# An Architecture-Aware Flutter Code Generation Framework for Enforcing Clean Architecture and Behavioral Design Patterns

Desy Intan Permatasari[1]*, Adam Shidqul Aziz[2], Maulidan Bagus Afridian Rasyid[3],
Ahmad Jarir At-Thobari[4], Umi Sa'adah[5], Nailussa'ada[6], Muhammad Fajrul Falah Subakti[7]
Informatics and Computer Engineering Department, Politeknik Elektronika Negeri Surabaya, Surabaya, Indonesia[1, 2, 5, 6, 7]
SimHive Group, Surabaya, Indonesia[3]
AgileTeknik Network, Surabaya, Indonesia[4]

***Abstract*—Large-scale Flutter applications can accumulate direct inter-module dependencies, inconsistent responsibility boundaries, and inflexible data-access implementations as their functionality grows. Conventional scaffolding approaches primarily automate project structures and repetitive code, while architectural relationships commonly remain dependent on developer-level implementation decisions. This study proposes an architecture-aware Flutter code generation framework that transforms JavaScript Object Notation (JSON)-based application specifications into modular source code while systematically applying Clean Architecture, Mediator, and Strategy rules. The main contribution lies not in combining these established architectural concepts, but in encoding their architectural roles as repeatable specification-to-code transformation rules. The framework was evaluated using three application specifications containing one, two, and eight functional modules. All three applications were successfully generated, executed, and built into installable Android APKs. In the evaluated Login and Register interaction, two direct cross-module relationships observed in the legacy architecture were replaced by mediator-based communication, resulting in no direct Login and Register reference in the generated structure. The generated architecture also represented REST API and SQLite data providers through a common Strategy-based abstraction. These findings demonstrate operational feasibility and provide structural evidence that the intended architectural rules were incorporated into the evaluated generated artifacts. However, the evaluation is limited to the supported JSON schema, predefined generation templates, three application specifications, and the evaluated architectural relationships; therefore, the results should not be interpreted as general evidence of long-term maintainability, scalability, or superiority over other Flutter code generators.**

***Keywords*—Architecture-aware code generation; specification-driven development; flutter; clean architecture; mediator pattern; strategy pattern; software modularity***

## I. Introduction

Flutter has emerged as a widely adopted framework for cross-platform mobile application development because it enables applications for multiple platforms to be developed from a shared codebase while supporting efficient development processes and near-native performance [1]. However, as mobile applications grow in functional scope and architectural complexity, maintaining a consistent source-code structure becomes increasingly challenging. Large-scale applications may accumulate direct dependencies among functional modules, unclear responsibility boundaries, and dependency-related problems that increase the difficulty of software maintenance and evolution [2], [3].

Clean Architecture addresses these challenges by separating software responsibilities into distinct layers and directing dependencies toward stable abstractions [4]. Previous studies have shown that its application can improve responsibility separation, source-code organization, and maintainability in mobile software projects [5], [6]. Nevertheless, Clean Architecture primarily defines responsibility boundaries and dependency directions. It does not by itself prescribe a uniform mechanism for coordinating communication among functional modules or for representing interchangeable operational behaviors, such as alternative data-access mechanisms.

Behavioral design patterns provide complementary mechanisms for addressing these concerns. The Mediator pattern centralizes communication among participating components and reduces the need for direct references between them [7]. Its implementation in software systems has demonstrated how component interactions can be coordinated through an intermediary rather than through direct object-to-object dependencies [8]. The Strategy pattern encapsulates alternative algorithms or behaviors behind a common interface, allowing a concrete implementation to be selected or replaced without modifying its client [9]. In a modular Flutter application, these patterns can support controlled inter-module communication and flexible selection of data providers, including REST application programming interface (API) and SQLite implementations.

Existing Flutter studies have generally applied Clean Architecture manually within application-specific projects. For example, Clean Architecture has been implemented in a Flutter-based tourism application by manually defining the presentation, domain, and data structures required by the application [10].

Related software-engineering research has also emphasized that reducing dependencies among software components is

*Corresponding author.

important for architectural scalability and long-term system evolution [11], while separation of concerns contributes to the management of software complexity [12]. A reusable Flutter Clean Architecture package has additionally been proposed to provide predefined architectural components and assist developers in separating business logic from platform-specific implementation [13].

Although these approaches support architectural adoption, they still require developers to define functional modules, configure their relationships, implement inter-module communication, select data-access mechanisms, and maintain architectural consistency at the source-code level. Reusable packages and manually implemented architectures can reduce initial development effort, but they do not necessarily ensure that the same architectural constraints are applied consistently across all modules and application versions. Similarly, conventional scaffolding tools may generate folders, classes, and repetitive source-code artifacts without systematically enforcing the relationships among the generated components.

Within the literature reviewed in this study, limited attention has been given to transforming declarative Flutter application specifications into multi-module applications while systematically enforcing responsibility separation, mediator-based inter-module communication, and strategy-based data-provider variation during source-code generation. The research gap addressed in this study is therefore not the absence of Clean Architecture, Mediator, or Strategy as individual software-engineering concepts, since these concepts are already well established. Rather, the unresolved issue concerns their systematic enforcement as part of an automated specification-to-code generation process. Without such enforcement, architectural conformance remains dependent on developer expertise and local implementation decisions, potentially resulting in inconsistent dependency structures and architectural drift.

To address this gap, this study proposes an architecture-aware Flutter code generation framework that receives application specifications in JavaScript Object Notation (JSON) format and transforms them into modular Flutter source code. The JSON input serves as a blueprint that defines application-level properties, pages, routes, functional modules, and selected data providers. During generation, the framework applies Clean Architecture rules to organize software responsibilities, Mediator rules to coordinate communication among functional modules, and Strategy rules to represent REST API and SQLite data providers through interchangeable abstractions.

The principal novelty of the proposed framework does not lie merely in combining Clean Architecture with the Mediator and Strategy patterns. Its scientific contribution lies in encoding these architectural decisions as repeatable specification-to-code generation rules. Consequently, responsibility boundaries, communication mechanisms, and data-provider abstractions are established during source-code generation rather than being left entirely to post-generation manual implementation. This mechanism is intended to produce consistent architectural structures across modules with different functional responsibilities.

This study makes three main contributions. First, it defines a JSON-based specification-to-architecture mapping that transforms application properties, pages, routes, functional modules, and data-provider configurations into organized Flutter source-code artifacts. Second, it implements an automated generation mechanism that systematically applies Clean Architecture boundaries, mediator-based inter-module communication, and strategy-based data-provider variation during source-code generation. Third, it provides an empirical feasibility and structural evaluation using three application specifications with increasing modular complexity, including operational generation and build observations, a legacy–generated architectural comparison, and inspection of the generated Mediator and Strategy structures.

The evaluation examines whether the framework can successfully generate, execute, and build Flutter applications from the tested JSON specifications. Structural architectural characteristics are assessed by comparing the available legacy and generated application structures, examining direct functional-module relationships, and inspecting whether the generated source structure incorporates the intended Mediator and Strategy mechanisms. The results are interpreted as evidence of operational feasibility and architectural conformance within the evaluated scenarios. They are not considered complete proof of long-term maintainability, scalability, or testability because these broader quality attributes require larger application samples, repeated experiments, additional software-quality measurements, and longitudinal maintenance evaluation.

The remainder of this study is organized as follows. Section II critically reviews previous studies on Clean Architecture, behavioral design patterns, Flutter architectural approaches, code generation, and software-quality evaluation. Section III describes the proposed framework, JSON specification, architectural generation rules, experimental scenarios, and evaluation procedure. Section IV presents and discusses the operational and structural evaluation results. Section V discusses the threats to validity and limitations of the study. Finally, Section VI presents the conclusions and directions for future research.

## II. Related Work

Clean Architecture has been applied in mobile software development to improve separation of concerns and control dependency directions. Rachovski et al. [4] used Clean Architecture principles to organize the design and implementation of a mobile platform, while Santiago-Salazar and Rico-Bautista [5] examined its impact on the performance and maintainability of native Android applications. In the Flutter context, Sinatria et al. [6] implemented Clean Architecture by separating presentation, domain, and data responsibilities. These studies indicate that layered architectural separation can clarify responsibility boundaries and improve source-code organization. However, the architectural structures in these studies were implemented manually within individual application projects, meaning that architectural consistency remained dependent on developer knowledge and implementation decisions.

Behavioral design patterns provide complementary mechanisms for managing software interactions and behavioral variation. The Mediator pattern centralizes communication among participating components, thereby reducing the need for direct references between them [7]. Petrova and Varbanov [8] demonstrated mediator-based communication and command handling using the MediatR library in a .NET environment. Meanwhile, the Strategy pattern encapsulates alternative algorithms or behaviors behind a common interface, allowing an implementation to be selected or replaced without modifying its client [9]. Although these studies establish the architectural roles of Mediator and Strategy, the patterns are generally applied manually at the source-code level. Consequently, their implementation may vary among modules, developers, and application versions.

Wijayanto and Sejati [10] applied Clean Architecture to a Flutter-based mobile tourism application and reported that the separation of presentation, domain, and data responsibilities supported application modularity and development. Nevertheless, the architectural components and their relationships were manually defined for the specific application rather than generated from a declarative specification. In a broader software-architecture context, Aljaloud and Razzaq [11] demonstrated the importance of reducing dependencies when restructuring monolithic software systems, while Idate et al. [12] emphasized separation of concerns as a mechanism for managing software complexity. These studies reinforce the importance of modularity and dependency management but do not provide an automated mechanism for generating modular Flutter applications.

A more directly relevant reusable solution was introduced by Boukhary and Colmenares [13] through the Flutter Clean Architecture Package. The package provides reusable architectural components that help developers separate business logic from platform-specific implementation. This approach reduces the effort required to establish a Clean Architecture structure in Flutter projects. However, a reusable architectural package differs from an architecture-aware code generator. Developers must still define the functional modules, configure dependencies, implement communication among modules, and determine how alternative data providers are integrated. Therefore, the package supports architectural implementation but does not automatically transform an application specification into a complete multi-module Flutter project with generated Mediator and Strategy structures.

Stirgwolt et al. [14] demonstrated that modular architectures can support system evolution by controlling relationships among components. Sangwan et al. [15] further emphasized that architectural dependency analysis is necessary because uncontrolled dependencies may increase change propagation and complicate software evolution. Sun et al. [16] investigated the relationship between dependency injection and software maintainability, demonstrating the relevance of dependency management to software-quality assessment.

Recent model-driven software-engineering studies provide a broader perspective on systematic and architecture-aware software generation. Lano et al. [17] proposed an agile model-driven re-engineering approach that uses software abstraction, semantic models, transformation, refactoring, and rearchitecting to support legacy-system modernization and platform migration. Their evaluation included practical re-engineering cases involving legacy systems and comparison of program-translation effectiveness, demonstrating how explicit model-driven transformations can support automated restructuring and target-code production.

Jiménez-Navajas et al. [18] proposed a model-to-text code-generation technique in which extended Unified Modeling Language (UML) design models are transformed into executable source code. Their approach uses explicit Model-to-Text transformations to generate Python and Qiskit code for classical-quantum systems and was evaluated through a multi-case study involving seven software systems. Although the target domain differs from Flutter mobile applications, the study provides a relevant example of specification-driven generation in which transformation rules and generated artifacts are explicitly defined and empirically evaluated.

Recent studies published in applied computer-science venues provide additional perspectives relevant to the present work. Tiali et al. [19] employed a model-driven engineering approach in which explicit transformation rules systematically map business-process representations into generated executable smart-contract artifacts. Although their source and target technologies differ from the present study, their work illustrates the importance of making transformation rules explicit when automated generation is claimed. In the mobile-software context, Yataco et al. [20] demonstrated the practical use of Clean Architecture in an implemented mobile system, showing the continuing relevance of responsibility separation and modular architectural organization in application development. From a software-quality perspective, Abufakher [21] empirically investigated GoF design patterns using multiple object-oriented quality metrics, illustrating the importance of explicitly defined measurement procedures when quantitative structural metrics are reported.

These studies strengthen the methodological positioning of the present work because they demonstrate that model- or specification-driven software generation can encode explicit transformation rules rather than relying exclusively on manual implementation. However, none of these studies addresses the generation of modular Flutter applications in which Clean Architecture organization, mediator-based inter-module communication, and strategy-based data-provider variation are jointly incorporated into the generated source structure.

Collectively, these studies demonstrate that architectural quality can be examined from multiple structural perspectives, including cohesion, coupling, modular organization, and dependency relationships. They also indicate that individual static metrics capture only selected characteristics of a software architecture and should not be interpreted as complete measures of long-term maintainability, scalability, testability, or developer effort. Accordingly, the present study focuses its empirical structural evaluation on directly observable architectural relationships in the evaluated artifacts, particularly direct functional-module dependencies and the generated Mediator and Strategy structures.

Table I compares representative related approaches according to their target platforms, input mechanisms, architectural support, levels of automation, evaluation methods, and remaining limitations. The comparison distinguishes manually implemented Clean Architecture applications, reusable architecture packages, model-driven generation approaches, and the architecture-aware Flutter code generation framework proposed in this study.

As shown in Table I, existing Flutter studies mainly apply Clean Architecture manually or provide reusable architectural components whose final configuration remains developer-controlled. In contrast, recent model-driven approaches [17], [18] demonstrate that explicit software representations can be transformed systematically into generated artifacts. The proposed framework combines these two directions by using a declarative JSON specification to generate Flutter source code while incorporating predefined Clean Architecture organization, Mediator-based inter-module communication, and Strategy-based data-provider variation.

Nevertheless, Table I should be interpreted as a capability-level comparison rather than a direct performance benchmark. The reviewed studies differ in platform, input representation, transformation objectives, generated technologies, and evaluation procedures. Therefore, the empirical evaluation in this study uses the available legacy application as an internal structural baseline, while a controlled comparison with functionally equivalent external Flutter generators remains future work.

TABLE I. COMPARISON OF RELATED ARCHITECTURAL APPROACHES

| Study | Platform / Input | Architectural Support | Automation Level | Evaluation | Remaining Limitation |
|---|---|---|---|---|---|
| Wijayanto and Sejati [10] | Flutter; manually implemented requirements | Clean Architecture for modular mobile | Manual application-specific implementation | Functional application case study | Not generated from declarative specification |
| Boukhary and Colmenares [13] | Flutter; reusable package and developer configuration | Reusable Clean Architecture components | Package-assisted; assembly remains developer-controlled | Package demonstration/application implementation | No complete specification-driven multi-module generation |
| Lano et al. [17] | Legacy software and semantic models | Refactoring, restructuring, rearchitecting, migration | Automated/model-driven transformation | Real-world legacy cases and translation evaluation | Different target problem/platform; no Flutter Mediator/Strategy generation |
| Jiménez-Navajas et al. [18] | Extended UML models | Model-driven source-code generation | Automated Model-to-Text transformation | Seven-system multi-case study | Different target domain/output technology; no Flutter architecture-pattern enforcement |
| Proposed framework | Flutter; JSON application specifications | Clean Architecture organization, Mediator communication, Strategy data-provider variation | Automated specification-to-code generation | One-, two-, and eight-module scenarios; execution/build validation; structural evaluation | Limited to supported JSON schema, templates, modules, and evaluated scenarios |

## III. METHODOLOGY

### A. Framework Overview and Architecture

This study evaluates a Clean Code Generator developed as part of the CleanApp environment. The generator receives an application specification in JSON format and produces Flutter source-code artifacts using predefined generation templates. The JSON configuration describes the structure and functional elements of the target application, while the generation mechanism incorporates architectural structures based on Clean Architecture, Mediator, and Strategy. The resulting Flutter project is subsequently executed, built into an Android application package (APK), and inspected through operational and structural architectural evaluation.

### B. JSON-Based Application Specification

The Clean Code Generator receives its input as a JSON configuration produced by the CleanApp Builder. The configuration describes the application structure to be generated, including application-level information, page definitions, navigation routes, data-provider configurations, optional features, and application branding information. The JSON specification therefore serves as a declarative representation of the target application, while the generator translates the supplied configuration into the corresponding Flutter source-code structure.

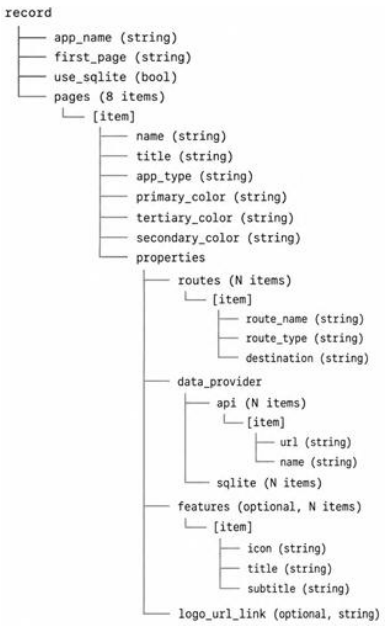


Fig. 1. Structure of the JSON application.

As shown in Fig. 1, the specification contains three application-level elements: app_name, first_page, and use_sqlite. The pages collection contains the definitions of the

functional pages included in the generated application. Each page contains descriptive attributes, including its name, title, application type, and interface colors. The properties element contains the configuration required to determine navigation, data access, optional features, and application branding.

Navigation is represented through the routes collection, in which each route contains route_name, route_type, and destination. Data-access requirements are represented by the data_provider element, which can contain API- and SQLite-based configurations. An API entry contains a url and name, while the SQLite configuration can contain multiple entries depending on the application specification.

The high-complexity configuration can additionally contain optional features, consisting of icon, title, and subtitle, as well as an optional logo_url_link.

TABLE II. JSON SPECIFICATION CHARACTERISTICS OF THE EXPERIMENTAL SCENARIOS

| Configuration element | Low Complexity | Medium Complexity | High Complexity |
|---|---|---|---|
| pages | 1 item | 2 items | 8 items |
| routes | 2 items | N items | N items |
| data_provider.api | 0 items | 0 items | N items |
| data_provider.sqlite | 0 items | 0 items | N items |
| features | Not shown | Not shown | Optional, N items |
| logo_url_link | Present | Not shown in displayed schema | Optional |

Note: N denotes a variable number of items in the displayed JSON schema rather than a fixed experimental count.

The number of elements varies according to the application scenario. Therefore, the JSON specification does not directly encode a fixed application implementation; instead, it defines the application elements that must be interpreted by the generator. The generator subsequently applies its predefined architectural rules and source-code templates to produce the corresponding Flutter project.

Table II highlights the differences among the JSON structures used in the three experimental scenarios. The low-complexity configuration contains a single page and two navigation routes, whereas the medium-complexity configuration contains two pages and supports a variable number of routes. In both displayed configurations, the API and SQLite provider collections contain no items. The high-complexity specification extends this structure to eight pages and supports multiple routes, API and SQLite entries, optional feature definitions, and an optional logo URL. Thus, experimental scenarios increase not only in the number of generated functional components but also in the richness of the configuration processed by the generator.

### C. Architecture-Aware Code Generation Mechanism

The proposed framework transforms the JSON specification into Flutter source-code artifacts using predefined code templates and architectural rules. Clean Architecture provides the structural organization of the generated application, while the Mediator and Strategy patterns define specific relationships among generated components. The Mediator mechanism is used to coordinate communication among functional modules without requiring direct module-to-module references, whereas the Strategy mechanism is used to organize alternative data-access implementations behind a common abstraction. Thus, the generator does not merely produce project folders and repetitive source code; it also embeds predefined architectural relationships into the generated project.

In the generated architecture, each functional module contains a controller that acts as the principal coordination point for the corresponding module. Interactions that involve other functional modules are routed through the generated mediator structure, while data-access alternatives such as REST API and SQLite are represented through Strategy-based components. This mechanism allows the same architectural conventions to be repeatedly applied to different application specifications and functional modules.

The architecture-aware generation mechanism can be summarized as a set of specification-to-code transformation rules. These rules determine how application elements and architectural conditions are translated into generated source-code structures. Table III summarizes the principal rules implemented by the framework and the architectural structures produced by each rule.

TABLE III. SPECIFICATION-TO-CODE ARCHITECTURAL GENERATION RULES

| Specification or Architectural Condition | Generation Rule | Generated Structure / Architectural Effect |
|---|---|---|
| Functional page or module definition | Generate the corresponding functional-module source structure, including its controller, using the predefined architectural templates | Module responsibilities are organized according to the predefined Clean Architecture structure |
| Interaction involving different functional modules | Route cross-module communication through the generated Mediator component rather than generating direct module-to-module references | Mediator-based inter-module coordination |
| REST API data-provider configuration | Generate the API-specific data-provider implementation behind the common data-provider abstraction | Strategy-based representation of REST API access |
| SQLite data-provider configuration | Generate the SQLite-specific data-provider implementation behind the same data-provider abstraction | Strategy-based representation of SQLite access |

In this study, architecture-aware generation refers to a code-generation process in which predefined architectural rules are applied during the transformation from the JSON specification to source-code artifacts, rather than generating only syntactic scaffolding. Architectural enforcement refers to the generation-time instantiation of these selected rules in the produced source structure, so that the corresponding architectural decisions are not left entirely to post-generation developer implementation. Architectural conformance is operationally defined as the observable agreement between the generated artifacts and the specific architectural rules evaluated in this study, namely the predefined module organization, Mediator-based routing of the evaluated cross-module interaction, and Strategy-based representation of REST API

and SQLite data providers. This definition is limited to the inspected architectural relationships and does not constitute formal verification of all Clean Architecture dependency constraints.

These rules distinguish the proposed framework from conventional scaffolding that primarily generates repetitive project artifacts. In the proposed approach, the generator also determines selected architectural relationships during source-code generation. Consequently, module organization, cross-module communication, and data-provider variation are instantiated according to predefined architectural rules rather than being left entirely to post-generation manual implementation.

### D. Code Generation Workflow

Fig. 2 summarizes the specification-to-code generation and validation workflow. The code-generation process begins when the Clean Code Generator receives a JSON configuration produced by the CleanApp Builder.

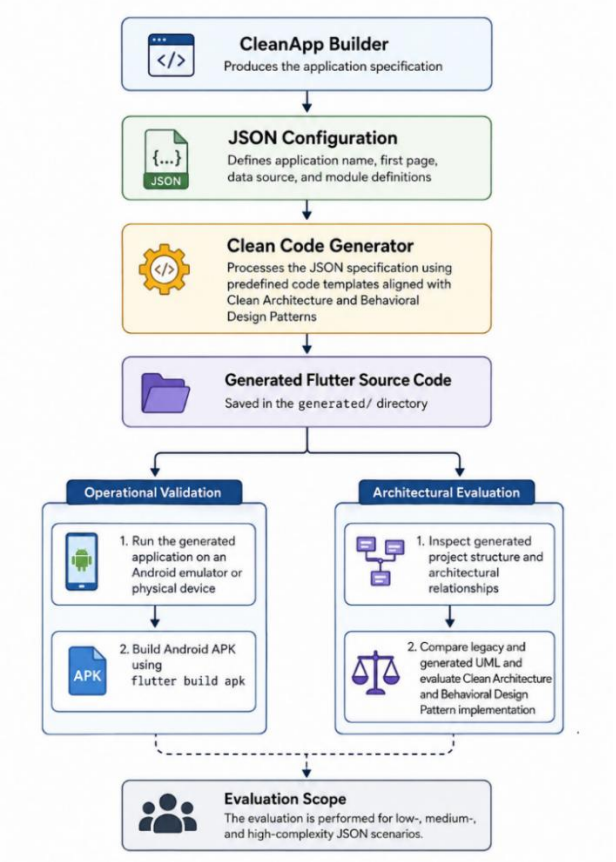


Fig. 2. Workflow of the specification-to-code generation and validation process.

The framework first reads the application-level configuration and identifies the pages, navigation routes, functional modules, and data-provider requirements defined in the specification. The corresponding source-code templates are then selected and used to generate the required Flutter project artifacts. During this process, the predefined architectural structures for Clean Architecture, Mediator-based communication, and Strategy-based data access are incorporated into the generated source code.

After generation, the resulting files are organized into a standalone Flutter project. The generated project is subsequently executed in the development environment or on an Android device to verify that the application can run successfully. Finally, the project is built into an Android APK and its source-code structure is inspected for architectural evaluation. This workflow therefore covers the complete process from declarative application specification to generated, executable, and structurally evaluated Flutter application.

### E. Experimental Design and Environment

The framework was evaluated using three JSON-based application specifications representing increasing levels of functional complexity. The low-complexity scenario contained one functional module implementing a registration form. The medium-complexity scenario contained two functional modules, Setting and User. The high-complexity scenario incorporated all functional modules available in the evaluated framework configuration. These three scenarios were selected to examine whether the same generation mechanism could process application specifications with different numbers of functional components while preserving the intended project structure.

Table IV summarizes the evaluated scenarios, including their relative configuration scale, number of functional modules, and functional scope.

TABLE IV. EXPERIMENTAL SCENARIOS

| Scenario | Scenario Scale | Modules | Functional Scope |
|---|---|---|---|
| S1 | Low | 1 | Registration |
| S2 | Medium | 2 | Setting and User |
| S3 | High | 8 | Login, E-commerce List, E-commerce Square, Note, Onboarding, Register, Settings, and User |

The experiments were conducted from August 2024 to June 2025. The primary development machine was equipped with an 11th-generation Intel Core i5 processor, 16 GB of RAM, and a 512 GB SSD. The software environment consisted of Flutter Software Development Kit (SDK) 3.24.5, Dart SDK 3.5.4, Android Studio Jellyfish 2023.3.1, and Visual Studio Code. The generated applications were additionally validated using two Android devices: one device equipped with an Exynos 9611 chipset, 4 GB of RAM, and Android 10, and another equipped with a MediaTek Dimensity 700 chipset, 8 GB of RAM, and Android 13.

In this experiment, the terms low, medium, and high complexity are used as descriptive scenario labels reflecting the increasing number of functional modules and the relative richness of the corresponding JSON specifications. They should not be interpreted as formal measurements of software complexity, dependency density, interaction complexity, generated-code volume, architectural depth, or cyclomatic complexity. The three scenarios were selected to examine whether the same generation mechanism could process

specifications with different numbers of functional components while preserving the intended project structure.

### F. Evaluation Procedure

The evaluation consisted of two complementary stages: operational validation and structural architectural evaluation. Operational validation examined whether each JSON specification could be successfully processed into Flutter source code, whether the generated project contained the expected directory and source-code structure, whether the application could be executed on the experimental environment, and whether an Android APK could be successfully built. Generation time, APK build time, and APK size were recorded as descriptive operational characteristics.

Structural architectural evaluation examined whether the generated source code exhibited the intended architectural relationships. The generated application structure was represented using UML class diagrams and compared with the available legacy application architecture. The comparison focused on direct relationships between functional modules and on whether the generated application routed cross-module communication through the Mediator component. The generated Strategy structure was also inspected to verify the representation of REST API and SQLite data-provider implementations through a common abstraction.

For the structural comparison, a direct functional-module dependency was counted when a class belonging to one functional module directly referenced a class belonging to another functional module without an intermediary architectural abstraction. Dependencies routed through the generated Mediator abstraction were treated as mediated relationships and were therefore not counted as direct functional-module dependencies. The comparison was restricted to the evaluated Login–Register interaction for which corresponding structural evidence was available in both the legacy and generated architectures.

Accordingly, the structural evaluation focuses on architectural relationships that can be directly inspected from the available generated and legacy artifacts. The evaluation does not attempt to provide a comprehensive quantitative assessment of software maintainability or overall architectural quality.

## IV. Results and Discussion

### A. Framework Validation

The operational validation was conducted using three application-generation scenarios with different levels of functional complexity. The scenarios consisted of low-complexity applications with a single module, medium-complexity applications with two modules, and high-complexity applications containing all available functional modules. For each scenario, the framework was required to successfully generate source code and produce a buildable Flutter application.

Operational success in this evaluation indicates that the generated projects could be produced, executed, and built successfully under the tested configurations. However, successful execution and APK construction do not by themselves establish functional correctness with respect to every behavioral requirement represented in the JSON specifications. A systematic requirements-to-behavior verification was outside the scope of the present evaluation.

TABLE V. Framework Validation

| Complexity | Modules | Code Generation | Application Execution | APK Build |
|---|---|---|---|---|
| Low | 1 | Successful | Successful | Successful |
| Medium | 2 | Successful | Successful | Successful |
| High | 8 | Successful | Successful | Successful |

Table V summarizes the operational validation results. All three evaluated specifications were successfully generated, executed, and built into Android APKs. Because each scenario was evaluated once, these observations demonstrate operational feasibility within the tested configurations rather than a generalized reliability rate.

### B. Structural Comparison of Legacy and Generated Architectures

The legacy and generated application structures were compared to examine differences in direct functional-module relationships after the introduction of mediator-based coordination. The comparison focuses on the evaluated Login–Register interaction, for which corresponding structural evidence is available in both architectures.

TABLE VI. Structural Comparison of the Evaluated Legacy and Generated Interaction

| Structural Aspect | Legacy Architecture | Generated Architecture |
|---|---|---|
| Evaluated interaction | Login ↔ Register | Login ↔ Register through Mediator |
| Direct functional-module dependencies | 2 directed relationships | 0 direct relationships |
| Inter-module coordination | Direct cross-module references | Mediator-based coordination |
| Architectural interpretation | Modules directly reference one another | Direct references replaced by mediated communication |

Table VI summarizes the structural comparison of the evaluated Login–Register interaction. In the legacy architecture, two directed cross-module relationships were observed between the corresponding components. In the generated architecture, no direct Login–Register relationship was observed because the interaction is routed through the Mediator component. Thus, the direct functional-module dependency count decreases from two to zero for this evaluated interaction. This result provides structural evidence of reduced direct coupling, although the generated modules remain dependent on the Mediator abstraction.

TABLE VII. Operational Measurements

| Complexity | Modules | Generation Time (ms) | Build Time (s) | APK Size (MB) |
|---|---|---|---|---|
| Low | 1 | 1,311 | 38.3 | 49.0 |
| Medium | 2 | 1,351 | 302.5 | 49.8 |
| High | 8 | 1,474 | 249.7 | 65.8 |

Table VII presents the descriptive operational measurements of the three evaluated specifications. Generation time increased from 1,311 ms for the single-module specification to 1,474 ms for the eight-module specification. APK size increased from 49.0 MB to 65.8 MB across the same configurations. In contrast, build time did not exhibit a monotonic relationship with the number of modules, with recorded values of 38.3 s, 302.5 s, and 249.7 s. Because each specification was evaluated once, these measurements are reported descriptively and are not interpreted as statistically validated evidence of runtime or build scalability.

The non-monotonic build-time pattern cannot be attributed to configuration scale from the present measurements alone. Potential contributing factors may include build-system state, dependency resolution, incremental-build or caching conditions, and background execution variability. Because these factors were not independently controlled or recorded during the experiment, the observed differences are treated as descriptive tooling and execution variability rather than evidence of a systematic relationship between module count and build time. Repeated measurements under controlled build conditions are required to distinguish these effects.

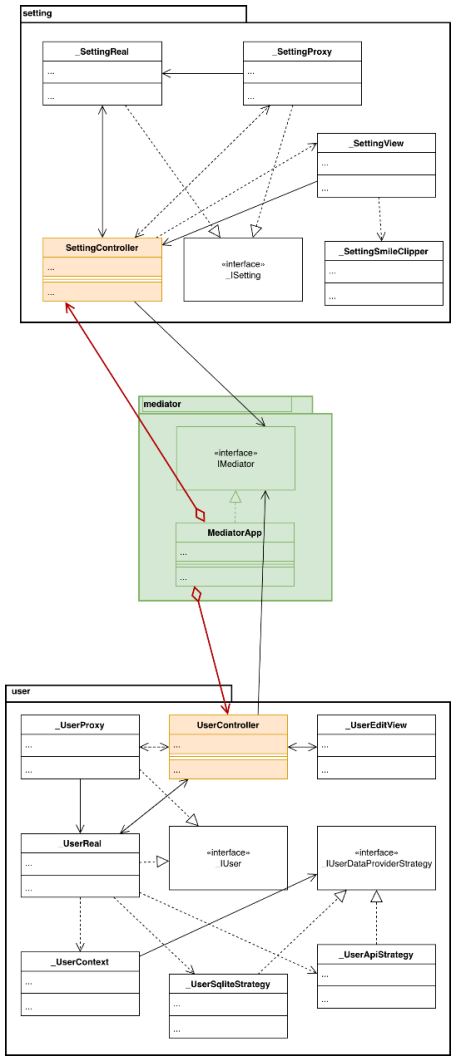


Fig. 3. UML diagram illustrating the mediator pattern implementation.

## C. Implementation of Mediator Pattern

Fig. 3 illustrates the Mediator structure generated by the framework. In the evaluated generated architecture, the inter-module interaction represented in the structural comparison is coordinated through the Mediator component rather than through the direct Login–Register references observed in the legacy architecture.

The generated Mediator structure routes the evaluated inter-module interactions through a common coordination mechanism rather than through direct cross-module references. This organization reduces explicit relationships between the participating functional modules and may limit change propagation caused by direct references. However, the actual maintenance effort required to modify or extend the generated modules was not measured in the present study.

## D. Implementation of Strategy Pattern

Fig. 4 illustrates the Strategy structure generated by the framework. The structure represents REST API and SQLite implementations through a common data-provider abstraction.

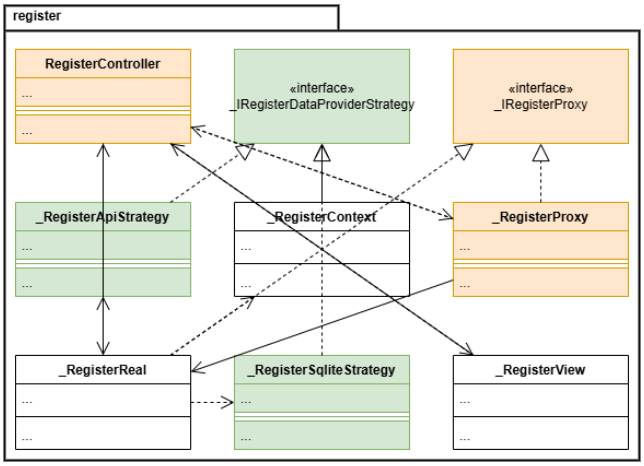


Fig. 4. UML diagram illustrating the strategy pattern implementation.

This organization separates provider-specific behavior from controller-facing logic and structurally supports provider substitution or extension. However, the experiment did not include a controlled change task measuring the files, source-code modifications, or developer effort required to introduce an additional provider. Therefore, the result is interpreted as structural support for extensibility rather than quantitative evidence of reduced maintenance effort.

## E. Discussion

The evaluation provides complementary evidence concerning operational feasibility and architectural structure. Operationally, the proposed framework successfully generated, executed, and built Flutter applications from all three tested JSON specifications, covering one-, two-, and eight-module configurations. These results indicate that the generation mechanism can operate across the evaluated levels of functional complexity. Nevertheless, the limited number of specifications means that the findings demonstrate feasibility within the tested configuration space rather than general reliability across arbitrary Flutter applications.

The comparison between the legacy and generated architectures provides additional evidence regarding inter-module dependency management. In the evaluated Login–Register interaction, the legacy architecture contains two direct cross-module relationships, whereas the generated architecture replaces these direct relationships with communication through the Mediator component. Accordingly, the direct functional-module dependency count decreases from two to zero for this evaluated interaction. This result supports the intended role of the Mediator generation rule in reducing direct relationships among functional modules. However, the mediator remains an architectural dependency and may itself become a coordination point as the number and complexity of module interactions increase.

The Strategy implementation provides complementary structural support for variation in data-access behavior. REST API and SQLite implementations are represented through a common provider abstraction, allowing provider-specific behavior to remain separated from controller-facing logic. This demonstrates that the generator can systematically encode alternative data-access mechanisms within the generated architecture. Nevertheless, no controlled change-impact experiment was performed to quantify the effort required to introduce an additional provider. Therefore, the current evidence supports architectural extensibility at the structural level but does not establish a quantitative reduction in maintenance effort.

The results should also be interpreted within the limitations of the selected evaluation evidence. Direct functional-module relationships and UML-based architectural inspection capture observable aspects of dependency organization and generated structure, but they do not represent all dimensions of software quality. Maintainability, scalability, and testability are multidimensional properties that may also depend on source-code complexity, test coverage, runtime behavior, developer comprehension, defect history, and long-term evolution. Accordingly, the present study limits its conclusions to the architectural relationships and operational behavior directly supported by the evaluated artifacts.

A direct empirical benchmark against external Flutter code generators was not conducted because the reviewed approaches differ in functional scope, input representation, architectural objectives, and availability of comparable implementations. For this reason, existing approaches are compared at the capability level in the Related Work section, while the empirical architectural comparison uses the available legacy application as an internal baseline. A controlled benchmark using functionally equivalent application specifications and reproducible external generators remains an important direction for future work.

Overall, the findings indicate that the proposed architecture-aware generation approach can transform the evaluated JSON specifications into executable Flutter applications while applying predefined rules for module organization, mediator-based communication, and strategy-based data access. The evidence supports architectural conformance and controlled direct coupling in the evaluated cases, but broader claims regarding long-term maintainability, large-scale scalability, and developer productivity require additional longitudinal and comparative experiments.

## V. Threats to Validity

The study does not report comprehensive quantitative source-code quality or architectural metrics, such as coupling measures, dependency-violation counts, cyclomatic complexity, generated lines of code, or class-level structural metrics. Accordingly, the reported structural evidence should be interpreted as targeted architectural inspection rather than as a comprehensive quantitative assessment of generated-code quality. Future evaluations should incorporate reproducibly defined source-code and architecture metrics alongside structural inspection. This study has several limitations that should be considered when interpreting the results.

### A. Internal Validity

The generated applications depend on predefined templates and architectural generation rules. Consequently, defects or biases in these templates may influence the resulting architectural structures. To reduce this threat, the same generator, JSON configuration mechanism, software environment, and evaluation procedure were applied to all three experimental scenarios. The generated outputs and UML structures were also inspected to verify that the intended Mediator and Strategy relationships were present.

### B. Construct Validity

The structural evaluation focuses on directly observable architectural relationships, including direct functional-module dependencies and the representation of Mediator- and Strategy-based structures. These observations capture selected aspects of architectural organization but do not directly measure broader software-quality attributes such as maintainability, scalability, testability, developer comprehension, or maintenance cost. Likewise, generation time, build time, and APK size are operational characteristics rather than direct measures of these quality attributes. Therefore, the findings are interpreted as evidence of operational feasibility and structural architectural conformance within the evaluated scenarios.

### C. Conclusion Validity

The evaluation involved three application specifications, and each scenario was executed once. Consequently, the reported generation and build results are descriptive observations rather than statistical estimates of framework reliability or performance. No claim of statistical superiority over other code generators is made. The three specifications were selected to represent increasing functional complexity rather than to constitute a statistically representative sample of Flutter applications.

### D. External Validity

The experiments were limited to Flutter applications generated from the supported JSON schema, predefined templates, and eight available functional modules. Applications requiring highly customized state management, platform-specific integrations, or architectural relationships outside the supported abstractions may require additional manual implementation. Therefore, the findings should be generalized

only to application structures that fall within the current framework configuration.

In practical terms, the current framework is most applicable to modular Flutter applications whose pages, routes, functional modules, and data-provider requirements can be represented by the supported JSON schema. Projects requiring highly customized architectural structures may require extensions to the schema or generation templates. Therefore, industrial adoption should also consider template customization and integration with manually maintained source code.

## VI. Conclusion

This study proposed an architecture-aware Flutter code generation framework that transforms JSON-based application specifications into modular source code while systematically applying Clean Architecture, Mediator, and Strategy rules. The primary contribution is not the isolated use of these established architectural concepts, but their encoding as repeatable specification-to-code generation rules that are instantiated in the evaluated generated application structures.

The framework successfully generated, executed, and built all three evaluated application specifications containing one, two, and eight functional modules. In the structural comparison, the two direct Login–Register relationships observed in the legacy architecture were replaced by mediator-based communication in the generated architecture, resulting in no direct Login–Register relationship for the evaluated interaction. The generated Strategy structure also represented REST API and SQLite data providers through a common abstraction, demonstrating that the intended data-provider variation was incorporated into the generated architecture.

These findings should be interpreted within the scope of the experiment. The evaluation was limited to three predefined specifications, template-generated Flutter applications, selected architectural relationships, and single executions of each scenario. Therefore, the results do not constitute general proof of long-term maintainability, unlimited scalability, or superiority over other Flutter code generators.

Future work should evaluate larger industrial applications, perform repeated performance experiments, compare the framework with functionally equivalent and reproducible generators, conduct controlled maintenance and change-impact studies, and assess developer effort and comprehension. Additional work may also extend the JSON specification and generation templates to support further architectural and behavioral design patterns.

More broadly, architecture-aware code generation provides a practical direction for transforming architectural decisions from developer-dependent guidelines into explicit and repeatable software-construction rules.

## Acknowledgment

The authors would like to thank SimHive Group (PT. Maulidan Teknologi Kreatif) for their cooperation and support during this research.

## Declarations

Conflict of Interest: The authors declare that they have no competing interests that could have influenced the work reported in this manuscript.

Funding: This research received no financial support from any funding agency, institution, or commercial organization.

Data Availability: The data and supporting materials used in this study are available from the authors upon reasonable request.

Declaration of Generative AI Use: Generative AI tools were used only for language refinement and editorial assistance during manuscript preparation. All scientific content, analysis, interpretations, and conclusions remain the responsibility of the authors.

## References

[1] S. A. Bhagat, “Review on Mobile Application Development Based on Flutter Platform,” Int. J. Res. Appl. Sci. Eng. Technol., vol. 10, no. 1, pp. 803–809, Jan. 2022, doi: 10.22214/ijraset.2022.39920.

[2] C. W. Butler and T. J. McCabe, “Cyclomatic Complexity-Based Encapsulation, Data Hiding, and Separation of Concerns,” Journal of Software Engineering and Applications, vol. 14, no. 01, pp. 44–66, 2021, doi: 10.4236/jsea.2021.141004.

[3] A. Khalilipour, M. Challenger, M. Onat, H. Gezgen, and G. Kardas, “Refactoring Legacy Software for Layer Separation,” International Journal of Software Engineering and Knowledge Engineering, vol. 31, no. 02, pp. 217–247, Feb. 2021, doi: 10.1142/S0218194021500066.

[4] T. Rachovski, S. Hadzhikoleva, E. Hadzhikolev, and A. Lengerov, “Using Clean Architecture Principles to Improve the Design and Implementation of the Mobiles Online Platform,” 2022, pp. 11–19. doi: 10.1007/978-981-16-7657-4_2.

[5] J. A. Santiago-Salazar and D. Rico-Bautista, “Clean Architecture: Impact on Performance and Maintainability of Native Android Projects,” 2024, pp. 82–90. doi: 10.1007/978-3-031-47372-2_8.

[6] M. B. Sinatria, Oman Komarudin, and Kamal Prihamdani, “Penerapan Clean Architecture Dalam Membangun Aplikasi Berbasis Mobile Dengan Framework Google Flutter,” Infotech Journal, vol. 9, no. 1, pp. 132–146, May 2023, doi: 10.31949/infotech.v9i1.5237.

[7] V. Sarcar, “Mediator Pattern,” in Java Design Patterns, Berkeley, CA: Apress, 2022, pp. 489–511. doi: 10.1007/978-1-4842-7971-7_23.

[8] P. Petrova and Z. Varbanov, “Implementation of the software design patterns mediator and CQRS with the library mediatr in .NET,” Mathematics, Computer Science and Education, vol. 7, no. 2, pp. 75–82, Dec. 2024, doi: 10.54664/URDG9808.

[9] V. Sarcar, “Strategy Pattern,” in Java Design Patterns, Berkeley, CA: Apress, 2022, pp. 455–468. doi: 10.1007/978-1-4842-7971-7_21.

[10] R. A. Wijayanto and RR. H. P. Sejati, “Implementing Flutter Clean Architecture for Mobile Tourism Application Development,” Int. J. Comput. Appl., vol. 185, no. 39, pp. 23–30, Nov. 2023, doi: 10.5120/ijca2023923197.

[11] A. Aljaloud and A. Razzaq, “An Innovative Metric-based Clustering Approach for Increased Scalability and Dependency Elimination in Monolithic Legacy Systems,” Engineering, Technology & Applied Science Research, vol. 13, no. 4, pp. 11375–11386, Aug. 2023, doi: 10.48084/etasr.6048.

[12] S. R. Idate, T. S. Rao, and D. J. Mali, “Context-Based Aspect-Oriented Requirement Engineering Model,” Engineering, Technology & Applied Science Research, vol. 13, no. 2, pp. 10460–10465, Apr. 2023, doi: 10.48084/etasr.5699.

[13] S. Boukhary and E. Colmenares, "A Clean Approach to Flutter Development through the Flutter Clean Architecture Package," in 2019 International Conference on Computational Science and Computational Intelligence (CSCI), IEEE, Dec. 2019, pp. 1115–1120. doi: 10.1109/CSCI49370.2019.00211.

[14] B. W. Stirgwolt, T. A. Mazzuchi, and S. Sarkani, "A model-based systems engineering approach for developing modular system architectures," Journal of Engineering Design, vol. 33, no. 2, pp. 95–119, Feb. 2022, doi: 10.1080/09544828.2021.1980203.

[15] R. S. Sangwan, R. L. Nord, and I. Ozkaya, "Architectural Dependency Analysis: Addressing the Elephant in the Room," Computer (Long. Beach. Calif)., vol. 54, no. 3, pp. 73–78, Mar. 2021, doi: 10.1109/MC.2021.3050538.

[16] P. Sun, D.-K. Kim, H. Ming, and L. Lu, "Measuring Impact of Dependency Injection on Software Maintainability," Computers, vol. 11, no. 9, p. 141, Sep. 2022, doi: 10.3390/computers11090141.

[17] K. Lano, H. Haughton, Z. Yuan, and H. Alfraihi, "Agile model-driven re-engineering," Innov. Syst. Softw. Eng., vol. 20, no. 4, pp. 559–584, Dec. 2024, doi: 10.1007/s11334-024-00568-z.

[18] L. Jiménez-Navajas, R. Pérez-Castillo, and M. Piattini, "Code generation for classical-quantum software systems modeled in UML," Softw. Syst. Model., vol. 24, no. 3, pp. 795–821, Jun. 2025, doi: 10.1007/s10270-024-01259-w.

[19] I. B. Tiali, Z. Aarab, A. Lyazidi, and M. D. Rahmani, "Model-Driven Transformation of Business Processes into Blockchain Smart Contracts," International Journal of Advanced Computer Science and Applications, vol. 16, no. 12, 2025, doi: 10.14569/IJACSA.2025.01612120.

[20] C. P. A. Yataco, M. A. L. Salinas, and J. A. H. Quispe, "Mobile Application Based on Clean Architecture for Sustainable Tourism Recommendation Using BERT," International Journal of Advanced Computer Science and Applications, vol. 16, no. 11, 2025, doi: 10.14569/IJACSA.2025.0161199.

[21] S. Abufakher, "Empirical Quantitive Investigation of the Effect of GoF Design Patterns on the Quality of Software Systems," International Journal of Advanced Computer Science and Applications, vol. 17, no. 3, 2026, doi: 10.14569/IJACSA.2026.0170371.