Comparative Study of Manual and Generated Data Transfer Object Implementation Performance

Chandro Pardede[1,2]*, Wilson Sihombing[1], Winfrey Nainggolan[1]
[1] Faculty of Informatics and Electrical Engineering, Del Institute of Technology, Sitoluama, 22381, Indonesia
[2] Graduate School of Natural Science and Technology, Kanazawa University, Kanazawa, Ishikawa, 920-1192, Japan
chandro.pardede@del.ac.id [1], iss22011@students.del.ac.id[2], iss22001@students.del.ac.id[3]

Article Info

Article history:

Received 2025-08-17
Revised 2025-09-15
Accepted 2025-10-18

Keyword:

Data Transfer Object,
Flutter,
json_serializable,
Parsing,
Performance.

ABSTRACT

The Data Transfer Object (DTO) plays a crucial role in Flutter application development, particularly in the process of data serialization and deserialization. This study compares two DTO implementation approaches namely manual and generated with a focus on execution speed and memory usage efficiency. Testing was conducted at three data complexity levels (Small, Medium, Large) over 100 iterations using Flutter DevTools. The results show that the generated approach (using libraries like json_serializable) outperforms the manual approach in parsing speed, with a ratio of 1:1.147, and memory efficiency, with a ratio of 1:1.42. While the manual approach offers more flexibility in handling conditional parsing logic, it is more error-prone and less efficient when processing large datasets. In contrast, the generated approach proves faster, more scalable, and reduces the potential for human errors, making it the optimal choice for projects requiring technical efficiency and rapid development. This study recommends using generated DTOs for applications with large data sets and high complexity, while manual DTOs are better suited for dynamic parsing needs.

I. Introduction

The development of modern mobile applications increasingly emphasizes efficiency and performance, especially on platforms like Flutter, which offers a cross-platform approach with performance close to native [1]. One common practice in Flutter development is the use of Data Transfer Objects (DTOs) to separate the data representation from the application's logic entities. DTOs act as a bridge between the data received from the server and the components within the application [2]. In practice, DTOs can be created manually or through code generation using libraries or annotations [3]. Each of these approaches has its own strengths in terms of data structuring, code clarity, and full control over the JSON format [4]. The novelty of this research lies in its specific comparative focus on the implementation of manual and generated Data Transfer Objects (DTOs) in Flutter application development, emphasizing runtime performance and memory usage efficiency. This differentiates it from previous studies that generally discuss Flutter optimization in a broader sense. The aim of this research is to evaluate the performance comparison between the two approaches, with the main focus on two aspects namely execution speed and memory usage efficiency. This study is motivated by the need to evaluate and compare the performance of these two approaches, particularly in terms of processing speed and memory efficiency during data mapping in Flutter application development. Furthermore, the implementation of DTOs aligns with the principles of clean architecture, which emphasizes the separation of business logic from technical details, thereby maintaining the modularity and scalability of the application. However, despite the critical role of DTOs, there is limited research that systematically compares the manual and generated approaches in the context of Flutter performance. Therefore, this research is expected to fill this gap.

While the manual approach allows for flexibility and full control over object structures, it also comes with several drawbacks. One of the main issues is the large amount of boilerplate code and the potential for human error during the coding process [5]. Manual coding is more prone to human error compared to the generated DTO approach. This is because, in the manual approach, developers have to explicitly write each fromJson() and toJson() function. This process heavily relies on the developer's accuracy in handling various data types and changing object structures. Any mistakes in writing the code, such as mismatched data types or errors in attribute mapping, can lead to bugs or logical errors in the application. On the other hand, generated DTOs using libraries such as json_serializable, freezed, or build_runner can significantly reduce these burdens [6]. However, this generated approach often introduces added complexity during build time, dependency on third-party libraries, and reduced clarity for novice developers unfamiliar with annotations and code generation.

As Flutter adoption grows across various industries, there is a significant opportunity to explore and define best practices in managing DTOs. This research aims to address the need for a deeper understanding of how each DTO approach performs in terms of processing speed and memory usage during data mapping in Flutter applications. Additionally, the study seeks to provide guidelines for development teams in choosing the most suitable DTO strategy based on the specific needs and context of their projects.

Nevertheless, several challenges remain, such as the rapid evolution of the ecosystem and its supporting libraries, as well as the varying needs of different projects that may affect the effectiveness of one approach over the other [7]. Furthermore, there are still very few systematic comparative studies that evaluate the performance and development efficiency between manual and generated DTO approaches. This poses a risk to developers and project managers in making informed technical decisions.

Based on the aforementioned background, this study is essential to provide a comparative analysis of the impact of using manual versus generated DTOs on runtime performance and development efficiency in Flutter applications. It is expected that this research will contribute to the Flutter developer community in making optimal technical decisions and promoting more efficient and standardized application development practices.

II. METHOD

A. Research Approach

This study employs a quantitative experimental method aimed at comparing the performance of two approaches to implementing Data Transfer Objects (DTOs) in Flutter applications: the manual approach and the generative approach. The evaluation is conducted by measuring two key metrics: execution time and memory usage during the processes of JSON deserialization and serialization. Through this approach, the researcher seeks to observe and objectively analyze the measurement results based on variations in data size and the DTO approach applied.

B. Dataset

TABLE I
DATA OBJECT CATEGORIES

Category	Total attributes
Small	52
Medium	104
Large	156

The The dataset used in this study is a mock JSON API, consisting of synthetic data artificially generated to represent the patterns and distributions of real data without using the actual data directly [8]. This data is presented in the JSON object format with various attributes.

Each dataset category is determined based on the complexity level of the number of attributes within a single JSON object. The Small category consists of 52 attributes, the Medium category has 104 attributes (double the Small category), and the Large category includes 156 attributes (double the Medium category). Thus, the dataset complexity increases proportionally with a ratio of 1:2:3 for each category.

Each JSON object contains various common data types in programming, such as String, int, double, bool, DateTime, List<T>, and Map<String, T>. This proportional increase in the number of attributes allows for the evaluation of DTO (Data Transfer Object) mapping performance, both manually and generatively, enabling an analysis of data processing efficiency from small to large scales.

C. DTO Implementation

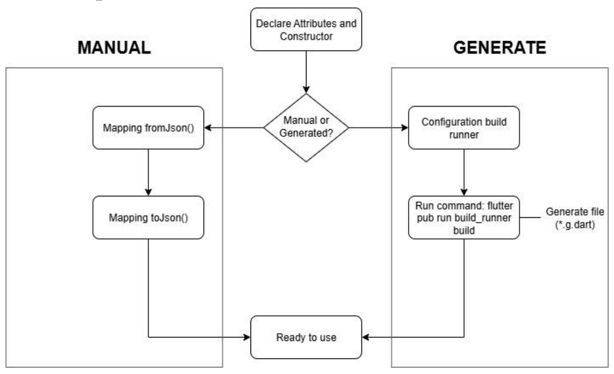
Declare Attributes and Constructor
MANUAL
GENERATE
Manual or Generated?
Mapping fromJson()
Mapping toJson()
Configuration build runner
Run command: flutter pub run build_runner build
Generate file (*.g.dart)
Ready to use

Figure 1. Workflow of generated and manual implementation

The implementation of Data Transfer Objects (DTOs) in this study adopts two main approaches: the manual approach and the generated (automatic) approach. These two methods are used to compare performance in terms of execution time and code-writing efficiency during the serialization and deserialization of objects. Figure 1 illustrates the implementation workflow of each approach. The workflow begins with the declaration of attributes and constructors, which serve as the foundation for DTO formation in both manual and automated approaches.

In the manual approach, object mapping is performed by explicitly writing the fromJson() and toJson() functions. Meanwhile, in the generated approach, the mapping process is assisted by code generation libraries such as json_serializable and build_runner. Once the initial configuration is completed, the command flutter pub run build_runner build is executed to generate a file with the extension *.g.dart, which contains the automatically generated code for the serialization and deserialization processes.

D. Testing Techniques

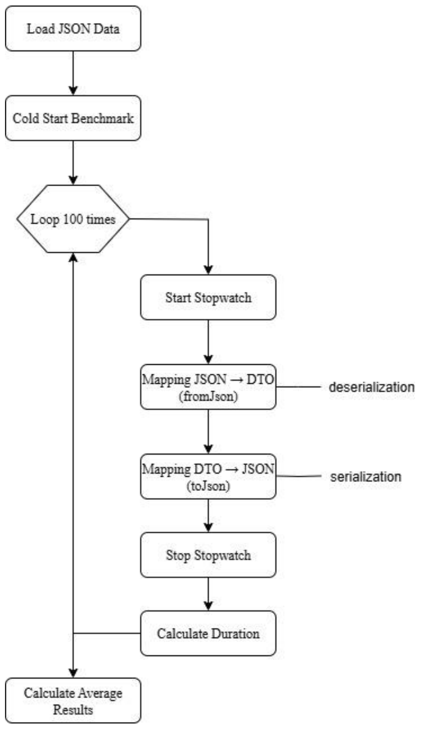

Figure 2. Testing workflow

The testing in this study is conducted to measure the performance of data mapping processes using both manual and generative approaches to Data Transfer Objects (DTOs). The evaluation focuses on two main aspects: execution time and memory usage during the deserialization (fromJson) and serialization (toJson) processes. One of the methods used to measure performance in this study is benchmarking with a stopwatch.

Benchmarking with a stopwatch is a performance evaluation technique that utilizes a high-precision time-measuring tool to record the execution duration of a code snippet at a micro level [9]. In this case, the Stopwatch class from the dart: core library is used. This class is designed to measure time precisely in microseconds (µs) to milliseconds (ms). The stopwatch is initialized before the mapping process begins and stopped immediately after the process ends, allowing the total execution duration to be accurately captured and compared between the two approaches.

1) Execution Speed Testing: The testing procedures were carried out as follows:

- The test dataset in JSON format is first loaded according to the predefined size categories: Small, Medium, and Large.
- An initial execution (cold start) is performed to map the JSON data to the DTO object and vice versa, aiming to measure the initialization time before the testing is repeated.
- The test is conducted over 100 iterations to obtain average values. In each iteration, a stopwatch is activated to begin measuring the duration of the process. The JSON data is then mapped to the DTO object using the fromJson function, and subsequently, the DTO object is converted back to JSON format using the toJson function. Once the conversion is complete, the stopwatch is stopped to record the time taken for that iteration.
- After all iterations are completed, the average execution time is calculated for each approach (manual and generated) across all dataset categories.
- To ensure measurement stability, a 100-millisecond delay is added at the end of each iteration to allow the garbage collector to clean up temporary memory allocations.

2) Memory Usage Testing: Memory measurement is conducted using the Memory feature available in Flutter DevTools. This tool is utilized to monitor and analyze memory usage in detail during the serialization and deserialization processes. The analysis is based on resource usage snapshots taken after the entire testing sequence is completed, providing a comprehensive view of memory allocation efficiency for each approach [10]. These results offer insights into the memory consumption of each mapping approach (manual and generated), as well as their overall impact on system efficiency.

E. Evaluation Metrics

1) Execution Time: This metric measures how fast the serialization (converting objects to JSON) and deserialization (converting JSON to objects) processes are executed for each approach (manual vs. generated DTO). The measurement is performed in microseconds (µs) and calculated based on the average value from a number of test iterations. Execution time reflects the duration required by the system to run a program, including the processes of fetching instructions and data from memory, as well as the sequential execution of commands within the processor [11].

2) Memory Usage: This metric aims to observe memory consumption during the data mapping process. Measurements are conducted using the Memory feature available in Flutter DevTools, which provides real-time statistics on heap memory usage as well as memory snapshots. This method allows researchers to monitor memory allocation and deallocation during serialization and deserialization processes. More than just statistical values, memory usage serves as a crucial indicator that reflects the health and performance of an application—especially in complex scenarios where applications handle large volumes of data intensively [12].

F. Testing Environment

To ensure that the testing results are fair, consistent, and reproducible, all experiments were conducted in a controlled hardware and software environment. This environment was carefully designed to resemble real-world conditions in which Flutter applications are typically used by end users. Such a setup is crucial to accurately reflect the actual performance of DTO mapping approaches, whether done manually or through code generation.

The testing was carried out using a laptop with standard specifications, as detailed in the following table:

TABLE II
DEVICE SPECIFICATIONS

Component	Specification
Processor	Intel(R) Core(TM) i5-10500H CPU @ 2.50GHz (12 CPUs), ~2.5GHz
RAM	16384MB
Operating System	Windows 11 Home Single Language 64-bit (10.0, Build 22631)
Flutter SDK	3.32.2
Programming Language	Dart 3.8.1
Code Editor	Android Studio Ladybug | 2024.2.1 Patch 2
Profiling Tool	DevTools via Android Studio

The experiment was conducted using a laptop with an Intel(R) Core(TM) i5-10500H CPU @ 2.50GHz (12 CPUs), 16 GB RAM, and Windows 11 Home Single Language 64-bit (Build 22631) as the operating system. The development environment utilized Flutter SDK version 3.32.2, Dart programming language 3.8.1, and Android Studio Ladybug (2024.2.1 Patch 2) as the main editor. To support performance profiling, Flutter DevTools was used, while the generated DTO implementation was built with the json_serializable and build_runner libraries. Execution speed was measured using the Stopwatch class from the dart:core library, while memory usage was analyzed through the Memory feature in DevTools.

III. RESULT AND DISCUSSION

A. Testing Results

The testing was conducted to evaluate and compare the data parsing performance between two approaches to implementing Data Transfer Objects (DTOs): the manual implementation and the generated implementation using the json_serializable library within the Flutter/Dart environment. All tests were performed in accordance with the environment and configurations described in Chapter 2, utilizing the specified hardware and data sets.

1) DTO Parsing Speed: The first test was conducted by measuring the parsing time of DTO objects—from JSON format to Dart objects, and vice versa, from Dart objects to JSON format. The testing process was repeated for 100 iterations for each method, and the parsing time results were recorded in a .csv file format. This data was then used to generate performance visualization graphs for the three data size categories, which can be seen below.

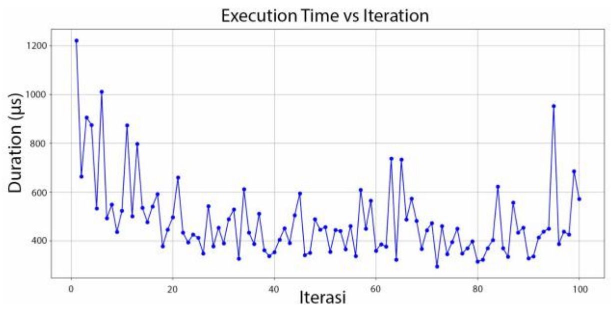

Figure 3. Manual parsing for small data

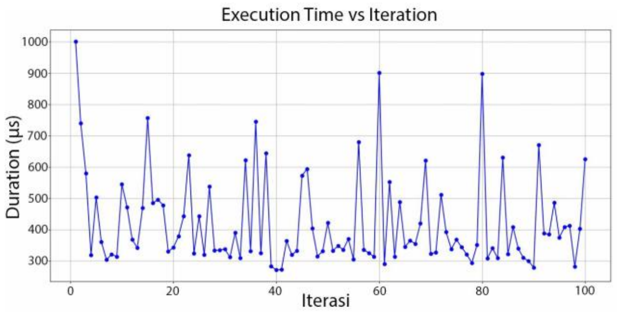

Figure 4. Generated parsing for small data

From the two graphs in the small data category, Figure 3 and Figure 4, which compare execution time in microseconds over 100 iterations, it is observed that Figure 3 (manual approach) demonstrates relatively stable performance after a significant spike in execution time during the first iteration, with most durations falling within the 400 to 800 µs range. In contrast, Figure 4 (generated approach) shows a slightly lower average execution time but with greater variation and several sharp spikes reaching up to 900 µs, indicating higher instability compared to the manual approach.

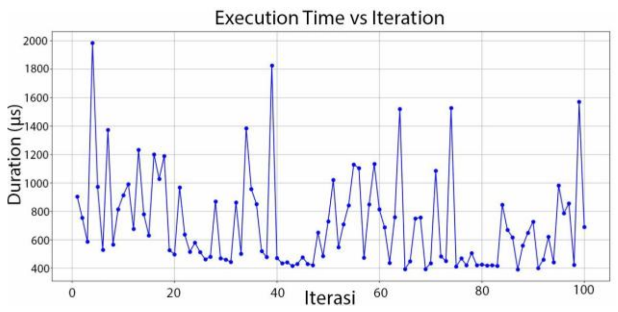

Figure 5. Manual parsing for medium data

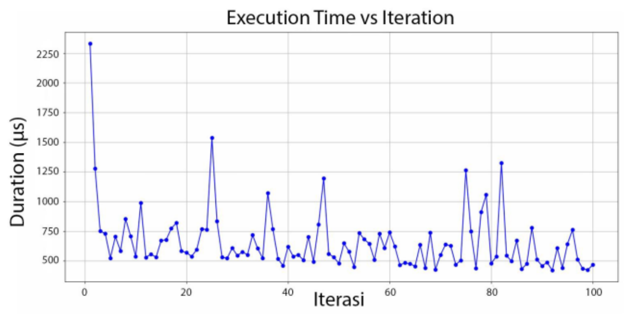

Figure 6. Generated parsing for medium data

From the two graphs in the medium data category, Figure 5 and Figure 6, which compare execution time in microseconds over 100 iterations, a significant difference in performance patterns can be observed. Figure 5 (manual approach) shows a higher average execution time with extremely wide variation, including several sharp spikes reaching nearly 2000 µs, indicating instability and inefficiency in the parsing process. In contrast, Figure 6 (generated approach) demonstrates much more stable performance after an initial significant spike, with a lower average execution time and most durations remaining below 1000 µs.

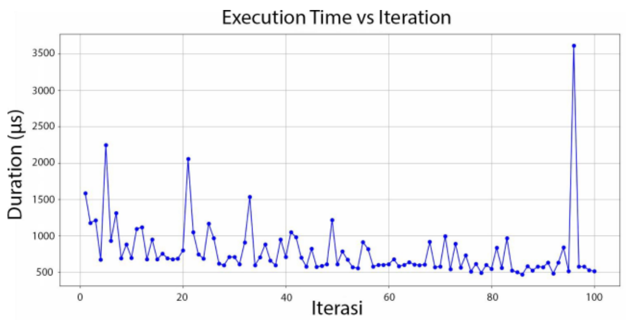

Figure 7. Manual parsing for large data

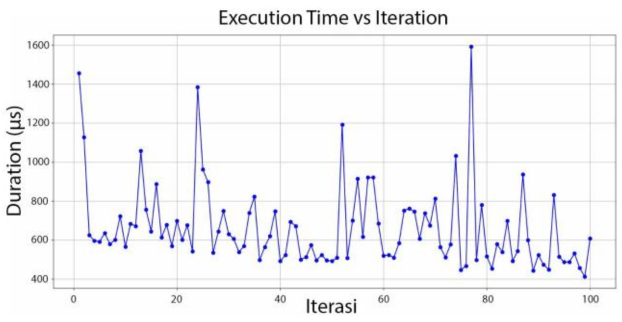

Figure 8. Generated parsing for large data

From the two graphs in the large data category—Figure 7 and Figure 8—which compare execution time in microseconds over 100 iterations, a striking performance difference is evident. Figure 7 (manual approach) shows a significantly higher and volatile average execution time, marked by extreme spikes exceeding 3500 µs. In contrast, Figure 8 (generated approach) demonstrates much better and more stable performance, with a lower average execution time and less extreme duration variations, where the highest peak only reaches around 1600 µs. In this large data set, the average speed of the manual approach was compared with the generated approach, resulting in a speed ratio of 1:1.147. The interpretation of this parsing speed ratio of 1:1.147 indicates that the generated DTO approach can execute the serialization and deserialization processes approximately 14.7% faster than the manual DTO. While this difference may appear relatively small in numerical terms, its significance largely depends on the context of the application being developed. In applications with large data sets or high workloads, the 14.7% difference can have a significant impact on response time and operational efficiency.

Based on the comparison of all six graphs, the parsing performance between the manual and generated methods shows a notable difference, particularly at the medium and large data scales. For small data, both methods demonstrate relatively comparable performance, although the manual method tends to be more stable. However, as data complexity and size increase, the generated method exhibits a clear advantage, with lower and more stable parsing times, while the manual method experiences drastic execution time spikes and high variability. This indicates that the generated method is more efficient and reliable when handling large-scale data parsing. While the performance difference is not significant at a smaller scale, for medium to large-scale data processing needs, the generated approach proves to be a more optimal choice in terms of performance.

2) Average Parsing Speed: To provide a clearer overview of the performance efficiency between the two methods, the average parsing time over 100 iterations for each data size category was calculated and visualized in the form of a bar chart, as shown in the Figure 9.

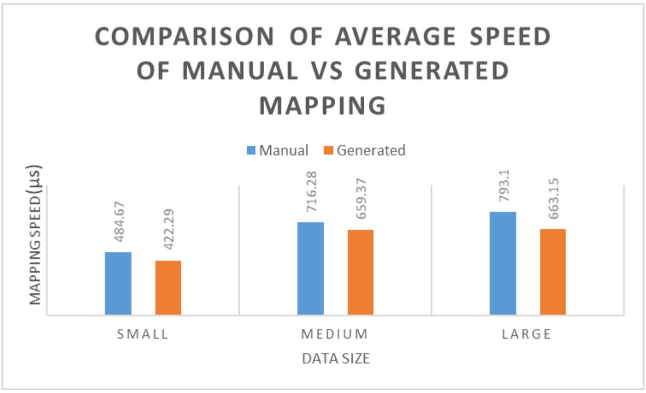

Figure 9. Comparison of average parsing speed

Based on the diagram, it can be seen that the average parsing time for the generated DTO is consistently lower than that of the manual DTO. For large-sized data, the manual approach results in a parsing time of 793.1 ms, while the generated approach results in 663.15 ms. For medium-sized data, the manual approach results in 716.28 ms, while the generated approach results in 659.37 ms. Meanwhile, for small-sized data, the manual approach results in 484.67 ms, and the generated approach results in 422.29 ms. The difference in average parsing times becomes more significant with larger data, indicating that the generated DTO has better scalability.

3) *Memory Usage:* In addition to speed measurements, memory usage was also tested after the parsing process was completed. Memory usage was measured in kilobytes (KB) using the DevTools feature available in the text editor used during development. The results of this test are presented in Figure 10.

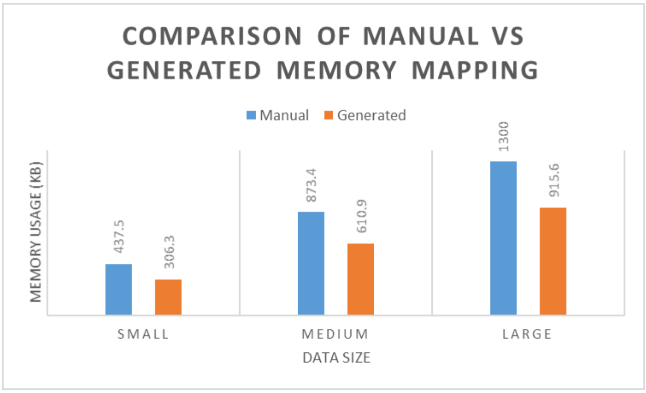

Figure 10. Comparison of memory usage in data parsing

The test results show that memory usage in the manual implementation tends to be higher and increases drastically with larger data sizes. For large-sized data, the manual approach results in a parsing time of 1300 ms, while the generated approach results in 915.6 ms. For medium-sized data, the manual approach results in 873.4 ms, while the generated approach results in 610.9 ms. Meanwhile, for small-sized data, the manual approach results in 484.67 ms, while the generated approach results in 422.29 ms. On the other hand, the generated DTO implementation shows better memory efficiency with relatively lower increases.

Based on the analysis, it can be concluded that the memory usage ratio between manual and generated parsing tends to be consistent, around 1:1.42 (manual = 437.5 KB; generated = 306.3 KB), regardless of data size variations or the number of iterations performed. Meanwhile, the parsing speed ratio is around 1:1.147 (manual = 484.67 ms; generated = 422.29 ms).

B. *Discussion*

The test results indicate that both manual and generated approaches in DTO implementation exhibit stable parsing performance across 100 test iterations. This is supported by findings stating that schema-based serializers consistently offer better parsing performance and memory efficiency compared to schema-less approaches such as manual JSON parsing [13]. From the tests conducted, the generated approach proved to be faster because the closure function was called and reused more frequently than in the manual approach. In the program code, the closure is created once at the beginning and then reused whenever needed, resulting in more efficient execution time. Although the generated DTO approach has proven superior in terms of parsing time efficiency and memory stability, there are several drawbacks that need to be considered. First, the use of generated DTO adds complexity to the build process because it heavily relies on third-party libraries such as json_serializable and build_runner. This dependency may cause compatibility issues when Flutter or Dart undergo version updates, requiring ongoing maintenance. Second, the code generation process can increase build time, especially in large projects with numerous data models, which can slow down the development cycle. Third, this approach tends to limit the flexibility of parsing logic, as annotations and code generators only support standard attribute mapping patterns. If developers need conditional data transformations, such as converting specific values, the generated approach becomes less optimal and often still requires additional manual code. Additionally, for beginner developers, the use of annotations and automatically generated files can create a steeper learning curve, as the code structure is not entirely explicit and can be difficult to understand without familiarity with the build_runner mechanism. The performance fluctuations observed during testing can be attributed to system conditions that are not entirely controllable. In the context of parallel and multithreaded programs, these irregularities may be caused by various factors, such as load imbalance between threads, task scheduling mechanisms by the operating system, synchronization overhead, and contention for shared resources [14]. These factors can lead to unpredictable execution time variations and contribute to performance spikes during benchmarking processes. Although both approaches demonstrate consistency, the generated DTO approach yields a faster average parsing time compared to the manual approach, especially as the data size increases. This indicates that the generated approach has more scalable characteristics and is better suited for applications with high data loads.

The test data on average parsing time reinforces this finding, where the generated DTO approach demonstrates higher parsing efficiency across small, medium, and large data scales. This efficiency becomes even more significant with large-sized object data, indicating that the generated approach not only reduces development workload by eliminating boilerplate code but also proves to be more robust in handling increasing data complexity. Therefore, scalability becomes a key added value, especially for applications that handle large volumes of data or operate in real-time contexts.

In addition to speed, memory usage testing revealed that the manual DTO approach tends to consume more memory, especially at larger data scales. The memory usage ratio between the manual and generated approaches reached 1:1.42, highlighting the generated approach's advantage in efficient memory management. This finding is particularly important in the context of resource-constrained systems, such as mobile applications that are sensitive to memory consumption.

These findings indicate that using the json_serializable library offers benefits not only in terms of maintainability and development efficiency, but also in terms of technical performance [15]. The generated DTO approach has proven to deliver competitive if not superior parsing results, particularly in memory efficiency and processing speed when dealing with large-scale data. Therefore, this approach is recommended for application development within the Flutter/Dart ecosystem.

Nevertheless, each approach has its own strengths and limitations. The generated approach excels in consistency, parsing time efficiency, and memory usage. However, it has limitations when special parsing handling is required, particularly when such needs cannot be addressed using standard annotations. For instance, when data transformation is needed based on certain conditions, such as adjusting attribute mapping depending on the value or data type received, these cases are typically not fully supported by standard annotations. In such scenarios, a manual approach is necessary to provide more flexible and tailored handling. On the other hand, the manual approach offers full flexibility and control over the parsing process but requires more maintenance effort and tends to be inefficient when data structures change frequently.

Based on the overall results and analysis, the generated DTO approach is recommended for most general use cases, especially when parsing efficiency and memory management are top priorities. However, the manual approach remains relevant for scenarios involving complex and non-standard parsing requirements, where custom parsing logic is necessary.

IV. CONCLUSION

The generated DTO approach is highly recommended for application development projects that involve numerous features and modules, particularly when managed by a multi-person development team. This method has been shown to effectively reduce code duplication, minimize human errors in attribute mapping, and simplify maintenance as system complexity grows. It is especially suitable for projects that demand high consistency in data transfer structures.

Opting for the generated DTO approach is advantageous in scenarios requiring technical efficiency and scalability, making it ideal for applications with high data throughput. Experimental results indicate that this approach outperforms manual DTOs by approximately 14.7% in parsing speed and offers better memory efficiency (≈1:1.42 ratio compared to manual), providing a significant benefit in repetitive and real-time workloads.

Moreover, the generated approach reduces boilerplate code, lowers the likelihood of human errors, shortens code review cycles, and eases refactoring when data schemas evolve. However, for projects requiring extensive flexibility in conditional parsing logic or highly dynamic data structures, the manual approach may still be preferable—though it demands greater maintenance and carries a higher risk of coding errors.

From a technical policy perspective, development companies are advised to establish code generation standards as part of their internal development guidelines. This standardization not only supports the efficiency of the development process but also facilitates module integration across applications through uniform and automated DTO formats.

Future research directions could focus on expanding testing with more complex real-world datasets, comparing various code generation libraries such as freezed, built_value, or Protobuf, and testing performance across different platforms (Android, iOS, web, desktop) and device classes. Additionally, future studies could introduce new metrics such as energy consumption, build time, and maintainability, accompanied by statistical analysis to test the significance of the results. A hybrid approach combining generated DTOs as the default with custom converters for specific cases also presents an interesting area for exploration to provide more comprehensive practical guidelines for the Flutter developer community.

BIBLIOGRAPHY

[1] F. Mushtaq, F. Azam, and M. W. Anwar, "Performance Comparison of Single Code Base Development Tools: Flutter, React Native, and Xamarin," *Proc. - 2024 14th Int. Conf. Softw. Technol. Eng. ICSTE 2024*, no. May 2025, pp. 17–23, 2024, doi: 10.1109/ICSTE63875.2024.00011.

[2] Fowler, M., Rice, D., Foemmel, M., Hieeatt, E., Mee, R., & Stafford, R. (2002). *Patterns of Enterprise Application Architecture*.

[3] T. Greifenberg et al., "Integration of handwritten and generated object-oriented code," Commun. Comput. Inf. Sci., vol. 580, pp. 112–132, 2015, doi: 10.1007/978-3-319-27869-8_7.

[4] T. Stahl and M. Völter, Model-Driven Software. 2006.

[5] T. P. Programmer, What others in the trenches say about.

[6] E. Umuhoza, "Domain-specific modeling and code generation for cross-platform multi-device mobile apps?," CEUR Workshop Proc., vol. 1499, pp. 50–60, 2015.

[7] Á. Fernández-Llamazares, I. Díaz-Reviriego, A. C. Luz, M. Cabeza, A. Pyhälä, and V. Reyes-García, "Rapid ecosystem change challenges the adaptive capacity of local environmental knowledge," Glob. Environ. Chang., vol. 31, pp. 272–284, 2015, doi: 10.1016/j.gloenvcha.2015.02.001.

[8] J. Jordon et al., "Synthetic Data -- what, why and how?," no. May, 2022, doi: 10.48550/arXiv.2205.03257.

[9] H. Swoboda, "Microbenchmark," Juv. Angiofibroma, no. 1976, pp. 3–9, 2017, doi: 10.1007/978-3-319-45343-9_1.

[10] M. Azis, A. Pinandito, and I. Maghfiroh, "Analisis Perbandingan Penggunaan State Management pada Aplikasi Ditonton menggunakan Framework Flutter," J. Pengemb. Teknol. Inf. dan Ilmu Komput., vol. 7, no. 1, pp. 148–153, 2023.

[11] Dewangga Andira Sulaeman, Ismah Nurul Sya'bani, M. Ashof Azria Azka, and Didik Aribowo, "Ruang Lingkup Organisasi Dan Arsitektur Komputer," J. Elektron. dan Tek. Inform. Ter. JENTIK), vol. 1, no. 4, pp. 164–177, 2023, doi: 10.59061/jentik.v1i4.519.

[12] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A Survey of Performance Optimization for Mobile Applications," IEEE Trans. Softw. Eng., vol. 48, no. 8, pp. 2879–2904, 2022, doi: 10.1109/TSE.2021.3071193.

[13] J. C. Viotti and M. Kinderkhedia, "A Benchmark of JSON-compatible Binary Serialization Specifications," 2022, [Online]. Available: http://arxiv.org/abs/2201.03051

[14] D. Kagaris, S. Dutta, and S. Eyerman, "Execution Time Estimation of Multithreaded Programs with Critical Sections," IEEE Trans. Parallel Distrib. Syst., vol. 33, no. 10, pp. 2470–2481, 2022, doi: 10.1109/TPDS.2022.3143455.

[15] J. C. Viotti and M. Kinderkhedia, "A Survey of JSON-compatible Binary Serialization Specifications," 2022, [Online]. Available: https://doi.org/10.48550/arXiv.2201.02089