how they interact through a shared example.

For instance, imagine that a software consulting firm maintains a curated library of architectural guidance for common challenges in microservices-based systems. At the beginning of a new client project, three stakeholders collaborate to adapt existing guidance to the project's architecture.

Knowledge Engineer: This role is responsible for maintaining the general guidance library. At the start of the engagement, the Knowledge Engineer uses the ADG tool to initialize a new guidance model for deployment strategies and adds a set of open decisions relevant to distributed environments. These decisions are linked to indicate dependencies, for example, that the choice of container runtime affects orchestration tooling. The decisions are further tagged based on categories like 'deployment', 'observability', or 'resilience', and edited as new insights arise. These activities are illustrated in Figure 2.

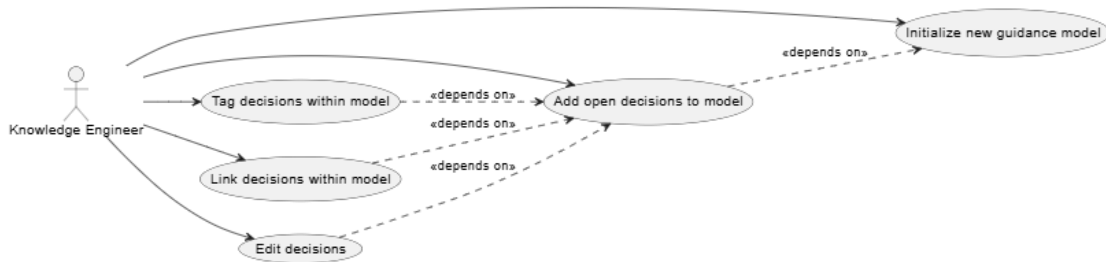

Figure 2: Use case diagram for the Knowledge Engineer Role

Model Tailor: Once the general model is complete, the Model Tailor selects it as a base and adapts it to the specific requirements of the new client project. The Model Tailor constructs a tailored decision model that blends parts of different guidance sets. For instance, a model for cloud-native deployment may be merged with a guidance model for security baselines. Throughout this process, the Model Tailor annotates decisions with comments to suggest refinements to the guidance content or clarify project-specific rationale. These use cases are shown in Figure 3.

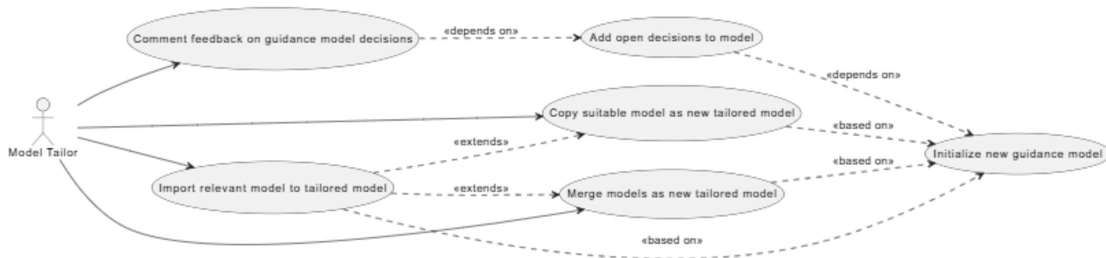

Figure 3: Use case diagram for the Model Tailor Role

Software Architect / Developer: Once the tailored model is finalized, it is handed off to the Software Architect or Developer, who is responsible for applying the decisions in context. They use the ADG tool to mark decisions as resolved, revise prior decisions as requirements evolve, and provide feedback to the other roles via comments. This part of the workflow is reflected in Figure 4.

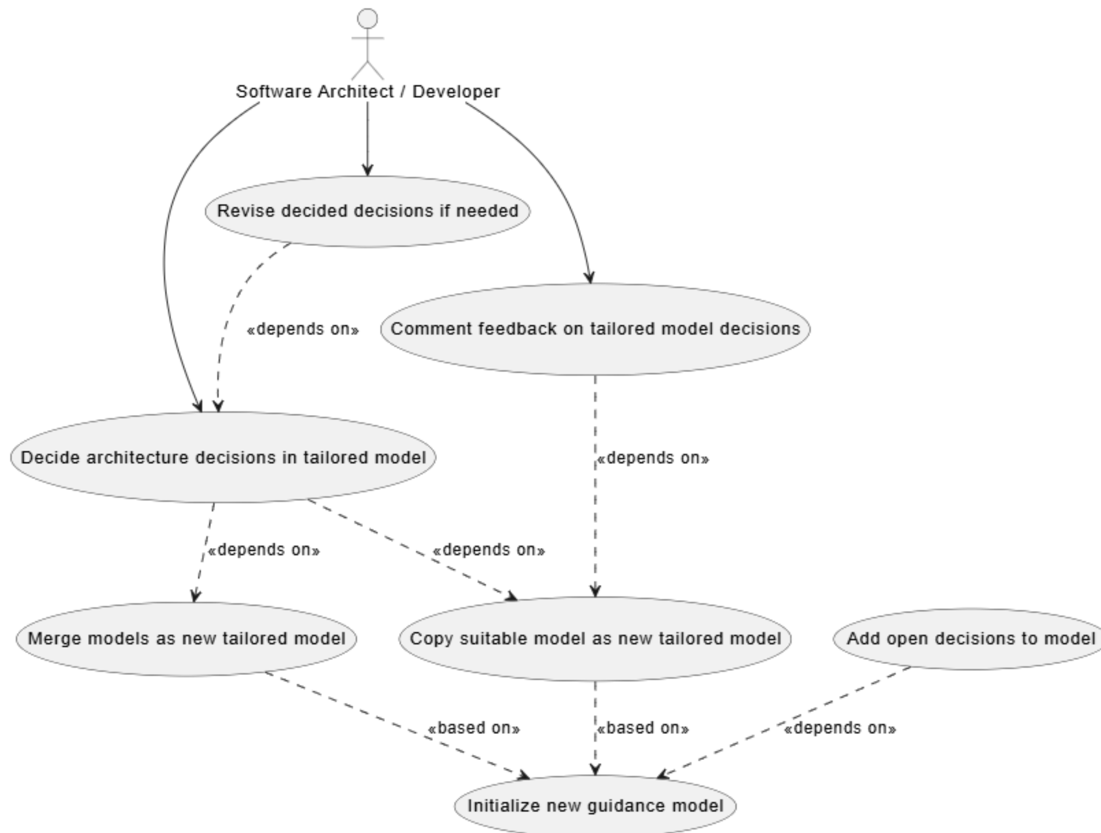

Figure 4: Use case diagram for the Software Architect / Developer Role

The tool was explicitly designed to support these roles with a set of commands that align with their respective workflows. Commands such as `add`, `edit`, `decide`, `copy`, `import`, and `merge` map directly to the core activities outlined in the use case diagrams. These user stories served not only as a guide during feature development but also as criteria for evaluation. Although the roles are distinct for the sake of modeling clarity, they are not mutually exclusive. In practice, a single user may act as a Knowledge Engineer, Model Tailor, and Software Architect at different stages of the project lifecycle.

3.2 Non-Functional Requirements

To complement the functional requirements defined by the user roles and use cases, this subsection defines the non-functional requirements (NFRs) that shaped the quality attributes of the tool. These requirements are categorized according to the *FURPS* model, focusing specifically on Usability, Reliability, Performance, and Supportability. Each quality dimension captures design considerations that contribute to a maintainable, robust, and user-friendly tool capable of supporting large-scale, real-world decision models.

Usability: To ensure a smooth, accessible user experience, several usability requirements were defined for the tool. First and foremost, the tool must provide informative and user-friendly error messages and usage tips when commands are misused or invoked incorrectly. This reduces frustration and enables faster error resolution, particularly for first-time users. Additionally, every command and subcommand must support a `--help` flag to offer on-demand documentation directly from the terminal. Consistency across commands is another crucial aspect. The syntax and structure of the tool should follow a uniform pattern to minimize cognitive load and reduce the learning curve. Finally, the tool should support user-specific customization through an optional configuration file. This flexibility allows the tool to adapt to varying user needs and project conventions without compromising its usability.

Reliability: Reliability is essential to ensure the tool behaves predictably and maintains the integrity of architectural decision models over time. To prevent structural inconsistencies, the tool must enforce constraints that stop cyclic dependencies when linking decisions. This is crucial to preserving a coherent, navigable decision graph. Additionally, the tool should be robust enough to recover from accidental file loss or manual edits. Another requirement is transactional behavior during command execution. If a command fails for any reason, the system must not be left in a partially modified or invalid state. For instance, metadata updates and content edits should only be committed after passing validation checks. To guard against malformed or incomplete decision files, the tool must validate the syntax to ensure that all records adhere to the expected ADR structure. This includes ensuring that all required metadata and content sections are included. Lastly, the tool should not lose or accidentally overwrite any user content.

Performance: Although performance is not the main focus, it is important for providing a responsive and productive user experience. The tool should be usable by teams of up to seven people who will work with it for several years, making thousands of decisions. Commands such as listing and filtering are common when exploring or analyzing large models. These operations should not disrupt the workflow with long loading times; rather, they should maintain interactivity. Most commands should execute in five seconds or less on a standard developer laptop when working with models containing up to 1000 decisions. The exception is commands that construct models, which are not used as much in the main workflow. This feature allows users to scale their decision models without sacrificing the usability of the tool or the efficiency of their workflow.

Supportability: The tool should allow for long-term support and extensibility. It is built to run reliably on major platforms, including Windows 10 (22H2) and Windows 11 (24H2), as well as macOS Monterey (12.x) and macOS Sequoia (15.x). It is also compatible with Ubuntu 20.04 and 22.04 LTS (long term support), ensuring accessibility for a broad developer base. The software architecture follows Clean Architecture principles, enabling developers familiar with this style to understand the structure and entry points within five to ten minutes. This promotes easier onboarding and community contributions to implement new commands. Additionally, the tool should support custom templates, such as Nygard and MADR. This allows users to customize the format by adding headers and metadata without causing errors, provided that the tool's minimum requirements are met.

3.3 Decision Content, Metadata, and Indexing

The ADG tool stores each architectural decision as a Markdown file using a structured template inspired by the Questions, Options, and Criteria (QOC) design rationale model proposed by MacLean et al. [18]. In this model, *Questions* define key design issues, *Options* represent possible alternatives, and *Criteria* express the goals, constraints, or quality attributes that guide the evaluation of the options. This framework was chosen because it provides a lightweight but powerful way to express the space of possible design decisions, make trade-offs explicit, and support structured reasoning without requiring a full formal model.

Accordingly, each ADR file created in ADG includes these three sections by default. The *Question* section frames the issue to be resolved, the *Options* section outlines the available alternatives, and the *Criteria* section defines the factors that influence the decision. Once a decision has been made, an additional *Outcome* section is

appended to document the selected option and rationale. An optional *Comments* section can also be included to support peer feedback or asynchronous discussion. This structured content format provides clarity, traceability, and consistency while remaining accessible for practical use in Markdown.

The tool uses HTML-compatible anchors to uniquely identify each section, while still allowing customizable section headers when rendered. Therefore, users can configure the tool to approximate templates such as *Nygard* or *MADR* by customizing the displayed section headers. While not all sections from these templates are supported by the tools commands, the key ones can be mapped to the available anchors using different headings. Additional headers can be added manually to the file to complete the templates without breaking the tool.

Example:

```
## <a name="question"></a> Question

What should be the initial architectural layout and structural
organization of the system?

## <a name="options"></a> Options

1. <a name="option-1"></a> Adopt Clean Architecture as the
foundational model with layered separation.

2. <a name="option-2"></a> Use a simpler layered architecture
(e.g., 3-tier: UI, Business Logic, Database).

3. <a name="option-3"></a> Allow organic, emergent structure
without enforcing a fixed layout.

## <a name="criteria"></a> Criteria

- Maintainability
- Adaptability to future changes
- Team familiarity and onboarding effort
- Testability and separation of concerns
- Tooling and framework compatibility
```

In addition to the content of the decision, each file includes a structured YAML front matter block that stores metadata. This metadata enables the efficient indexing, filtering, validation, and traceability of decisions. Using YAML ensures that the metadata is readable and compatible with version control systems, and it offers a structured format that the tool can easily parse and update. Metadata can also be added to conform to a desired template, such as MADR. The tool will not break so long as the default metadata are not deleted.

Each decision within a model has a *unique ID*. When decisions from a model are imported or merged, these IDs are updated to prevent duplication. The user can assign the *title* any arbitrary name, as long as it is a valid file name, since both the title and ID are used to generate the filename. The *status* attribute, either `open` or `decided`, is inspired by Kruchtens ontology of architectural decisions [9], which treats decision status as a key element for managing their lifecycle and traceability. *Tags* categorize decisions and are particularly useful in filtering and organizing views, such as through the `list` command. *Links* express relationships between decisions, following Zimmermann's concept of a connected decision network [16]. Two predefined link types are supported: `precedes` and `succeeds`, which represent temporal or logical ordering and form a directed acyclic graph. To ensure structural integrity, the tool validates new links to prevent cycles. In addition, users can define *custom links*, such as `invalidates` or `depends-on`, similar to the flexible relationship modeling described by Zimmermann. Finally, each decision may include metadata on *comments*, including author name and timestamp, enabling traceable, collaborative feedback over time.

Example:

```
---
adr_id: "0002"
title: decide-entity-boundaries
status: open
tags:
    - entities
    - domain-model
    - business-rules
    - clean-architecture
links:
    precedes:
        - "0003"
    succeeds:
        - "0001"
comments: []
---
```

To support fast metadata access and ensure consistency across large decision models, the ADG tool maintains a centralized index file named `index.yaml` in each model directory. This index serves as the summary of all decisions in the model, collecting key metadata, such as status, tags, links, and file paths, without needing to repeatedly parse each Markdown file. While this introduces some redundancy (since metadata is also embedded in each decision file), the duplication is intentional: it allows the tool to perform structure-level operations (e.g., filtering, listing, validation, or visualization) efficiently and robustly, even in large models. The index is updated incrementally whenever metadata-relevant operations are executed, such as adding new decisions, marking them as decided, or updating links. In cases where decision files are edited manually or deleted outside the tool, the index may become outdated. To resolve this, ADG provides a dedicated command to rebuild the index by scanning and parsing all decision files from scratch. This decoupling of metadata management from Markdown parsing not only improves runtime performance but also isolates error recovery logic, providing fault tolerance against user-driven file system changes. The tool treats the directory containing the index as the model root and supports nested subdirectories within it. Non-decision files are ignored, allowing users to organize large models hierarchically, e.g., by architectural layer or team responsibility, while retaining consistent, centralized indexing.

Example:


```yaml
decisions:
    "0001":
        adr_id: "0001"
        title: define-architecture-layout
        status: open
        tags:
            - architecture
            - layout
            - clean-architecture
            - structure
        links:
            precedes:
                - "0002"
            succeeds: []
        comments: []
    "0002":
        adr_id: "0002"
        title: decide-entity-boundaries
        status: open
        tags:
            - entities
            - domain-model
            - business-rules
            - clean-architecture
        links:
            precedes:
                - "0003"
            succeeds:
                - "0001"
        comments: []
    # more decision ...
```

To ensure the consistency and integrity of decision models, the tool has a built-in validation mechanism. This mechanism checks for common issues, such as missing metadata fields, duplicate IDs, and structural problems, like missing sections. Validation errors are reported in a structured format that allows users to quickly identify and resolve anomalies. Validation is closely linked to the index file, which serves as the canonical reference for decision metadata. If users make manual edits to decision files without updating the index, they can detect discrepancies via validation and rebuild an index file consistent with the decision files. This workflow maintains a clean separation between human-edited content and tool-managed metadata.

3.4 Architecture Design and Implementation

The ADG tool was implemented using the Go programming language. One benefit of Go is that it produces statically linked binaries that are easy to distribute and run on different platforms without external dependencies. This makes the tool highly portable and suitable for adoption in various development environments. Additionally, Go provides robust support for file input/output and file system traversal, both of which are essential to the ADR management model based on structured Markdown files. The tool operates independently, without the need for a central server or database. All decisions and model metadata are stored in local files in human-readable formats, which ensures transparency, version control compatibility, and offline usability. The tool's internal design adheres to the principles of Clean Architecture, which promote separation of concerns, testability, and long-term maintainability.

Architectural Style

The implementation of the tool follows the principles of Clean Architecture to promote modularity, long-term maintainability, and platform independence, qualities essential for a command-line tool intended to evolve alongside diverse use cases. This architecture was chosen because it enforces a strict separation of concerns and enables future extensibility without entangling business logic with infrastructure code.

In accordance with Clean Architecture, the codebase is organized into four concentric layers: *Domain*, *Application*, *Adapter*, and *Infrastructure*. Outer layers may depend on inner layers, but not vice versa. The domain layer contains the core logic for managing decisions and models. It defines the essential business rules, data structures, and validation routines, independent of how they are used or persisted. This ensures that the most critical logic is isolated from volatile implementation concerns. The application layer contains use case logic for each command, structured into interactors that coordinate domain models to perform tasks. It also defines input and output ports to facilitate communication with the adapter layer while preserving encapsulation. The adapter layer defines the actual CLI commands and terminal output formatting. It acts as the entry and exit point for each use case, parsing user input and rendering results. Finally, the infrastructure layer handles implementation-specific concerns such as file system access, YAML parsing, and configuration loading. Although file-based storage is central to the ADG tool, it is abstracted behind interface boundaries so that the system can remain portable and testable. This design enables the possibility of replacing local files with a remote API or database in the future, without affecting the core decision-handling logic.

Additionally, a top-level `cmd` package handles dependency injection, wiring up all required components for the CLI runtime. This ensures loose coupling and testability across the application stack.

Package structure:

```
adg/
+- cmd/               # Dependency injection and CLI bootstrap
+- internal/
   +- domain/         # Core business logic
   |  +- decision/
   |  +- model/
   |  +- config/
   +- application/    # Use case logic
   |  +- inputport/
   |  +- outputport/
   |  +- interactor/
   |     +- decision/
   |     +- model/
   +- adapter/        # CLI interface (commands and printers)
   |  +- command/
   |  +- printer/
   +- infrastructure/ # File system and configuration access
      +- decision/
      +- model/
      +- config/
```

Domain Model

The tool is based on a simple data model for architectural decisions, as shown in Figure 5. Each decision is represented by a 'Decision' structure containing all metadata fields and stored at the top of the decision file and in the central index file. Comments are embedded as a list containing the author, date, and comment text. The Links structure supports chronological relationships (precedes/succeeds) and custom semantic links. A separate 'DecisionContent' structure manages the actual content of a decision. This structure holds the textual data for each section (e.g., Question, Options, Criteria, Outcome, and Comments) and includes the ID that associates it with its corresponding metadata entry.

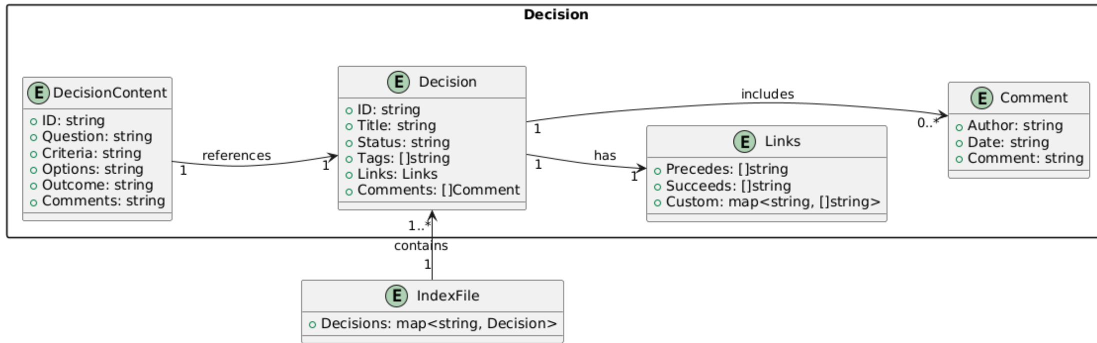

Figure 5: Domain Model of the Architectural Decision Guidance Tool implementation

Go Libraries and Test Suite

The implementation relies on a focused set of Go libraries that support command-line interfaces, configuration management, file parsing, and testing. The two main libraries used for building the CLI are `cobra` [19] and `viper` [20]. Cobra provides a structured and extensible way to define commands, flags, and help documentation, making it ideal for a feature-rich CLI application like ADG. Viper complements Cobra by handling configuration files, environment variables, and default settings, enabling user-level customization without complicating the CLI logic.

For metadata serialization and deserialization, `yaml.v3` [21] is used to parse decision metadata and configuration files, chosen for its compatibility with YAML 1.2 [22] and its robust handling of nested data structures.

The testing suite is based on `testify` [23], which provides a fluent assertion library and support for mock generation. To isolate tests and validate inter-layer communication, the project also uses `mockery` [24] to automatically generate mocks for interfaces. This enables precise unit testing of interactors and domain logic without relying on file system access or I/O. Together, these tools support a high level of test coverage and make it easier to refactor or extend the system confidently.

Maintaining and Extending the Tool

The tool is designed to be modular and extensible, making it straightforward to introduce new commands or behaviors. Each command follows a structured pattern based on the Clean Architecture layering. To add a new command, the developer typically begins by creating a new interactor that encapsulates the use case logic. This interactor is defined as an implementation of an `inputport` interface and internally delegates to services from the domain layer. Next, a corresponding printer is implemented as part of the `outputport`, which formats the result for terminal output. This printer is passed into the interactor to decouple the application logic from presentation formatting. A new command is then created in the adapter layer using Cobra, where the user input is parsed and passed to the interactor through the defined input port. Finally, the new command is wired into the application via the `cmd` package. This is where dependency injection occurs, binding together the interactor, printer, and any required services such as model or decision handling. Optionally, if the new use case involves specialized domain logic or external integration, the developer can extend the `domain` or `infrastructure` layers with new services or data types.

This layered design makes it easy to isolate changes, maintain clean abstractions, and test each component independently. It also provides a clear onboarding path for contributors, since each component has a specific responsibility and location in the project structure.

3.5 Available Commands

The ADG tool provides a wide range of commands designed to support the complete lifecycle of architectural decision modeling. An overview can be seen in Table 2. These commands were developed in alignment with the user roles and use cases identified earlier, ensuring that each role can effectively carry out their tasks using the tool. Each command is equipped with contextual help via the `--help` flag, enabling discoverability and reducing the learning curve. By combining a rich command set with thoughtful metadata integration, the ADG tool provides a robust foundation for structured, collaborative architecture documentation.

Command	Description	Flags
`add`	Adds one or more decision points to a model	`-model, -title`
`comment`	Add a comment to a decision	`-id, -model, -author, -text`
`copy`	Copies a model, optionally a subset based on filters	`-model, -target, -id, -tag, -status, -title`
`decide`	Marks a decision as decided by selecting one of its options	`-id, -model, -option, -rationale, -author, -force`
`edit`	Updates the content or metadata of an existing decision file by modifying sections such as the question, criteria, or options	`-id, -model, -question, -criteria, -option`
`import`	Imports a decision model into an existing model	`-model, -target, -id, -tag, -status, -title`
`init`	Initializes a new decision model directory and index file	
`link`	Link two decisions using optional custom tags or default logic	`-from, -to, -model, -tag, -reverse-tag`
`list`	Lists decisions with optional filters	`-model, -id, -tag, -status, -title, -format`
`merge`	Merges two models into a new one	`-model1, -model2, -target, -id, -tag, -status, -title`
`rebuild`	Rebuilds the index file from existing decisions	`-model`

Command	Description	Flags
`reset-config`	Reset all configuration (or just template headers)	`-template`
`revise`	Create a new revision of a decision, set to `open`	`-id, -model`
`set-config`	Set default configuration values	`-model, -author, -question, -options, -criteria, -outcome, -comments, -template, -config-path`
`tag`	Adds tags to a decision	`-id, -model, -tag`
`validate`	Validates if decision files and index match	`-model`
`view`	Displays specific sections of a decision based on filters provided through section-specific flags	`-id, -model, -question, -options, -criteria, -outcome, -comments`

Table 2: Overview of ADG CLI commands with descriptions and available flags.

4 Usage Scenario

To illustrate the practical application of the Architectural Decision Guidance (ADG) tool, this section walks through the setup of the tool, followed by a tutorial demonstrating the tool's capabilities throughout the decision-making workflow.

4.1 Setup

The ADG tool can be used either by executing one of the available pre-compiled binaries or by building the executable from source. This setup flexibility allows it to be deployed across a range of development environments.

Pre-Compiled Binaries. The project repository includes platform-specific binaries targeting common operating systems:

```
adg/
+- adg_linux        # Linux (x86_64)
+- adg_mac_arm      # macOS (Apple Silicon)
+- adg_mac_intel    # macOS (Intel)
+- adg_win.exe      # Windows
```

To execute the tool, the appropriate binary for the respective platform may be run directly from the `adg/` directory. For example, on Linux systems, the following command displays the tool's help interface:

```
./adg_linux --help
```

In Unix-like environments, executable permissions may not be set by default. This can be resolved by applying the `chmod` command:

```
chmod +x adg_linux    % or the binary matching the target system
```

In cases where the provided binary is incompatible or fails to execute, the tool may be compiled from source.

Building from Source. The ADG tool is implemented in Go, and manual compilation requires a Go version 1.20 or higher. When building from source, the working directory is expected to be the root of the `adg/` project. Compilation can be triggered with the following command:

```
go build -o adg main.go
```

This will result in a binary named `adg` in the same directory. On Windows platforms, renaming the output to `adg.exe` may be necessary for execution. Functionality can be verified by running:

```
./adg --help
```

Once setup has been completed successfully, the tool is ready for use in modeling, validating, and maintaining architectural decision records. The next subsection provides a structured walkthrough of the workflow.

4.2 Workflow Example

To demonstrate the practical use of the ADG tool in real-world development contexts, this section introduces two typical usage modes: a lightweight, file-centric workflow that emphasizes flexibility and minimal tooling overhead, and a full-featured, command-driven workflow tailored for advanced users and larger teams. These examples showcase how the tool can be adapted to varying levels of expertise, collaboration styles, and architectural governance needs.

Basic Workflow: Lightweight File-Based Usage

In the simplest usage scenario, a single user or a small team may rely primarily on the ADG tool to initialize the model and manage structural consistency, while performing content editing and decision reasoning manually via a text editor. This approach is ideal for users who value full transparency and want to work directly with the Markdown files while leveraging the tool's indexing and validation features when needed.

1. **Model Initialization:** The user begins by creating a new model directory using the `init` command:

   ```
   ./adg init --model ./my-model
   ```

2. **Adding Decisions:** One or more open decisions are created using the `add` command with user-defined titles:

   ```
   ./adg add --model ./my-model --title "Choose Logging Framework"
   ```

3. **Editing in a File Editor:** Instead of using the CLI to modify content, users open the generated Markdown files in their preferred text editor (e.g., VS Code, Vim, Sublime) and manually fill in the sections such as *Question*, *Options*, and *Criteria*. Since all decision files are human-readable, no additional tooling is required.

4. **Maintaining Consistency:** If metadata is manually edited or decision files are renamed or deleted outside of the CLI, the model index may become inconsistent. In such cases, users can run the following to regenerate the index from scratch:

   ```
   ./adg rebuild --model ./my-model
   ```

This workflow emphasizes user autonomy and integrates naturally with traditional developer tools, making it an excellent entry point for newcomers or teams transitioning from unstructured documentation to formalized decision tracking.

Advanced Workflow: Command-Driven Power Usage

More advanced users, such as software architects or knowledge engineers managing reusable models across multiple projects, can take advantage of the full feature set offered by the ADG CLI. This includes not only basic operations but also advanced functionality such as model merging, tagging, linking, and decision revision. In this mode, the entire decision lifecycle is handled within the tool, promoting structure, consistency, and traceability.

1. **Creating and Reusing Guidance Models:** Users initialize or import structured decision models using commands like `init`, `copy`, `import`, or `merge`, facilitating model reuse and team alignment across projects.
2. **Structured Editing and Decision Making:** Instead of manually editing the decision files in a text editor as explained in the basic workflow, decisions can be modified directly via CLI flags using `edit` and `decide`, which support precise, metadata-driven updates. Rationale is provided through flags like `-rationale`.
3. **Decision Lifecycle Management:** Commands such as `revise` and `tag` allow users to reopen previously decided entries or group decisions thematically. The `link` command supports creating semantic and chronological connections between decisions, enabling navigation through complex decision graphs.
4. **Validation and Feedback:** The `list` and `view` commands help navigating the model's content and allow filtered inspection of decisions based on metadata such as tags, status, or section type. Users can also leave comments through the `comment` command to foster collaboration.

To further increase productivity, the CLI can generate an autocompletion script for various shell environments, such as Bash, PowerShell, Fish and Zsh. This feature allows users to press the `Tab` key while typing commands to automatically complete them. For example, typing `ed` and pressing `Tab` will suggest or complete the `edit` command. Pressing `tab` without anything typed will cycle through all the available commands. As the tool is distributed as a standalone executable, advanced users can also create custom scripts that leverage its commands. For instance, during testing, a PowerShell script was developed that uses the tool to generate a new test model, populate it with a predefined number of decisions containing randomized content and tags (e.g., 1000), and link each decision sequentially to the next.

This CLI-user workflow is particularly well-suited for long-term architecture governance, cross-team knowledge sharing, and consistent application of recurring patterns. It emphasizes automation, traceability, and structured reuse while still offering flexibility when needed.

These two usage modes illustrate the spectrum of workflows supported by the ADG tool, from minimal, file-driven interaction to full-scale decision model engineering. Teams may freely switch between modes depending on their maturity level, tooling preferences, or collaboration needs.

5 Empirical Evaluation

To assess the usability, clarity, and practical value of the ADG tool, an empirical evaluation involving two participants was conducted. The evaluation simulated realistic decision-modeling workflows and captured qualitative feedback based on actual tool usage. The goal was not to conduct a large-scale user study but rather to gather focused insights from representative users under controlled conditions. The evaluation followed a structured format. Each participant was given access to the tool, two pre-configured guidance models, and a task description. Participants were asked to perform a sequence of operations using the tool and document their experience. After completing each sub-task, the participants answered a few questions regarding the usability, learnability, and overall quality of the tool. This section outlines the setup and briefly describes the tasks performed. It also presents and analyzes the results. See Appendix A for the full task description.

Two participants from different backgrounds were involved in the evaluation. The first was a professor and software architect with experience in decision modeling who had used earlier versions of the ADG tool. The second was a graduate student with no prior experience with the tool. Together, they represented two perspectives: an experienced decision modeler and a first-time user encountering the tool in a practical context.

5.1 Setup and Task Description

Each participant was provided with the following materials:

- A compiled binary of the ADG tool for their respective operating system.
- A prefilled guidance model containing decisions based on microservice and event-driven architecture.
- An instruction document outlining a structured task designed to mimic a realistic architectural decision-making scenario.

The setup required no external dependencies or installations, enabling participants to run the binary in a terminal environment. Participants were asked to follow the instructions at their own pace using only the tool's built-in help system (`--help`)

and their interpretation of the task file. No external guidance or training was provided. The goal of the evaluation was not to measure performance or completeness, but rather to explore how well the ADG tool supports typical workflows, and to identify areas of confusion, friction, or room for improvement.

The evaluation was organized as a structured hands-on exercise simulating the typical workflow of a software architect interacting with a reusable decision model. Participants worked through eight sequential subtasks, each focusing on a specific capability of the ADG tool:

1. **Explore the Model:** Participants listed and viewed decisions from an existing guidance model to understand its structure and metadata organization.
2. **Make a Decision:** Participants completed several open decisions using different approaches: selecting a predefined option, justifying the rationale, or entering a custom option. This demonstrated how decisions evolve from open to decided states and how rationale is documented.
3. **Add and Edit a Decision:** Participants added a new decision to the model and modified its contents using both the CLI and a text editor, highlighting the dual editing modes supported by the tool.
4. **Tag a Decision:** Participants labeled decisions with tags and explored how filtering by tag enhances model navigation and organization.
5. **Link Decisions:** Participants created both chronological and custom semantic links between decisions, exploring the decision graph capabilities and cycle detection mechanisms.
6. **Comment on a Decision:** Participants provided feedback by adding comments to existing decisions, simulating collaboration and stakeholder interaction.
7. **Validate the Model:** Participants manually introduced inconsistencies in metadata and content, then used the validation command to detect and resolve issues, illustrating the tool's robustness against manual edits.
8. **Import Another Model:** Participants imported decisions from a separate guidance model, reinforcing concepts of model reuse and highlighting the differences between manual copying and CLI-based import operations.

Each subtask was followed by reflective questions that prompted users to assess usability, clarity, and their overall experience with the ADG tool. Participants were given 60 minutes total to complete the task, with a 10-minute break in between. After completing the task, participants submitted their answers and their command line output.

5.2 Results and Feedback

The experiment was successful in that both participants were able to interact with the ADG tool effectively with minor difficulties. The overall structure of the tool, the metadata model, and the command-line interface were generally perceived as accessible and well-aligned with the participants' expectations, particularly for those familiar with developer tooling and terminal workflows. All task were completed without external help. This indicates that the tool offers a coherent user experience for its target audience. Nonetheless, several valuable insights and improvement opportunities emerged from participants' written responses and the concluding discussion.

During the structured tasks, participants encountered occasional issues with command-line syntax and flag usage. While the tool's built-in `--help` functionality was appreciated and frequently used, misunderstandings about expected flag values were a recurring challenge. For instance, certain flags such as `--rationale` or `--format` were occasionally omitted or unclear, which led to avoidable errors. Another point of confusion was the repeatability of certain flags such as `--option`. To add multiple options in a single command, each option must be explicitly preceded by the `--option` flag. However, participants intuitively expected that they can only add one option and simply need to execute the command multiple times, once for each option. On the other hand, certain commands can handle multiple inputs with only one flag using comma-separated values. For example, `--id "0001,0002"`. In this case, the apostrophes are crucial for the command to work. The participants did not expect this and also perceived it as inconsistent because this flag handles repeated values differently than the previous one.

Participants also highlighted the cognitive gap between command semantics and flag naming. One notable example was the `--force` flag used to record decisions with options that were not predefined. Although functionally correct, one participant would prefer a more intuitive alternative such as `--create`. Similarly, participants suggested creating short forms of flags, such as `-r` for `--rationale`. One participant also noted that the `--model` flag was often required with the same

value and found it cumbersome to type for every command. While it is possible to configure this flag with a default value, this option was unclear to the participant. The help text for this flag mentions configuration but never explains how to set it up. Additionally, the tool could provide help output when the flag is not configured, informing the user of this option. Alternatively, it could provide an optional tutorial command or an introductory guide in the repository to help users become familiar with all available features.

One participant noted frequent typing errors and expressed the need to carefully review their input for mistakes. In such cases, setting up the autocompletion script could significantly improve the user experience by reducing typos and streamlining command entry. Another participant preferred reusing previous commands via the CLI history, changing only the relevant parts. However, this sometimes leads to mistakes where old, incompatible flags are used for new commands. Again, setting up autocompletion might encourage users to avoid using the history and prevent these types of mistakes. Additionally, the input validation messages were sometimes too brief, making it difficult to understand how to resolve command failures. Future versions of the tool could improve usability by providing more contextual guidance and enhancing error messages when flags are misused or missing.

In terms of editing workflow, both participants expressed a preference of using a hybrid approach. While the CLI is effective for creating and modifying metadata, text editing, especially involving multi-line rationale or criteria, was viewed as more comfortable and error-proof when done in a dedicated editor like VS Code. Feedback from the final discussion session reinforced the positive impressions of the tool's conceptual model and structure. Participants praised the use of reusable decision models and agreed that the tool facilitates consistency and traceability across architectural decisions. The command structure was considered logical, and the learning curve was manageable for technical users. Overall, the evaluation confirmed that the ADG tool supports its intended use cases effectively and is accessible to both experienced and novice users. The insights gathered point toward several opportunities for refinement, particularly in the areas of usability, terminology, and feature discoverability. These findings will inform future iterations of the tool, especially with regard to improved documentation, richer feedback mechanisms.

6 Clean Architecture Guidance Model

The ADG tool was used to recreate and expand upon the Clean Architecture decision handbook developed during the previous project thesis [2]. In that earlier work, eight recurring architectural decisions were analyzed and documented using the Nygard ADR format, forming the initial set of decisions numbered 1 to 8. Each decision contained placeholder text to guide users in tailoring the content to their specific context. With the introduction of the ADG tool, the content of the original handbook's decisions was migrated into a structured model using the tool's format. During this process, the original decisions were preserved and rewritten as open decisions with prefilled content, including common options and evaluation criteria. Seven new decisions numbered 9 to 15 were added to enhance the guidance model and expand its coverage of additional concerns relevant to Clean Architecture, such as error handling and testing strategies. The complete model functions as a reusable, extensible framework that provides guidance to software architects and teams applying Clean Architecture principles to their projects. The following subsections outline the newly added decisions 9 through 15.

6.1 Defining Use Case Interfaces

A fundamental aspect of Clean Architecture is enforcing strict boundaries between architectural layers, especially between the use case layer and outer layers, such as interface adapters or frameworks. Decision 9 outlines how to implement these boundaries in practice by defining the interface contracts that govern interaction across layers. Three main approaches were considered. The first option strictly adheres to the Clean Architecture principle by allowing the use case layer to define abstract interfaces (e.g., IUserRepository), which the outer layers then implement. This approach ensures full adherence to the dependency inversion principle. Another approach is to place the interfaces in a shared contract layer (or the domain layer) that is technically decoupled from both the use case layer and the infrastructure layer. However, this approach weakens architectural clarity. The third option, which lets outer layers expose concrete APIs and makes use cases depend on them, violates the inversion principle and undermines testability and independence.

Using the ADG tool, this decision was added to the Clean Architecture guidance model as a continuation of the earlier decision on use case selection (Decision 3). It establishes the technical boundary between internal logic and external interfaces. It promotes maintainability, facilitates mocking in tests, and reinforces architectural intent through explicit ownership of interface contracts.

```
---
adr_id: "0009"
title: define-usecase-interfaces
status: open
tags:
    - interfaces
    - dependency-inversion
    - use-cases
    - clean-architecture
links:
    precedes:
        - "0010"
    succeeds:
        - "0003"
comments: []
---
```

 Question

How should the interface contracts between the use case layer and the outer layers (e.g., interface adapters, frameworks) be defined to support dependency inversion and layer independence?

 Options

1. Let the use case layer define abstract interfaces that are implemented by external layers (e.g., ‘UserRepository‘ interface).

2. Define interfaces in a shared contract layer decoupled from both use case and infrastructure.

3. Allow outer layers to define concrete APIs and let use cases depend directly on them (inversion not enforced).

 Criteria

- Clarity of architectural boundaries and ownership
- Degree of adherence to dependency inversion principle
- Ease of mocking or substituting components in tests
- Maintainability and discoverability of contracts
  across modules

6.2 Defining an Error Handling Strategy

Robust and consistent error handling is essential for building reliable systems, especially when adhering to the layered constraints of the Clean Architecture model. Decision 10 focuses on defining an appropriate strategy for propagating and handling errors across architectural layers. The goal is to maintain separation of concerns and avoid framework-specific dependencies in the core business logic. The preferred option is to use explicit return values, such as error objects at every layer boundary. This approach prevents uncontrolled panics or exceptions and aligns well with Clean Architecture principles. A second, more relaxed option permits exceptions or panics in the outer layers (e.g., frameworks) but requires catching and translating them into controlled forms before they reach the inner layers. The third alternative permits exception propagation across all layers and relies on a global handler to map exceptions to user-facing responses. However, this increases coupling and violates the core tenets of Clean Architecture.

This decision builds on the previous interface design decision (Decision 9) by clarifying how communication between layers should behave in failure scenarios. The selected strategy must ensure that errors remain transparent, testable, and traceable without introducing transport or infrastructure concerns to business logic.

```
---
adr_id: "0010"
title: define-error-handling-strategy
status: open
tags:
    - error-handling
    - exceptions
    - contracts
    - clean-architecture
links:
    precedes:
        - "0011"
    succeeds:
        - "0009"
comments: []
---
```

 Question

What strategy should be used for handling and propagating errors across architectural layers in a way that preserves separation of concerns and keeps business logic independent of frameworks?

 Options

1. Use return types (e.g., error objects or ‘Result‘ types) at every boundary, avoiding exceptions or panic flows.

2. Allow exceptions or panics in outer layers, but catch and translate them into controlled forms before reaching core layers.

3. Propagate exceptions across all layers with a global handler that maps them to user-facing responses.

 Criteria

- Transparency and consistency of error behavior
- Coupling introduced between layers through exception types
- Ease of testing and mocking error paths
- Ability to localize and translate error messages for clients
- Support for logging, auditing, and observability

6.3 Managing Cross-Cutting Concerns

Cross-cutting concerns, such as logging, monitoring, authentication, and authorization, are essential to any non-trivial system, yet they pose a challenge to architectural clarity when applied inconsistently or too deeply within business logic. Decision 11 addresses how to integrate these concerns in a way that respects the separation of layers and responsibilities dictated by Clean Architecture. The most architecturally sound approach is to apply middleware or decorator patterns in the outer layers (e.g., interface adapters). This ensures that concerns such as logging and authentication can be handled consistently without leaking into the core use case logic. A more dynamic alternative is to use aspect-oriented programming, such as interceptors or annotations, to inject cross-cutting behavior. While powerful, this approach can reduce transparency and make control flow harder to trace. The least desirable option is to embed concern-specific logic directly into use cases or infrastructure code, as this increases coupling, making the system harder to test and maintain.

This decision builds upon the earlier error-handling strategy (Decision 10) by addressing another form of behavior that cuts across layers. The selected approach must balance modularity, maintainability, traceability, and runtime efficiency.

```
---
adr_id: "0011"
title: manage-cross-cutting-concerns
status: open
tags:
    - "cross-cutting"
    - "logging"
    - "monitoring"
    - "auth"
    - "clean-architecture"
links:
    precedes:
        - "0012"
    succeeds:
        - "0010"
comments: []
---
```

```
## <a name="question"></a> Question

How should cross-cutting concerns such as logging, monitoring,
or authentication be handled across layers while preserving
the architectural integrity of the system?

## <a name="options"></a> Options

1. <a name="option-1"></a> Use middleware or decorator patterns
in the outer layers (e.g., interface adapters) to inject
cross-cutting logic.

2. <a name="option-2"></a> Apply aspect-oriented techniques
(e.g., interceptors, annotations) to weave in concerns
dynamically.

3. <a name="option-3"></a> Embed cross-cutting logic directly
into use cases and infrastructure code where needed.

## <a name="criteria"></a> Criteria

- Degree of decoupling and modularity
- Ease of applying concerns uniformly across layers
- Testability of use cases without concern-side effects
- Visibility and traceability of system behavior
    (e.g., logging context)
- Performance and overhead introduced by concern injection
```

6.4 Defining a Testing Strategy Across Layers

Testing is critical for ensuring the correctness and maintainability of a system. In the context of Clean Architecture, each layer has distinct responsibilities and should be tested both in isolation and in coordination with the others. Decision 12 focuses on selecting an appropriate testing strategy that respects architectural boundaries while providing sufficient coverage and feedback. The first option emphasizes testing each architectural layer in isolation, particularly with unit tests for use cases and entities, and uses integration tests only at adapter boundaries. This adheres to the separation of concerns principle. Another option emphasizes full-stack integration tests to validate end-to-end behavior. This approach simplifies test design, but it may also blur architectural boundaries and increase test fragility. The last option proposes a hybrid strategy that combines focused unit tests for core business logic with integration tests across workflows.

Building on prior structural choices, such as interface boundaries (Decision 9) and error handling (Decision 10), this decision defines how these abstractions are exercised and validated in practice. The strategy must balance test speed, maintainability, and coverage while enabling long-term architectural resilience.

```
---
adr_id: "0012"
title: define-testing-strategy
status: open
tags:
    - "testing"
    - "unit-test"
    - "integration-test"
    - "architecture-layers"
links:
    precedes:
        - "0013"
    succeeds:
        - "0011"
comments: []
---

## <a name="question"></a> Question

What testing strategy should be used for the different
```

architectural layers to ensure correctness while maintaining separation of concerns?

 Options

1. Write isolated unit tests per layer (e.g., use cases, entities) with integration tests only at adapter boundaries.

2. Focus on full-stack integration tests covering end-to-end behavior, with minimal unit testing.

3. Apply a hybrid strategy combining unit tests for critical logic and integration tests for workflows.

 Criteria

- Test execution speed and feedback cycle
- Test coverage of business-critical paths
- Isolation of architectural concerns during testing
- Maintenance effort and fragility of test suites
- Tooling and framework support for mocking and assertions

6.5 Defining a Dependency Injection Strategy

Maintaining a modular and testable architecture requires careful management of how components are wired together and how dependencies are passed across layers. Decision 13 addresses the selection of a dependency injection (DI) strategy that supports inversion of control without sacrificing maintainability or clarity. Option 1 proposes manual dependency injection, in which components are explicitly connected during application startup. This approach maximizes transparency and control, but it can become verbose as the application grows. Option 2 introduces a lightweight DI framework or container to automate wiring. This improves modularity and runtime flexibility, but introduces external tooling and abstractions. Option 3 suggests using global state or service locators to provide access to shared services. This reduces boilerplate, but it introduces tight coupling and hidden dependencies, which contradicts Clean Architectural principles.

This decision affects testability, as defined in Decision 12, and builds directly on the structural boundaries defined earlier in the model. A well-chosen dependency injection (DI) approach enables easier mocking, clearer control flow, and greater maintainability across architectural boundaries.

```
---
adr_id: "0013"
title: define-dependency-injection-strategy
status: open
tags:
    - "dependency-injection"
    - "configuration"
    - "modularity"
    - "clean-architecture"
links:
    precedes:
        - "0014"
    succeeds:
        - "0012"
comments: []
---

## <a name="question"></a> Question

How should dependencies be managed and injected across
```

architectural layers to support inversion of control, testability, and modular design?

 Options

1. Use manual dependency injection by explicitly wiring components during application startup.

2. Use a lightweight dependency injection container or framework.

3. Inject dependencies via global variables or service locators accessible throughout the application.

 Criteria

- Clarity and visibility of wiring logic
- Adherence to inversion of control principle
- Ease of testing and substituting mocks or stubs
- Runtime flexibility and configurability
- Tooling support and learning curve

6.6 Structuring Use Case Modules

As software systems become more complex, the internal structure of the use case layer is essential for ensuring maintainability and scalability. Decision 14 explores strategies for organizing use case modules to promote modularity, navigability, and alignment with business logic. One option, suggests organizing use cases by feature. In this approach, each business capability (e.g., user registration or invoice generation) is isolated in its own subdirectory. This approach encourages cohesion and traceability within features, but it can result in duplication across modules. Option 2 advocates grouping use cases by technical role or function. For example, all use cases or interfaces could be placed together. While this approach simplifies infrastructure concerns, it can also blur the boundaries between business contexts, hindering modular development. Option 3 proposes using domain-driven design principles, specifically aggregates or bounded contexts, to cluster related logic. This enhances conceptual alignment, but it can increase the initial design overhead.

This decision is tied to previous choices, such as dependency injection (Decision 13) and the test strategy (Decision 12), since grouping use cases affects both wiring and test isolation. A well-structured use case layer reinforces architectural separation while supporting future system development.


```
---
adr_id: "0014"
title: structure-usecase-modules
status: open
tags:
    - "modularity"
    - "use-cases"
    - "feature-modules"
    - "organization"
links:
    precedes:
        - "0015"
    succeeds:
        - "0013"
comments: []
---
```

 Question

How should feature modules be structured within the use case layer to maintain clarity, modularity, and scalability of the application?

 Options

1. Organize use cases by feature verticals (e.g., ‘user/register‘, ‘invoice/generate‘) with dedicated subdirectories per feature.

2. Group use cases by technical function (e.g., all use cases, all interfaces) across the application.

3. Follow domain-driven design aggregates or bounded contexts to group related business capabilities.

 Criteria

- Discoverability and navigation of use case logic
- Encapsulation and independence of features or domains
- Reusability and testability of logic across modules
- Scalability as the application grows
- Alignment with business concepts and responsibilities

6.7 Defining Input/Output Boundaries

A key design principle of Clean Architecture is the separation of concerns between the application core and the external world. Decision 15 addresses how to structure input/output boundaries to maintain clean separation while supporting flexibility and maintainability.

Option 1 recommends creating explicit input and output interfaces in the adapter layer. These interfaces translate requests and responses (e.g., HTTP, CLI, or messaging formats) into internal models that the use case layer can understand. This maintains a clear boundary and supports protocol independence. However, option 2 relaxes this separation, allowing use case handlers to directly consume and produce external types (e.g., HTTP request structs or CLI flags). While this simplifies implementation, it introduces tight coupling between the application core and the transport layer, which undermines the principles of Clean Architecture. option 3 proposes a middleware-style boundary where incoming data is transformed and validated in a preprocessing pipeline before reaching the core logic. This approach can provide flexible extension points, but it may also increase complexity and reduce transparency. This decision is based on previous decisions, such as selecting adapter patterns (Decision 4) and defining use case interfaces (Decision 9). Together, these decisions shape how external requests flow through the system while preserving the architectural integrity of each layer.


```
---
adr_id: "0015"
title: define-io-boundary-strategy
status: open
tags:
    - boundaries
    - io
    - interface-adapter
    - clean-architecture
links:
    precedes: []
    succeeds:
        - "0004"
        - "0009"
        - "0010"
comments: []
---
```

 Question

How should the boundaries between the application core and the outside world (e.g., HTTP, CLI, gRPC, messaging systems) be structured to maintain layer separation and support adaptability?

 Options

1. Define explicit input and output interfaces in the interface adapter layer that map external requests/responses to internal models.

2. Embed parsing, formatting, and transport logic directly into use case handlers (e.g., use case accepts HTTP request types).

3. Use a middleware pipeline that transforms all I/O at the application boundary before passing to the use case.

 Criteria

- Separation of concerns between layers
- Reusability of core business logic across channels
- Testability of input/output handling
- Flexibility to support multiple protocols or transport formats (e.g., JSON, Protobuf)
- Simplicity and maintainability of adapters

7 Conclusion

This thesis introduces the Architectural Decision Guidance (ADG) tool. ADG is a command-line application designed to support the structured, repeatable decision-making process in software development projects. Section 1 introduces and outlines the structure of the thesis.

Section 2 reviewed the existing literature and tools related to architecture decision records (ADRs), architecture modeling, and decision guidance. Building from reserach of a previous thesis, a lack of tools that bridge the gap between formal modeling approaches and the everyday workflows of developers and architects was identified.

Section 3 presented the design and implementation of the ADG tool in detail. The section introduced the three user roles (*Knowledge Engineer*, *Model Tailor*, and *Software Architect/Developer*) and their associated use cases, which were illustrated with diagrams and a walkthrough example. Based on these roles, a set of non-functional requirements was derived using the FURPS model to address usability, reliability, performance, and supportability. The section explained the tool's metadata model and indexing mechanism, including the rationale for separating file-level metadata from the index file to enable efficient lookups and validation. The software architecture, which is implemented in Go and structured according to Clean Architecture principles, was described in depth. This description included how the architecture enables modularity, portability, and extensibility. Section 3.5 reviewed the command set, providing a comprehensive table documenting the available flags and features for each command.

Section 4 illustrated the practical usage of the tool through a two-level workflow example. The example showed novice users how to engage with the tool using a limited set of commands and demonstrated how advanced users can leverage its full set of structured capabilities for editing, linking, and validating models in a CLI-centric workflow. Setup instructions and execution strategies were also provided to facilitate adoption.

The practical usability of the tool was assessed in Section 5 through an empirical evaluation involving two users. The participants carried out a guided task that covered typical ADG operations, such as model exploration, decision-making, tagging, linking, commenting, and validation. The results revealed several areas for improvement and refinement. Users encountered challenges with repeated flag usage (e.g., adding multiple options) and expressed a need for more informative

error messages and help texts. Setting up shell auto-completion was also identified as a valuable feature for reducing input errors and increasing efficiency.

Section 6 describes how the tool was used to implement and extend a reusable guidance model for Clean Architecture. This section showed how a set of existing ADRs from previous work were migrated to the ADG format, enriched with metadata, and expanded with seven new decisions that address additional architectural concerns. The result was a structured, extensible model with fifteen decisions that provides a foundation for project-specific tailoring and reuse.

Although the ADG tool covers a wide range of functionality for decision modeling and reuse, the evaluation identified several areas for improvement. These include clarifying the behavior of repeatable flags, improving error messages for invalid inputs, and providing more intuitive feedback during interactions. Autocompletion support and clearer help texts were also recognized as valuable additions to reduce the learning curve and support efficient use. In parallel, several additional extensions were identified during development and documented in the project backlog. These include quality-of-life features such as commands for creating multiple decisions at once, issuing a decision immediately upon creation, or chaining commands in a single terminal expression. Suggestions for parameters, such as model names or available decision IDs, could make the tool more interactive and reduce the need for memorization. Structural improvements include being able to add decisions in subfolders within a model (instead of having to move the files manually) and adopting a custom URI schema to assign globally unique and referenceable decision identifiers. Furthermore, a dedicated command could support converting existing ADRs in the Nygard format into ADG-compatible form by adding necessary anchors and metadata. Together, these enhancements would significantly increase the usability of the ADG tool.

In summary, this thesis demonstrated that decision guidance can be effectively implemented in a lightweight, command-line tool by combining structured metadata, a modular architecture, and user-centric workflows. The ADG tool lays the groundwork for improving architectural documentation practices in real-world software development and provides clear opportunities for future development. Currently, the code is hosted on the university's private GitLab instance; however, it is planned to be released as open source on GitHub in the near future to encourage wider adoption, community contributions, and long-term sustainability.

References

[1] ADR GitHub Organization, "Architectural Decision Records," accessed: June 9, 2025. [Online]. Available: https://adr.github.io

[2] R. Schellander, "Concept Alternatives for the Management of Architectural Decisions in Clean Architectures," 2024.

[3] O. Zimmermann, "Architectural Decisions — The Making Of," 2021, accessed: June 9, 2025. [Online]. Available: https://www.ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html

[4] ADR GitHub Organization, "ADR Templates," accessed: June 9, 2025. [Online]. Available: https://adr.github.io/adr-templates/

[5] M. Nygard, "Documenting Architecture Decisions," 2011, accessed: June 9, 2025. [Online]. Available: https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions

[6] ADR GitHub Organization, "Markdown Architectural Decision Records," accessed: September 6, 2024. [Online]. Available: https://adr.github.io/madr/

[7] O. Zimmermann, "Y-Statements," 2021, accessed: September 6, 2024. [Online]. Available: https://medium.com/olzzio/y-statements-10eb07b5a177

[8] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, 2005.

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

[10] U. Heesch, P. Avgeriou, and R. Hilliard, "A Documentation Framework for Architecture Decisions," *The Journal of Systems & Software*, 2012.

[11] ——, "Forces on Architecture Decisions – A Viewpoint," 2012.

[12] V.-P. Eloranta, U. Heesch, P. Avgeriou, N. Harrison, and K. Koskimies, "Lightweight Evaluation of Software Architecture Decisions," *Relating System Quality and Software Architecture*, 2014.

[13] J. Van der Ven, A. Jansen, P. Avgeriou, and D. K. Hammer, "Using architectural decisions," 2006.

[14] A. Jansen, J. Van der Ven, P. Avgeriou, and D. K. Hammer, "Tool Support for Architectural Decisions," *WICSA*, 2007.

[15] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, "N.: Reusable Architectural Decision Models for Enterprise Application Development," *Third International Conference on the Quality of Software-Architectures (QoSA 2007), pp. 15-32*, 2007.

[16] O. Zimmermann, "An architectural decision modeling framework for service oriented architecture design," 2009.

[17] N. Schuster, O. Zimmermann, and C. Pautasso, "ADkwik: Web 2.0 Collaboration System for Architectural Decision Engineering." *19th International Conference on Software Engineering and Knowledge Engineering, SEKE 2007*, 2007.

[18] A. MacLean, R. Young, V. Bellotti, and T. Moran, "Questions, Options, and Criteria: Elements of Design Space Analysis," *Human-Computer Interaction*, 1991.

[19] S. Francia, "Cobra: A Framework for Modern CLI Apps in Go," accessed: September 6, 2024. [Online]. Available: https://cobra.dev/,https://github.com/spf13/cobra

[20] ——, "Viper: Go configuration with fangs," accessed: September 6, 2024. [Online]. Available: https://github.com/spf13/viper

[21] go yaml, "YAML support for the Go language. ," accessed: September 6, 2024. [Online]. Available: https://github.com/go-yaml/yaml

[22] C. Evans, "YAML Ain't Markup Language," accessed: September 6, 2024. [Online]. Available: https://yaml.org/

[23] Stretch, Inc., "A toolkit with common assertions and mocks that plays nicely with the standard library," accessed: September 6, 2024. [Online]. Available: https://www.gotestify.com/,https://github.com/stretchr/testify

[24] Vektra, "A mock code autogenerator for Go ," accessed: September 6, 2024. [Online]. Available: https://vektra.github.io/mockery/latest/

A Task Description of Empirical Evaluation

Overview

You are a *Software Architect* working on an internal platform that manages a landscape of microservices. Your organization has a dedicated *Knowledge Engineer* whose role is to create reusable "guidance models" for recurring architectural decisions. One such model already exists for typical decisions in microservice-based systems. However, your organization does *not* have a dedicated *Model Tailor*. This role is typically responsible for selecting and adapting relevant guidance models to create a project-specific decision set that Software Architects and Developers can use to make concrete decisions. Therefore, you will also take on the responsibilities of the Model Tailor. Your task is to *explore, extend, and tailor the existing model* to fit your specific project needs, and to *make architectural decisions* using the ADG CLI tool.

Learning goals

- Familiarize yourself with the structure of a decision model
- Use the CLI to navigate, view, and make decisions
- Add new decision templates and structure them meaningfully
- Tag, link, and comment on decisions
- Validate your decision model for consistency
- Reuse content by importing decisions from another model

Each subtask is designed to highlight a specific part of the CLI. If something is unclear:

- Run help for any command (replace 'adg' with your binary):

```
./adg <command> -h
```

- Get general command overview:

```
./adg -h
```

- Or ask the experiment supervisor (Raphael) if you are stuck.

  If you have used this tool before and set a custom configuration, run the following command so that all participants have the same setup (otherwise skip this step):

```
./adg reset-config
```

1. Explore the model

1.1 List the existing decisions

As the *Model Tailor*, your first step is to get an overview of the existing decisions in the microservices guidance model. This helps you understand the current scope, status, and categorization of the available decision templates.

Run the following command to list all decisions:

```
./adg list --model ./models/microservices
```

- How many decisions are present?
- What information does each entry provide?
- Is there any additional metadata you would find useful?
- You can change the format of the output using the '–format' flag. What additional information is available with the other formats that the default (simple) output does not include (e.g., 'md' format)?

1.2 View a specific decision

You are particularly interested in reviewing the details of a specific decision. You can look it up by either its *ID* (e.g., ‘0001‘) or its *title* (e.g., ‘"service-communication"‘).

Examples:

```
./adg view --model ./models/microservices --id 0001
```

```
./adg view --model ./models/microservices
           --id  "service-communication"
```

- What is this decision about?
- What content is shown by default?
- What happens when you add the ‘–question‘ flag?

1.3 Copy the model to a working directory

After reviewing the model, you decide that it is a good starting point for your project. You now copy the model into your working directory ‘./adr‘:

```
./adg copy --model ./models/microservices --target ./adr
```

2. Make a decision

2.1 Complete an open decision

As a *Software Architect* you now want to *mark* an open decision as decided based on your current architectural direction. Start by reviewing the proposed option in decision ‘0001‘. You can either open the file directly in a text editor such as *Visual Studio Code* or use the CLI:

```
./adg view --model ./adr --id 0001 --options
```

Review the available options and pick one that fits your needs. You can select an option by its number or its exact title:

```
./adg decide --model ./adr --id 0001 --option 2
```

```
./adg decide --model ./adr --id 0001 --option "gRPC"
```

- What changed in the file compared to the original in './models/microservices'?
- What happens if you try the same command on this decision again?
- Try the suggested 'revise' command. What happens?

2.2 Make a decision with an explanation

You now turn to decision '0002', but this time you want to *justify* your selection. Look at the available flags of the 'decide' command and use the correct one to provide your reasoning when making the decision.

- Which flag allows you to justify your decision?
- Where is this rationale recorded?

2.3 Make a decision with a custom option

Next, you review decision '0003', but none of the listed options seem appropriate for your use case. You decide to add your own option, for example using Kubernetes DNS.

Inspect the available options first:

```
./adg view --model ./adr --id 0003 --options
```

Now try deciding with your own custom option (replace ‘<your-new-option>‘ with your actual idea):

```
./adg decide --model ./adr --id 0003 --option "<your-new-option>"
```

You may encounter an error here.

- Why does this command fail?
- Try the same command again, this time explicitly confirming that you’re aware the option is not predefined:

```
./adg decide --model ./adr --id 0003 --option "<your-new-option>"
             --force
```

- What is the effect of using ‘–force‘?
- How do you feel about this safeguard mechanism?

3. Add and edit a new decision

You notice that an important architectural choice has not yet been captured in the model. As the responsible *Model Tailor*, you want to *add a new decision* to document this missing aspect.

Think of a relevant decision for your project, for example, choosing a monitoring solution, deciding on a testing framework, or selecting an API versioning strategy.

Create a new decision template with a meaningful title (replace ‘<your-new-decision>‘ with your actual idea):

```
./adg add --model ./adr --title "<your-new-decision>"
```

This command generates a new Markdown file with a unique ID and the provided title. The file includes a metadata header, followed by empty sections for ‘Question‘, ‘Options‘, and ‘Criteria‘.

Now fill the sections of this decision. You can use the CLI to edit it by ID or title. Replace ‘<id/title>‘, ‘<section>‘, and ‘<section-content>‘ as needed:

```
./adg edit --model ./adr --id <id/title> --<section> <section-content>
```

Alternatively, open the newly created file in a text editor (e.g., Visual Studio Code) and edit the content manually.

- Was the process of adding and editing a decision intuitive?
- What happens when you add an option via the ‘edit‘ command? How is the outcome different from editing the question or criteria sections?
- Which workflow do you prefer: using the CLI or editing the file directly? Why?
- (Optional) Try deciding the newly added decision using ‘adg decide‘. Did everything work as expected?

4. Tag a decision

You want to label your newly added decision with a tag so that it can be grouped and filtered more easily later on, for example, by topic, concern, or architectural layer.

Use the following command (replace placeholders appropriately):

```
./adg tag --model ./adr --id <id/title> --tag <your-tag>
```

- What changed in the file? Where is the tag stored?
- Label another decision (e.g., ‘0002‘) with the same tag.
- What command would you use to *list all decisions with this specific tag*?

5. Link decisions

5.1 Create a chronological link

You realize that your newly added decision is dependent on another. To reflect this in the model, you want to *indicate that your decision should be made after another.*

Inspect the other decisions in your model and identify one that logically follow this kind of relationship. Then link it to your new decision by ID or title (replace placeholders accordingly):

```
./adg link --model ./adr --from <id/title> --to <id/title>
```

- What changed in both files, where can you see that the two decisions are now linked?
- Which default labels are used to describe the relationship between the '–from' and '–to' decision?
- Would you have used different labels to describe the relationship?
- What happens if you try to link decision from '0003' to '0001'?
- Why is this not allowed in your model?

5.2 Create a custom link

You now encounter a situation where *a newly introduced decision renders an earlier decision obsolete or invalid.* This can happen, for instance, when architectural assumptions change or a new standard replaces an old approach.

To reflect this in your model, first create a new decision using the 'add' command. It doesn't need to be fully filled out, just give it a meaningful title that makes the conflict or contradiction clear (e.g., "Deprecate API Gateway").

```
./adg add --model ./adr --title <title>
```

Then, create a custom link between the new decision and the one it invalidates. Use the '–tag' and '–reverse-tag' flags to define appropriate labels:

```
./adg link --model ./adr --from <id/title> --to <id/title>
           --tag <from-tag> --reverse-tag <to-tag>
```

Use '-h' if you need help understanding how these flags work.

- Which existing decision does your new one invalidate?
- Which label did you use for '–tag' and which for '–reverse-tag'?
- How is this different from the chronological links you created earlier?
- Where do these tags show up in the files?

6. Comment on a decision

As you inspect decision '0004' ('monitoring'), you realize that its listed criteria are incomplete. For example, while it mentions "easy integration," it neglects aspects such as *alerting support*, *dashboarding capabilities*, or *data retention*, all of which could significantly impact the choice of monitoring tool.

As the *Model Tailor*, you decide to add a comment to the original model to point this out and give feedback to the Knowledge Engineer who created this decision.

Run the following command and include a short, specific suggestion (replace placeholder):

```
./adg comment --model ./models/microservices --id 0004
              --text "<your-feedback>"
```

- Assuming you have not made custom configurations, what error do you get?

Run the command again with the suggested flag.

- What changed in the file after the comment was added?
- What did your comment include?
- What role does the '–author' flag play in documenting feedback?
- If you wanted to *actually fix* the decision and add the missing criteria, which command would you use?

7. Validate the model

As your model evolves and grows, it's critical to ensure *consistency between metadata and content*. The CLI provides a 'validate' command to detect mismatches or structural problems.

In your working directory './adr', you will find a file named 'index.yaml'.

- Open this file and examine its contents.
- What kind of information is stored here?
- How does this index relate to the decision files?

7.1 Manually modify metadata

Let's simulate a common mistake: manually editing metadata without updating the index. This could lead to inconsistencies that the tool needs to detect.

Open the decision file where you previously added a tag. Then, *replace the tag with an empty bracket*:

From this:

```
tags:
  - testing
```

To this:

```
tags: []
```

Now run the validation:

```
./adg validate --model ./adr
```

- What does the output of this command say?
- Are you able to identify which file the error refers to?
- Try running the suggested fix. What does it do? (tip: use '-h' to get more information)
- Do you think manually editing the metadata (e.g., tags, links) should be allowed, or should this only be done through the CLI?

7.2 Remove a section and validate again

Now simulate a more structural issue: a user accidentally deletes part of the decision content.

Open any decision file and *remove the full 'Question' section*:

```
### <a name="question"></a> Question

The question of this decision
```

Then re-run the validation:

```
./adg validate --model ./adr
```

- What error does the tool report?

- How would you fix this issue and restore the missing section?
- Could you use a CLI command to restore or re-edit the section instead of manually fixing the file?

8. Import another model

You are now exploring how to expand your microservices model to support *event-driven messaging patterns*. The decisions in your current model no longer fully capture the architectural scope you are working with. Fortunately, your organization's Knowledge Engineer has also curated a guidance model ‘event-driven‘.

Start by exploring the existing guidance model to get an overview of the relevant decisions:

```
./adg list --model ./models/event-driven
```

Pick one or two decisions that would make sense to reuse, then import the decisions into your working model:

```
./adg import --model ./adr --source ./models/event-driven --id <id1,id2>
```

Questions:

- What changed with the imported decisions?
- Are titles and metadata (tags, links, comments) preserved?
- How are the IDs of the imported decisions handled during import?
- Why would importing through the CLI be more reliable than manually copying files?

Try manually copying another decision from ‘./models/event-driven‘ into your ‘./adr‘ folder and run:

```
./adg validate --model ./adr
```

What happens? What's the difference compared to using ‘import‘?

Conclusion

You have now explored the full range of core functionality offered by the ADG CLI tool. Starting from an existing decision model, you:

- Navigated and understood architectural decisions
- Tailored the model by making and editing decisions
- Labeled and structured decisions with tags and links
- Validated model consistency using metadata and section structure checks
- And finally, reused architectural knowledge by importing from external guidance models.

Before moving on to task 2, please copy and save your entire terminal output from this task in a file.

Once you're done, you may clear your terminal or open a new instance. Then proceed to Task 2, where you'll create your own decision model from scratch in a more exploratory and open-ended setting.