DOI 10.34185/1562-9945-3-164-2026-06
UDC 004.415

A.O. Velma, M.P. Makarevych, N.V. Golian

COMPARATIVE ANALYSIS OF STATE MANAGEMENT MODELS IN MOBILE APPLICATIONS DEVELOPED USING FLUTTER

Annotation. The objective of this study is to formalize state management models in Flutter applications and evaluate them against criteria of transition determinism, performance, scalability, and architectural complexity. State management approaches are interpreted as extended transition systems with asynchronous events; safety and liveness properties are formalized using temporal logic. A formalized Flutter state management model is proposed as a class of reactive computational systems with defined safety and liveness properties. In addition, the study incorporates benchmark-based analysis of frame rendering time and CPU utilization to provide an empirical comparison of the considered approaches. The obtained results are further interpreted in the context of real-world development scenarios, enabling the formulation of practical recommendations for selecting appropriate state management architectures depending on application scale and complexity. The study revealed significant differences among approaches in determinism, performance, and architectural complexity, demonstrating that Riverpod provides the most balanced performance characteristics due to efficient dependency tracking, while Bloc and Cubit ensure higher structural clarity at the cost of increased computational overhead. MobX shows moderate performance with advantages in managing complex interdependent state, whereas Provider remains suitable for small-scale applications due to its simplicity. These findings enable informed architecture selection based on production application requirements, including system scale, performance constraints, and maintainability needs.
Keywords: state management, Flutter, reactive systems, formalization, Cubit, Riverpod, Provider, MobX, performance, scalability.

Problem statement. With the growing complexity of mobile applications, the volume of dynamic data and the number of interactions between interface components are also increasing. Modern mobile applications require efficient and scalable mechanisms for managing dynamic data and user interface interactions.

Flutter, as a cross-platform framework, provides several state management solutions that differ in architecture, performance, and developer experience (Flutter Official Docs, 2025). The choice of the most appropriate state management approach significantly affects the stability, performance, and maintainability of the application. In the Flutter environment, incorrect or irrational application of state management methods can lead to performance degradation, architectural complexity, increased error rates, and difficulties in project scaling (SolGuruz, 2025).

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Furthermore, the diversity of state management approaches creates a need for their systematic analysis and comparison in terms of efficiency, stability, and usability. In this regard, the study of methods and algorithms for mobile application state management is a relevant task of modern software engineering.

Analysis of recent research and publications. The issue of comparing state management approaches in Flutter has been explored in a number of scientific and practical publications. In particular, a paper published in the JOTI (Jurnal Online Teknik Informatika) journal of Universitas Dinamika conducted a comparative performance analysis of Provider and Riverpod libraries for datasets of 1,000 to 10,000 elements, with results showing a stable advantage of Riverpod in CPU usage and memory efficiency (JOTI, 2023).

This research is important because it is based on quantitative measurements in a controlled environment, rather than merely qualitative analysis of architectural characteristics. Another significant contribution to this field was made by the iCoderz Solutions platform, which published a comparative benchmark of five state management libraries, including Provider, Riverpod, Bloc, GetX, and MobX. The researchers measured frame rendering time and CPU load under identical workloads – list element operations – and established that Riverpod demonstrates the lowest frame rendering time (12 ms) and the lowest CPU load among the reviewed solutions (iCoderz, 2025).

Shakil examines in detail the architectural features of Bloc and Cubit, emphasizing that Bloc's event-driven model, although introducing greater overhead, ensures strict separation of concerns, which is critical in large team projects (Shakil, 2025). Kumar in his comparative review analyzes practical implementation cases of Provider, Bloc, and Riverpod in production projects of various scales – from e-commerce startups to enterprise fintech applications – and identifies criteria for choosing between them depending on project specifics (Kumar, 2025).

The analysis of MobX's reactive approach in the context of Flutter was carried out in the Foresight Mobile review, where special attention is paid to the transparency of the dependency tracking mechanism and its comparison with declarative models of Provider and Riverpod (Foresight Mobile, 2025). MobX's transparent reactive programming paradigm, originally developed for JavaScript, provides a scientific basis in the work of Meijer (2010), which formally grounds reactive programming as a computational model.

At the same time, most of the analyzed publications focus either on individual libraries or on comparing two or three solutions, without systematically covering the entire current spectrum of tools. This confirms the need for a comprehensive comparative analysis covering Provider, Riverpod, Cubit, and MobX within a single frame of reference.

Research objective. Despite the availability of a large number of state management approaches in Flutter, there are gaps in the systematic comparison of their efficiency, developer ergonomics, and impact on application performance. Many studies are limited to describing individual patterns or comparing only popular approaches, without considering the comprehensive impact on stability, scalability, and code maintainability (Foresight Mobile, 2025). The objective of this study is to conduct a comprehensive analysis of modern state management methods in Flutter – Provider, Riverpod, Cubit, and MobX – from the perspective of

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

their architectural characteristics, performance metrics, and usability, as well as to determine optimal usage scenarios for each approach for different types of mobile applications. The research results should provide developers with practical recommendations for selecting state management approaches, thereby improving the quality and stability of the software product (Kumar, 2025).

Main research findings. Since direct performance measurements require experimental implementation, we refer to existing benchmark studies for comparing state management solutions relevant to Flutter. Performance research shows that state management libraries differ in the efficiency of interface rendering processing and CPU load under similar workloads (iCoderz, 2025; JOTI, 2023).

These differences are not merely incidental: they reflect fundamental architectural decisions embedded in each library's design, specifically, how and when state changes are propagated to the widget tree. Libraries that rely on fine-grained reactive tracking tend to minimize unnecessary rebuilds, whereas those built around explicit event streams introduce structural overhead that, while beneficial for traceability and testability, carries a measurable runtime cost. Understanding the quantitative magnitude of these trade-offs is essential for making informed architectural decisions in production Flutter development.

The benchmark methodology employed in the referenced studies involves constructing functionally equivalent application prototypes, isomorphic in their UI structure and data operations, and subjecting each to identical workloads, typically list rendering and state update operations across datasets of varying size. This controlled approach isolates library-specific overhead from application logic, yielding comparable baseline metrics. According to this benchmark data, average frame rendering times (in milliseconds) and CPU usage trends for common Flutter state management libraries are presented in Table 1:

Table 1

Comparative performance metrics of Flutter state management libraries

State Management Solution	**Frame Rendering Time (ms)**	**CPU Usage**
Provider	–16 ms	Moderate
Riverpod	–12 ms	Low
MobX	–18 ms	Medium
Bloc/Cubit	–20 ms	High

These data suggest that Riverpod demonstrates comparatively more efficient UI update performance under similar conditions, providing lower frame rendering time and lower CPU load. The formal theoretical basis for such state-transition behavior can be found in the timed automata model (Alur & Dill, 1994), which provides a mathematical framework for reasoning about asynchronous event-driven transitions. MobX shows moderate performance with acceptable response times and slightly higher rendering times, while Bloc (and consequently

Cubit as its simplified variant) produces higher rendering times and CPU load due to the overhead of event and stream processing (iCoderz, 2025).

Additional studies focusing on comparing Provider and Riverpod show that Riverpod consistently maintains a slight advantage in CPU usage and memory efficiency compared to Provider under identical application workloads across various data volumes (1,000–10,000 elements) (JOTI, 2023). Although these benchmarks do not measure Cubit separately from Bloc, it is widely known that Bloc (and Cubit as part of the Bloc family) provides a structured event-driven state flow at the cost of additional processing, which may affect performance in scenarios with frequent state changes (Shakil, 2025).

Provider: architectural features and practical application. Provider is one of the most widely used and officially recommended state management solutions in Flutter. This library is built on top of the built-in InheritedWidget mechanism and significantly simplifies its usage, providing a convenient API for dependency injection and state propagation through the widget tree (Flutter Official Docs, 2025).

Due to its conceptual simplicity and native integration with the Flutter SDK, Provider has gained widespread use primarily in small and medium-scale projects. From a technical standpoint, Provider implements the ChangeNotifier pattern, in which the state object inherits the ChangeNotifier class and calls the notifyListeners() method on each change. Widgets registered via Consumer or Provider.of() are automatically rebuilt on each such notification. The advantage of this approach is its transparency and minimal boilerplate code. In practice, Provider works best in scenarios such as managing a user profile in an authentication application, synchronizing application theme across multiple interface components, or managing a simple shopping cart (Adam, 2025). However, Provider has significant limitations in large and architecturally complex projects: it lacks compile-time safety and is prone to excessive rebuilds unless additional optimizations – such as precise Selectors or ProxyProviders – are applied. The average frame rendering time when using Provider is approximately 16 ms under moderate CPU load (iCoderz, 2025; Kumar, 2025).

From a formal perspective, Provider can be interpreted as a reactive system based on explicit notification-driven state propagation, where each state change triggers updates in all subscribed components. Unlike more advanced approaches with fine-grained dependency tracking, Provider does not inherently differentiate between direct and indirect dependencies, which may result in redundant widget rebuilds.

This characteristic reflects a comparatively coarse-grained transition model, where state changes are propagated uniformly rather than selectively. While such an approach ensures conceptual clarity and ease of implementation, it limits the efficiency of update propagation in applications with complex state structures.

Within the proposed formalization, Provider can therefore be classified as a system with explicit but non-optimized transition relations, where the absence of structured dependency tracking reduces determinism and scalability under increasing system complexity. Consequently, although Provider remains an effective solution for rapid development and low-

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

complexity scenarios, its architectural simplicity imposes constraints on performance optimization and maintainability in large-scale applications.

Cubit: simplified implementation of event-driven state management. Cubit is a simplified variant of the Bloc library, included in the flutter_bloc package. Unlike full Bloc, Cubit does not use a formal event scheme – instead, it offers direct method calls that change state. This significantly reduces the amount of boilerplate code and lowers the entry threshold, while preserving the core advantages of the pattern: clear separation of business logic from the interface, reactive UI updates via BlocBuilder, and convenient integration with DevTools for state transition monitoring (Shakil, 2025). The event-driven, unidirectional state management pattern of Bloc and Cubit aligns with the behavioral design patterns described by Gamma et al. (1994), particularly the Observer pattern, which underpins reactive widget rebuilding.

Bloc, in turn, implements a strict unidirectional data flow scheme: interface components send events (Events) to the Bloc class, which processes these events and emits new states (States) via streams (Streams). Such an approach ensures complete transparency of data flow, making each state change clearly traceable and reproducible, making debugging and testing significantly more convenient. Bloc is widely used in large-scale enterprise applications, particularly in fintech solutions, where transaction management, authentication, and notifications are implemented through separate Bloc modules with detailed logging of each state transition (Kumar, 2025; SolGuruz, 2025). Cubit thus occupies a niche between Provider and Bloc, perfectly suited for implementing forms with validation, paginated data loading, and managing asynchronous operations while maintaining testability and separation of concerns.

MobX: reactive dependency tracking in Flutter. MobX is a reactive state management library, originally developed for JavaScript and successfully ported to Dart and Flutter (Meijer, 2010). Its key concept, transparent reactive programming, consists in the framework automatically tracking which parts of the interface access which observable state and, upon any change, precisely updating only the dependent components. The developer declares class fields as @observable, methods that change state as @action, and computed values as @computed. The entire reaction mechanism occurs automatically, without the need to manually notify dependent components (iCoderz, 2025).

MobX usage is particularly justified in scenarios with complex, interdependent state. For example, in a task management application, the completion state of a task, the display filter, and the counter of incomplete tasks are interconnected: changing any of them must cascade updates to the corresponding parts of the UI. MobX handles such relationships elegantly and automatically. MobX performance at 18 ms frame rendering time and moderate CPU load places the library between Riverpod and Bloc. MobX is an attractive solution for developers familiar with the JavaScript ecosystem and reactive programming patterns (Shakil, 2025; iCoderz, 2025).

From a formal standpoint, MobX can be interpreted as a reactive computational system with an implicit dependency graph, where state transitions propagate through dynamically established relationships between observables and observers. This distinguishes it from event-driven approaches such as Bloc or Cubit, where transitions are explicitly defined and pro-

cessed sequentially. In MobX, the order and scope of updates are determined by the structure of dependencies, which reduces the amount of boilerplate code but introduces a less explicit transition model.

Such an approach aligns with the general principle of fine-grained reactivity, where only those components that directly depend on the modified state are updated, minimizing unnecessary widget rebuilds. At the same time, the implicit nature of dependency tracking may reduce transparency of data flow, which can complicate debugging and formal verification of determinism in large-scale systems. Therefore, within the proposed formalization, MobX can be classified as a system with partially implicit transition relations, where the propagation of state changes is governed by reactive bindings rather than explicitly defined state transitions.

Riverpod: advanced dependency management architecture. Riverpod is the successor to Provider, developed by the same author – Remi Rousselet. Despite their shared conceptual roots, Riverpod is a fundamentally reconceived architecture, not simply an improved version of its predecessor. The key difference is that Riverpod places providers outside the widget tree, making them globally accessible regardless of where in the hierarchy the call is located. This eliminates a class of errors related to missing providers in context and ensures compile-time checks, which is critical in large team projects (Foresight Mobile, 2025).

The Riverpod architecture is based on the concept of providers as immutable references to state or logic. The key object for interaction is ref, an object that allows reading, observing, and tracking dependencies between providers. The declarative style of Riverpod allows conveniently describing asynchronous operations through AsyncNotifier and FutureProvider, automatically handling loading and error states without additional boilerplate. The library also supports family providers, which allow parameterizing providers and effectively managing state for dynamically created objects. Riverpod demonstrates its strengths in scenarios with intensive asynchronous data loading such as news aggregation applications and IoT platforms with numerous data streams (Kumar, 2025).

Demonstrating an average frame rendering time of approximately 12 ms and low CPU load, Riverpod confirms its advantage over Provider and other reviewed solutions. This is explained by a more intelligent dependency tracking mechanism that allows rebuilding only those parts of the tree that actually depend on the changed state, avoiding cascade rebuilds. Riverpod encourages a clear separation of responsibilities, where each provider is responsible for a specific part of state or business logic, making the code modular, easily testable, and convenient for reuse (JOTI, 2023; iCoderz, 2025).

From a formal perspective, Riverpod can be interpreted as a reactive system with explicitly defined dependency relations, where providers form a directed acyclic graph of state dependencies. Each state transition is propagated through this graph in a controlled and predictable manner, ensuring a high degree of determinism compared to approaches with implicit reactivity. Unlike MobX, where dependencies are inferred dynamically, Riverpod requires explicit declaration of relationships between providers, which improves transparency and traceability of data flow.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

This explicit dependency model aligns with the formalization of state management as extended transition systems, where each provider represents a node with clearly defined inputs and outputs. As a result, Riverpod simplifies reasoning about system behavior under asynchronous updates and facilitates verification of correctness properties such as consistency and absence of unintended side effects. Furthermore, compile-time safety mechanisms reduce the likelihood of runtime errors, which is particularly important in large-scale and team-based development environments.

Overall, Riverpod can be classified as a highly deterministic and scalable state management solution, combining efficient dependency tracking with strong architectural guarantees, making it well-suited for complex applications with intensive data flows and strict requirements for maintainability.

Conclusions. Taking into account the published research results, it can be concluded that Riverpod demonstrates the most balanced performance metrics among the reviewed solutions, particularly in scenarios with frequent interface updates and dependency management (iCoderz, 2025; JOTI, 2023). MobX provides moderate performance through reactive dependency tracking, which can reduce the number of unnecessary rebuilds with proper configuration. Bloc and Cubit demonstrate stable but comparatively higher processing costs due to the heavier architectural pattern oriented toward large-scale and structured applications (Kumar, 2025; SolGuruz, 2025).

Provider in additional studies falls behind Riverpod in CPU usage and memory efficiency under identical workloads (JOTI, 2023), but remains the optimal choice for small-scale projects due to its minimal entry threshold. Since the considered differences are measured under controlled benchmark conditions, the practical impact of library choice depends on specific scenarios: dataset size, frequency of asynchronous updates, and UI architecture complexity (iCoderz, 2025).

The practical choice between Provider, Riverpod, Cubit, and MobX should be based on a comprehensive analysis of project scale, testability requirements, team experience, and subject domain specifics (Flutter Official Docs, 2025; Foresight Mobile, 2025). Prospects for further research include conducting own performance measurements in real production conditions with specified load scenarios, as well as extending the analysis to include newer solutions, particularly GetX and Signals, which are gaining popularity in the Flutter community as of 2024–2026. It is also advisable to study the impact of state management library choice on test coverage quality and the costs of long-term codebase maintenance in team projects.

ЛІТЕРАТУРА

1. Shakil M. Top 5 Flutter State Management Solutions 2025. Complete Guide [Electronic resource]. Medium, 2025. URL: https://medium.com/@mshakilawan735/top-5-flutter-state-management-solutions-2025.

2. Alur R., Dill D. L. A theory of timed automata. Theoretical Computer Science. 1994. Vol. 126, No. 2. P. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

3. Baier C., Katoen J.-P. Principles of model checking. MIT Press, 2008. DOI: 10.7551/mitpress/9780262026499.001.0001.

4. Adam T. State Management Solutions: Comparing Provider, Riverpod, BLoC, and MobX [Electronic resource]. Medium, 2025. URL: https://medium.com/@tofiqueadam/state-management-solutions-comparing-provider-riverpod-bloc-and-mobx.
5. Flutter Official Docs. State Management Options [Electronic resource]. Google / Flutter Team, 2025. URL: https://docs.flutter.dev/data-and-backend/state-mgmt/options.
6. Gamma E., Helm R., Johnson R., Vlissides J. Design patterns: Elements of reusable object-oriented software. Addison-Wesley, 1994. DOI: 10.5555/186897.
7. Harel D., Pnueli A. On the development of reactive systems. In: Logics and models of concurrent systems. Springer, 1985. P. 477–498. DOI: 10.1007/978-3-642-82453-1_17.
8. JOTI. State Management Comparison in Flutter [Electronic resource]. Jurnal Online Teknik Informatika, Universitas Dinamika, 2023.
URL: https://e-journals.dinamika.ac.id/joti/article/view/1164.
9. Foresight Mobile. Best State Management for Flutter [Electronic resource]. Foresight Mobile, 2025. URL: https://foresightmobile.com/blog/whats-the-best-state-management-library-for-flutter.
10. Meijer E. Your mouse is a database. Communications of the ACM. 2010. Vol. 53, No. 4. P. 66–73. DOI: 10.1145/1721654.1721672.
11. Kumar A. Mastering State Management in Flutter: GetX vs Riverpod vs Bloc vs Provider (2025 Comparison) [Electronic resource]. Medium, 2025.
URL: https://medium.com/@anilkumar2681/mastering-state-management-in-flutter-getx-vs-riverpod-vs-bloc-vs-provider-2025-comparison-a48429710b96.
12. iCoderz Solutions. Top Flutter State Management Packages of 2025 [Electronic resource]. iCoderz Solutions, 2025. URL: https://www.icoderzsolutions.com/blog/flutter-state-management-packages.
13. SolGuruz. Flutter State Management Packages, Best Practices, and More [Electronic resource]. SolGuruz, 2025. URL: https://solguruz.com/blog/flutter-state-management/.

REFERENCES

1. Shakil, M. (2025). Top 5 Flutter state management solutions 2025: Complete guide. Medium. https://medium.com/@mshakilawan735/top-5-flutter-state-management-solutions-2025
2. Alur, R., & Dill, D. L. (1994). A theory of timed automata. Theoretical Computer Science, 126(2), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8
3. Baier, C., & Katoen, J.-P. (2008). Principles of model checking. MIT Press. https://doi.org/10.7551/mitpress/9780262026499.001.0001
4. Adam, T. (2025). State management solutions: Comparing Provider, Riverpod, BLoC, and MobX. Medium. https://medium.com/@tofiqueadam/state-management-solutions-comparing-provider-riverpod-bloc-and-mobx
5. Flutter Team. (2025). State management options. https://docs.flutter.dev/data-and-backend/state-mgmt/options
6. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley. https://doi.org/10.5555/186897

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

7. Harel, D., & Pnueli, A. (1985). On the development of reactive systems. In Logics and models of concurrent systems (pp. 477–498). Springer. https://doi.org/10.1007/978-3-642-82453-1_17
8. JOTI. (2023). State management comparison in Flutter. Jurnal Online Teknik Informatika. https://e-journals.dinamika.ac.id/joti/article/view/1164
9. Foresight Mobile. (2025). Best state management for Flutter. https://foresightmobile.com/blog/whats-the-best-state-management-library-for-flutter
10. Meijer, E. (2010). Your mouse is a database. Communications of the ACM, 53(4), 66–73. https://doi.org/10.1145/1721654.1721672
11. Kumar, A. (2025). Mastering state management in Flutter: GetX vs Riverpod vs Bloc vs Provider (2025 comparison). Medium. https://medium.com/@anilkumar2681/mastering-state-management-in-flutter-getx-vs-riverpod-vs-bloc-vs-provider-2025-comparison-a48429710b96
12. iCoderz Solutions. (2025). Top Flutter state management packages of 2025. https://www.icoderzsolutions.com/blog/flutter-state-management-packages/
13. SolGuruz. (2025). Flutter state management packages, best practices, and more. https://solguruz.com/blog/flutter-state-management/

Received 30.03.2026.
Accepted 02.04.2026.
Published 30.04.2026

Порівняльний аналіз моделей управління станом в мобільних застосунках, розроблених за допомогою Flutter

Метою цього дослідження є формалізація моделей управління станом у застосунках Flutter та їх оцінка за критеріями детермінізму переходів, продуктивності, масштабованості та архітектурної складності. Підходи до управління станом інтерпретуються як розширені перехідні системи з асинхронними подіями; властивості безпеки та життєздатності формалізуються за допомогою часової логіки. Формалізовану модель управління станом Flutter пропонується як клас реактивних обчислювальних систем з визначеними властивостями безпеки та життєздатності. Крім того, дослідження включає аналіз часу рендерингу кадрів та використання процесора на основі бенчмарків, щоб забезпечити емпіричне порівняння розглянутих підходів. Отримані результати додатково інтерпретуються в контексті реальних сценаріїв розробки, що дозволяє сформулювати практичні рекомендації щодо вибору відповідних архітектур управління станом залежно від масштабу та складності програми. Дослідження виявило значні відмінності між підходами в детермінізмі, продуктивності та архітектурній складності, демонструючи, що Riverpod забезпечує найбільш збалансовані характеристики продуктивності завдяки ефективному відстеженню залежностей, тоді як Bloc та Cubit забезпечують вищу структурну чіткість ціною збільшення обчислювальних витрат. MobX демонструє помірну продуктивність з перевагами в управлінні складним взаємозалежним станом, тоді як Provider залишається придатним для невеликих програм завдяки своїй простоті. Ці результати дозволяють обґрунтовано вибирати архітектуру на основі вимог виробничої програми, включаючи масштаб системи, обмеження продуктивності та потреби в обслуговуванні.

Ключові слова: управління станом, Flutter, реактивні системи, формалізація, Cubit, Riverpod, Provider, MobX, продуктивність, масштабованість.

Anastasiia Velma - master's student, department of software engineering, Kharkiv National University of Radio Electronics, Ukraine.
ORCID: https://orcid.org/0009-0006-9256-4226
Mykhailo Makarevych - master's student, department of software engineering, Kharkiv National University of Radio Electronics, Ukraine.
ORCID: https://orcid.org/0009-0006-4906-5261
Nataliia Golian - candidate of technical sciences, associate professor, department of software engineering, Kharkiv National University of Radio Electronics, Ukraine.
ORCID: https://orcid.org/0000-0002-1390-3116

Вельма Анастасія Олександрівна - магістрант кафедри програмної інженерії, Харківський національний університет радіоелектроніки, Україна.
ORCID: https://orcid.org/0009-0006-9256-4226
Макаревич Михайло Павлович – магістрант кафедри програмної інженерії, Харківський національний університет радіоелектроніки, Україна.
ORCID: https://orcid.org/0009-0006-4906-5261
Голян Наталія Вікторівна – к.т.н., доцент, доцент кафедри програмної інженерії, Харківський національний університет радіоелектроніки, Україна.
ORCID: https://orcid.org/0000-0002-1390-3116

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)