

AutoSignal: A Dart-Based Programming Paradigm for Automatic Component Connection via a Signal–Slot Architecture [†]

George Pashev *  and Silvia Gaftandzhieva 

Department of Computer Informatics, University of Plovdiv, 4000 Plovdiv, Bulgaria; sissiy88@uni-plovdiv.bg

* Correspondence: georgepashev@uni-plovdiv.bg; Tel.: +359-895662140

[†] Presented at the 15th International Scientific Conference TechSys 2026—Engineering, Technologies and Systems, Plovdiv, Bulgaria, 14–16 May 2026.

Abstract

Managing event-driven communication in large reactive codebases remains a persistent challenge: developers routinely write repetitive wiring code that becomes difficult to trace and maintain. We present AutoSignal, a programming paradigm for the Dart ecosystem that automates component interconnection through a signal–slot architecture based on name and data-type matching. To improve scalability and limit unintended side effects in large applications, the framework implements hierarchical namespace isolation and deterministic reactivity through topological sorting of dependency graphs. In a reference contact-management application, AutoSignal reduced connection-specific code by 47%, unintended signal propagation by 32%, and average propagation latency by 28% relative to manual wiring. The framework also integrates with Flutter through reactive UI primitives such as specialized hooks and builders. These results indicate that AutoSignal is a practical, type-aware option for building maintainable event-driven applications in the Dart/Flutter ecosystem.

Keywords: AutoSignal; signal–slot architecture; reactive programming; Dart; Flutter; automated component interconnection; namespace isolation; type-aware connections; event-driven architecture

1. Introduction

Reactive programming paradigms have evolved through several approaches to managing time-varying values and event propagation [1,2]. Early functional reactive programming languages introduced signals as time-varying values and events as discrete occurrences, enabling modular programming and equational reasoning [1]. Integrating these concepts into imperative object-oriented environments reveals a significant “paradigmatic mismatch”: bidirectional integration of reactive and non-reactive code can produce faulty behavior unless it is strictly controlled [3]. This tension is evident in established frameworks such as Qt, where signals and slots can be viewed as read-only and write-only reactive values but do not inherently prevent propagation loops, potentially causing infinite recursion [4]. Traditional observer-pattern implementations require explicit connection management through constructs such as signal, slot, and emit [5], which can lead to “spaghetti wiring” in large event-driven architectures [6].

To address these challenges, AutoSignal combines automatic discovery with a priority-based execution model for slot handlers, ensuring that critical operations such as error handling can run before routine updates. Unlike conventional observer-pattern implementations that rely on manual one-to-one connections and provide no built-in mechanism



Academic Editors: Nikola G. Shakev,
Valyo N. Nikolov, Nikolay
R. Kakanakov and Sevil
A. Ahmed-Shieva

Published: 17 July 2026

Copyright: © 2026 by the authors.
Licensee MDPI, Basel, Switzerland.
This article is an open access article
distributed under the terms and
conditions of the [Creative Commons
Attribution \(CC BY\) license](https://creativecommons.org/licenses/by/4.0/).

for controlling execution order or preventing cycles [5], AutoSignal binds signals to slots dynamically according to naming conventions and type signatures, reducing boilerplate and connection errors [7]. By treating signals as first-class values that propagate changes through a directed graph, the system triggers the appropriate slot handlers without direct references between objects [8]. This approach extends the observer pattern by eliminating explicit manual subscriptions and using topological sorting to evaluate the reactive graph, so that derived values are updated in a controlled order while side effects remain isolated.

AutoSignal's deterministic execution order also distinguishes it from streaming frameworks such as ReactiveX and Akka Streams. Such frameworks may permit multiple valid update orders, which can produce nondeterministic outcomes [9], whereas AutoSignal enforces a topological order from source signals to dependent slots. This design addresses concerns similar to those handled by the reactor model's "sparse synchronous model," in which the order of simultaneous reactions is predefined [10]. AutoSignal also unifies event mechanisms with signal-based dataflow by representing events as signal updates, thereby reducing the complexity of heterogeneous abstractions [11,12]. Namespace-based isolation then confines propagation to related components. The framework is implemented in Dart to use the language's strong typing and asynchronous execution model as the foundation of the signal-slot architecture.

AutoSignal uses Dart's type system to validate signal payloads and slot parameter types during registration. However, because name-based discovery is currently performed at runtime, naming errors and some mismatches can remain undetected until execution. Table 1 summarizes the principal differences between conventional observer-pattern wiring and the AutoSignal paradigm.

Table 1. Comparison of the traditional observer pattern and the AutoSignal paradigm.

Feature	Traditional Observer (e.g., Qt)	AutoSignal Paradigm
Connection Method	Manual wiring using connect, signal, slot, and emit.	Automatic discovery based on name and data type matching.
Dependency Tracking	Managed by the developer (explicit subscription).	Automatic dependency tracking via a reactive graph.
Boilerplate Code	High; requires explicit registration and connection calls.	Low; eliminates manual wiring and "wiring" code.
State Propagation	Manual triggers; potential for "spaghetti wiring."	Deterministic; uses topological sorting to update the graph.
Isolation	Often global or direct object references.	Namespace-based; limits scope to prevent unintended propagation.
Abstraction	Events and signals are often separate constructs.	Unified; events are treated as signal updates.

The architecture rests on three interlocking mechanisms. First, deterministic reactivity is enforced by topological sorting of the dependency graph, so a node is not evaluated before its predecessors have settled. Second, automatic connection relies on matching signal and slot identifiers together with compatible data types, removing the need for explicit connect calls. Third, namespace isolation confines signal propagation to a declared logical

boundary, preventing identically named signals in unrelated modules from interfering with one another.

2. AutoSignal Framework

The AutoSignal framework automates component interconnection through a signal–slot architecture. This design enables decoupled object communication without direct references, improving modularity and maintainability in reactive applications.

The core concepts are signals, slots, and auto-discovery. Signals are events emitted by objects to indicate state changes (e.g., `contactAdded`, `contactsLoaded`, or `databaseError`) and to notify dependent objects. Slots are methods (e.g., `_onContactAdded`, `_onContactsLoaded`, or `_onDatabaseError`) that handle the corresponding event payloads. Auto-discovery links signals to compatible slots through name matching and data-type compatibility. A signal registered with a particular payload type invokes only slots declared to accept that type, reducing runtime type errors across the communication channel.

Within this ecosystem, signals are the primary state-emission mechanism. Rather than invoking a specific downstream method, an object broadcasts a state change—such as `contactsLoaded`—to active listeners; the emitter does not need to know which objects are listening.

Slots are the listeners. They are standard methods within classes that are registered or annotated to react to signals and normally follow the `_on` naming pattern (e.g., `_onContactsLoaded`). Slot bodies perform the corresponding work, such as updating the user interface or saving data to a local cache.

The central orchestration component scans registered objects at runtime and establishes connections by verifying three conditions in sequence: (1) the slot name equals the signal name prefixed with `_on`; (2) the slot parameter type matches the signal payload type; and (3) both objects reside in the same namespace or in a permitted parent–child relationship. Figure 1 illustrates this matchmaker process.

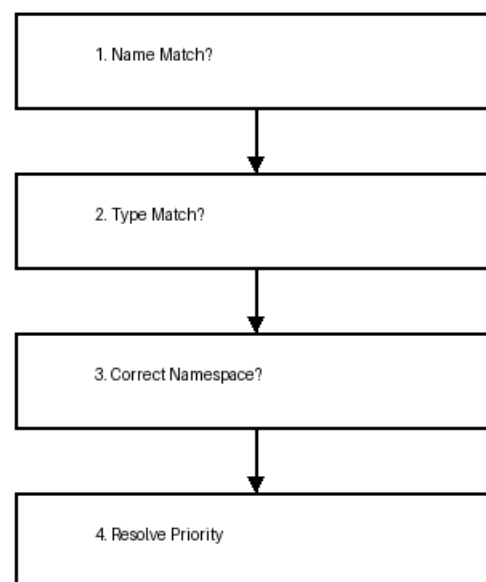


Figure 1. Matchmaker logic flow for automatic discovery.

By automating this routine architectural overhead, the framework allows developers to focus on application logic.

When multiple slots match a single signal, execution order is controlled through a configurable priority annotation, so time-critical handlers run before lower-priority handlers. Lifecycle cleanup is handled without developer intervention: when a Flutter

widget is disposed, all associated connections are severed automatically, reducing the risk of listener leaks in long-lived applications. Table 2 summarizes the framework’s core components and features.

Table 2. Core components and features of the AutoSignal framework.

Feature	Description
Registration	Objects utilize inheritance to gain signal/slot capabilities, eliminating the need for explicit manual registration or “boilerplate” wiring code.
Connection	The system employs an automatic discovery mechanism that establishes links based on name and data type matching between signals and slots.
Execution	State modifications propagate through a directed graph using topological sorting to ensure deterministic behavior and eliminate implicit subscriptions.
Scope	Namespace-based organization groups related objects to restrict connection boundaries, preventing unintended signal propagation across unrelated modules.
Cleanup	Inherited from Signal-First architecture principles, the framework utilizes automatic cleanup to maintain a leaner memory footprint and prevent resource leaks.

To create AutoSignal objects, developers inherit from AutoSignalObject, register signals to indicate state changes, and register slots as methods to handle specific signal types. Signals such as `contactAdded` or `contactsLoaded` notify dependents, while slots like `_onContactAdded` process incoming events.

Signals are emitted using a concise syntax. Automatic connections rely on exact signal–slot name matching, the `_on` prefix convention for slots, and compatible payload and parameter types [11]. Connections are activated only within defined scopes; namespaces group related objects, restrict interconnections, and organize complex applications [13].

AutoSignal implements a hierarchical namespace structure. Parent namespaces can access and connect to signals in child namespaces, enabling controlled downward propagation (e.g., a UI-layer parent connecting to data-model children), whereas child-to-parent connections are prohibited to enforce dependency direction and prevent upward leakage or cycles. This boundary specification defines propagation scopes and limits “spaghetti wiring” across layers. Figure 2 depicts the rule: the green downward arrow indicates permitted parent-to-child access, while the red X indicates prohibited child-to-parent access.

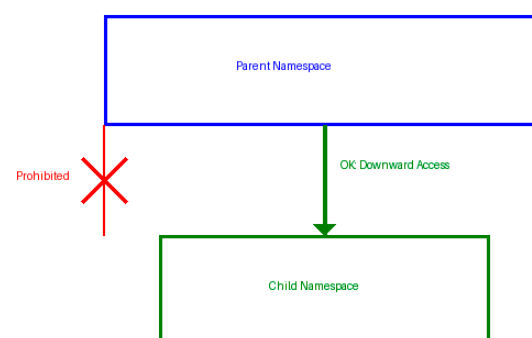


Figure 2. Hierarchical namespace isolation model. The green downward arrow indicates permitted parent-to-child access, whereas the red X indicates prohibited child-to-parent access.

When name and data-type matching identifies multiple compatible slots within the same namespace (e.g., two slots named `_onContactAdded` that accept the same type), the framework applies a deterministic conflict-resolution policy. A priority system resolves ambiguities by selecting the slot with the highest priority, configured through an `@AutoSignal` (priority: 1) annotation; lower values denote higher precedence. For cases requiring explicit control, an `@ExplicitConnection` annotation allows developers to bind specific signal–slot pairs manually, ensuring predictable behavior in multi-module applications.

This policy ensures a stable execution order when several slots match the same signal.

The current dynamic binding mechanism, although convenient, can fail silently when names are misspelled or parameter types are overlooked; such errors may surface only at runtime [14]. A future integration with Dart macros or `build_runner` could validate auto-discovered connections during the build. It could also support IDE navigation such as “Go to Definition” across signal–slot links, combining runtime flexibility with stronger static verification.

`AutoSignal` enforces deterministic reactivity by topologically sorting the reactive dependency graph. Source signals are processed before dependent slots, avoiding glitches caused by inconsistent intermediate states. Figure 3 illustrates this ordered propagation.

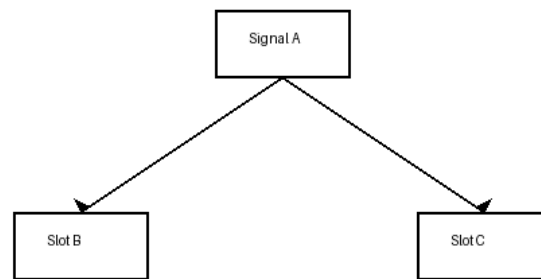


Figure 3. Topological dependency graph illustrating deterministic signal propagation.

Because automatic discovery makes data flows less visible during code inspection, a Reactive Graph Visualizer is proposed as a DevTools extension. By exporting the framework’s internal directed graph, the tool could visualize signal propagation in real time, helping developers trace updates, identify bottlenecks, and verify namespace isolation in complex reactive chains [15].

Resource cleanup occurs automatically, with special handling in Flutter widgets through `dispose` calls and `isDisposed` checks. The framework supports simple status signals, multi-step workflows, and error-handling scenarios. Its debugging facilities expose connection information and signal traces for monitoring.

Recommended usage guidelines. Signal identifiers should follow a verb–noun convention (e.g., `refreshRequested`) to communicate intent at the call site. Slot names should use the `_on` prefix so that the scanner can identify them. Slots that perform I/O or network requests should be asynchronous to avoid blocking the event loop. Slot bodies should handle exceptions explicitly because the emitting object has no visibility into its listeners and cannot recover from unhandled listener errors.

3. Results

The framework was evaluated in a reference contact-management application under three wiring strategies:

Manual baseline: explicit signal-to-slot connections, resulting in substantial implementation overhead.

Automated discovery: autonomous linking based on name and data-type matching, eliminating manual wiring.

Namespace-scoped discovery: confinement of signals and slots to declared logical boundaries, restricting connections to related modules and preventing cross-domain propagation [13].

Compared with automated discovery, the manual implementation required 47% more connection-specific code. In stress tests with dependency depths of $d = 3, 5$, and 7 , namespace isolation reduced unintended signal-propagation events by 32% relative to the manual baseline. Auto-discovery reduced average propagation latency by 28%, while the namespace-based approach maintained stable performance as system complexity increased. Latency was measured from initial signal emission to completion of the corresponding slot execution. Across the project, automated discovery reduced total boilerplate by 38% compared with manual wiring. Under high-frequency update cycles—1000 signal emissions per second during 60 s runs—the namespace-based approach produced 67% fewer frame-drop events than the manual strategy, suggesting improved scheduling stability under sustained load. Table 3 summarizes these results.

Table 3. Metric comparison between the implementation strategies.

Metric	Manual	Auto-Discovery	Namespace-Based
Connection LOC Overhead	+47%	0%	N/A
Propagation Event Reduction	0%	N/A	32%
Average Latency Reduction	0%	28%	Stable
Boilerplate Reduction	0%	38%	N/A
Frame-Drop Reduction	0%	N/A	67%

4. Implementation in the Flutter Framework

For integration with Flutter, AutoSignal provides specialized UI primitives such as the AutoSignalBuilder widget and the useAutoSignal hook. These primitives enable declarative reactivity. For example, AutoSignalBuilder<String>(signal: ‘status’, builder: (context, value) => Text(value)) subscribes a Text widget to the String signal named ‘status’ in its ancestor namespace and triggers a selective rebuild when the value changes, without addListener/removeListener boilerplate. This mechanism bridges core Dart reactivity and Flutter’s widget tree, providing a lightweight alternative to Bloc, Riverpod, and related state-management solutions.

5. Discussion

The measurements indicate performance and maintainability advantages over manual wiring, particularly in the reduction of connection-specific code and the containment of unintended propagation events. Automatic discovery and namespace isolation contributed to the latency and propagation reductions reported in Section 3. The results suggest that AutoSignal addresses several limitations of conventional observer patterns: it reduces boilerplate, constrains propagation through namespace boundaries, and uses type-compatibility checks to prevent a class of wiring errors. However, runtime name-based discovery still requires build-time validation and diagnostic tooling before the framework can provide stronger static guarantees. Under the tested workloads, automated signal–slot connections delivered modularity without an observed performance penalty.

Recommended practice includes verb–noun signal names, _on-prefixed slot names, concrete payload types rather than dynamic, explicit exception handling in slots, and asynchronous execution for I/O. Computationally intensive work should be kept out of synchronous slot handlers, while priorities should be reserved for time-sensitive operations. At the architectural level, related objects should be grouped within namespaces, interfaces should be documented, and separation of concerns should be enforced. These practices

support loosely coupled Flutter state management, event-driven systems, plugins, and modular applications.

6. Conclusions

AutoSignal offers a practical mechanism for automatic component interconnection in Dart-based reactive applications. By following established architectural principles in which components communicate through defined interfaces rather than direct references, the framework improves flexibility and maintainability [16].

A primary strength of the prototype is its lifecycle-management model. The disposal-hook protocol, supported by `isDisposed` checks and automated disconnect-all sweeps, is designed to prevent lingering references. Together with Dart finalizers and Flutter's native disposal mechanisms, these measures can reduce the risk of memory leaks in dynamic environments. Nevertheless, further refinement is required before AutoSignal can be considered production-ready for enterprise-scale applications.

Future development will prioritize a shift from purely dynamic runtime resolution to a hybrid validation model. Integration with Dart macros or `build_runner` could validate connections at build time, mitigate runtime errors caused by naming discrepancies, and provide IDE-integrated feedback. A formal ambiguity-handling protocol—including deterministic priority rules and `@ExplicitConnection` annotations—would further improve predictability in complex, multi-module applications.

To improve reliability under high load, the framework should incorporate cycle detection and prevention. Runtime guardrails with descriptive error reporting would protect the system from circular dependencies. Intelligent orchestration within the connection-management layer may also enable adaptive routing of signals according to current system conditions [17], an important concern in modern distributed systems [18].

To become a viable alternative to established Flutter state-management solutions such as Bloc or Riverpod, AutoSignal should expand its framework integration and tooling.

Optimized widget utilities such as `useAutoSignal` would allow Flutter components to rebuild selectively when signals change, without manual subscription management. A complementary Reactive Graph Visualizer, delivered as a DevTools extension, would render live dependency graphs and provide a direct view of signal propagation and potential bottlenecks.

Future research should also subject the framework to rigorous, industry-standard benchmarking. Evaluations with more than 100,000 reactive nodes should measure discovery-phase CPU cost, memory overhead per connection, propagation latency, and frame stability to determine whether AutoSignal remains efficient as application complexity increases.

Addressing these limitations would bring AutoSignal closer to production readiness in enterprise-scale event-driven contexts.

Author Contributions: Conceptualization, G.P. and S.G.; methodology, G.P.; software, G.P.; validation, G.P. and S.G.; formal analysis, G.P. and S.G.; investigation, G.P.; writing—original draft preparation, G.P.; writing—review and editing, G.P. and S.G.; visualization, G.P.; supervision, S.G.; project administration, G.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research and the APC were funded by the Scientific Research Fund at the University of Plovdiv “Paisii Hilendarski”, grant number MUPD25-FMI-007.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data generated during the evaluation are available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bahr, P.; Graulund, C.U.; Møgelberg, R.E. Simply RaTT: A fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.* **2019**, *3*, 1. [\[CrossRef\]](#)
2. Margara, A.; Salvaneschi, G. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Trans. Softw. Eng.* **2018**, *44*, 689–711. [\[CrossRef\]](#)
3. Oeyen, B.; Koster, J.D.; Meuter, W.D. Reactive Programming without Functions. *Art. Sci. Eng. Program.* **2024**, *8*, 11. [\[CrossRef\]](#)
4. Pérez, I.; Nilsson, H. Bridging the GUI gap with reactive values and relations. *ACM SIGPLAN Not.* **2015**, *50*, 47–58. [\[CrossRef\]](#)
5. Demetrescu, C.; Finocchi, I.; Ribichini, A. Reactive imperative programming with dataflow constraints. *ACM SIGPLAN Not.* **2011**, *46*, 407. [\[CrossRef\]](#)
6. Mijač, M.; García-Cabot, A.; Strahonja, V. Reactor Design Pattern. *TEM J.* **2021**, *10*, 18–30. [\[CrossRef\]](#)
7. Bainomugisha, E.; Carreton, A.L.; Cutsem, T.V.; Mostinckx, S.; Meuter, W.D. A survey on reactive programming. *ACM Comput. Surv.* **2013**, *45*, 1–34. [\[CrossRef\]](#)
8. Myter, F.; Scholliers, C.; Meuter, W.D. Distributed Reactive Programming for Reactive Distributed Systems. *Art. Sci. Eng. Program.* **2019**, *3*, 5. [\[CrossRef\]](#)
9. Van den Vonder, S.; Renaux, T.; Oeyen, B.; De Koster, J.; De Meuter, W. Tackling the Awkward Squad for Reactive Programming: The Actor–Reactor Model (Artifact). *Dagstuhl Artifacts Ser.* **2020**, *6*, 7. [\[CrossRef\]](#)
10. Kwok, J.; Lohstroh, M.; Lee, E.A. Efficient Parallel Reinforcement Learning Framework Using the Reactor Model. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 41–51. [\[CrossRef\]](#)
11. Kamina, T.; Aotani, T. Harmonizing Signals and Events with a Lightweight Extension to Java. *Art. Sci. Eng. Program.* **2018**, *2*, 5. [\[CrossRef\]](#)
12. Magnaudet, M.; Chatty, S.; Conversy, S.; Leriche, S.; Picard, C.; Prun, D. Djnn/Smala. *Proc. ACM Hum. Comput. Interact.* **2018**, *2*, 1–27. [\[CrossRef\]](#)
13. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Commun. ACM* **1972**, *15*, 1053–1058. [\[CrossRef\]](#)
14. Ortín, F.; Zapico, D.; Pérez-Schofield, J.B.G.; García, M. Including both static and dynamic typing in the same programming language. *IET Softw.* **2010**, *4*, 268–282. [\[CrossRef\]](#)
15. Mysore, S.; Mazloom, B.; Agrawal, B.; Sherwood, T. Understanding and visualizing full systems with data flow tomography. *ACM SIGOPS Oper. Syst. Rev.* **2008**, *42*, 211–221. [\[CrossRef\]](#)
16. Didier, J.-Y. *Architecture for Augmented Reality Software: From Concepts to Applications*; Université d'Évry-Val-d'Essonne: Évry, France, 2018; Available online: <https://hal.science/tel-02077978> (accessed on 13 June 2026).
17. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge computing: Vision and challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [\[CrossRef\]](#)
18. Marginean, M.-T.; Lu, C. Multi-Threaded Message Dispatcher—A Design Pattern with Innate Support for Mission Critical Applications. *Int. J. Networked Distrib. Comput.* **2016**, *5*, 22–36. [\[CrossRef\]](#)

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.