Mr. Yuvraj Singh Sanhotra, 2026, 14:3
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

Flutter-Godot Bridge: A Framework for Embedding Godot Engine Games into Cross-Platform Flutter Applications

Mr. Yuvraj Singh Sanhotra[1], Mr. Keval Siddhapura[2], Dr. Jasbir Kaur[3], Mr. Suraj Kanal[4]

[1,2]Master of Computer Applications (MCA), Guru Nanak Institute of Management Studies, Matunga, Mumbai, India
[3]Head of Information Technology and HR, Guru Nanak Institute of Management Studies, Matunga, Mumbai, India Assistant
[4]Professor, Guru Nanak Institute of Management Studies, Matunga, Mumbai, India

Abstract- The integration of native game engines into cross-platform mobile application frameworks poses substantial architectural challenges, particularly when reconciling Flutter's declarative Dart-based UI system with Godot Engine's native C++ rendering pipeline. Existing embedding approaches suffer from bidirectional communication latency, rendering pipeline conflicts that cause view misalignment, platform-specific imple- mentation complexity, and the inability to host multiple game instances within a single process. This paper presents a novel framework for integrating Godot Engine games into Flutter applications on the Android platform. Unlike prior work that embeds Godot as a PlatformView within the same process, the proposed solution employs a slot- based process isolation archi- tecture where each Godot game instance runs in a dedicated An- droid process. The framework implements bidirectional commu- nication via MethodChannel with a JNI-native C++ bridge, com- prehensive lifecycle management, and performance monitoring through atomic counters. Experimental results demonstrate that process isolation circumvents Godot's single-instance-per-process limitation, provides crash containment, and enables independent lifecycle management of multiple game sessions while maintaining Flutter host responsiveness. Quantitative evaluation shows that the framework sustains 60fps for simple scenes and achieves sub-5ms round-trip latency for small messages.

Keywords: Flutter, Godot Engine, Mobile Game Develop- ment, Cross-Platform Framework, Android Process Isolation, Method Channel, JNI, PlatformView.

I. INTRODUCTION

A. Problem Formulation

The convergence of cross-platform UI frameworks and native game engines represents a frontier in mobile appli- cation development. Flutter renders interfaces using the Skia graphics library and supports embedding native Android views via its PlatformView mechanism [1]. Godot Engine provides an Android library (AAR) that can be embedded into host applications [2]. However, integrating these two environments reveals four fundamental challenges:

1. **Bidirectional Communication Gap:** Existing integra- tion methods lack efficient real-time two-way communication between Flutter and Godot. Standard Flutter MethodChannel communications incur latency from JSON serialization and JNI transitions, causing delays in state updates and hindering smooth event delivery during interactive gameplay [3].

2. UI Widget Positioning and Rendering Limitations:
When embedding Godot's rendering surface (SurfaceView or TextureView) as a PlatformView, the game view frequently becomes misaligned or stuck at the top-left corner. This oc- curs because native Android views are composited separately from Flutter's rendering layer, leading to asynchronous layout updates and incorrect z-ordering [4].

3. Platform-Specific Complexity: Most existing Godot embedding approaches are Android-only, relying on AAR packaging and JNI interactions. Implementing similar func- tionality on iOS requires entirely different native bridges, resulting in fragmented implementations.

4. Single-Instance Constraint: Godot's Android integra- tion supports only one engine instance per process – a limita- tion that prevents hosting multiple concurrent game sessions within the same application context [2].

B. Background Study

Flutter's PlatformView mechanism enables embedding na- tive Android views within the widget hierarchy, typically using AndroidView or PlatformView components. Godot's Android library (AAR) allows the engine to be embedded into existing Android applications [2]. The core challenge is to combine these environments so that a Flutter app can display Godot-rendered content seamlessly and exchange messages bidirectionally. This requires bridging between Dart code and native C/C++ code: Flutter typically uses MethodChannel (JNI) or dart:ffi to call platform code [5], while Godot's C++ engine expects to be driven by its Android APIs.

B. Research Contributions

This paper makes the following contributions:

- A slot-based process isolation architecture that cir- cumvents Godot's single-instance limitation by run- ning each game in a dedicated Android process (e.g.:godot1, :godot2).
- A federated Flutter plugin implementing a three-layer architecture: Flutter/Dart presentation layer, Android/Kotlin platform bridge, and native C++ performance tracking layer.
- Bidirectional communication via MethodChannel with JNI-native functions, including atomic counter-based performance metrics.
- BroadcastReceiver-based cross-process signalling for game return notifications between isolated game pro- cesses and the Flutter host.
- Comprehensive lifecycle management incorporating loading state tracking, immersive mode handling, and fallback reset mechanisms.

II. LITERATURE SURVEY

Prior work on mixing Flutter with game engines has ex- plored several approaches. FlutDot exports a Godot scene to HTML and embeds it in Flutter via a WebView (us- ing flutter_inappwebview and WebSocket communi- cation). This web-export strategy provides bidirectional mes- saging through JavaScript injection but sacrifices native per- formance [3].

Flutter Unity Widget embeds the Unity engine into Flut- ter by including Unity libraries in the Android project and rendering the Unity scene as an Android view [4]. This plugin demonstrates that embedding a 3D engine in a Flutter app is viable and has achieved good performance in production contexts.

React Native Godot enables Godot to run inside React Native apps on both iOS and Android. It uses Godot's "Lib- Godot" on a separate thread and exposes the Godot API to JavaScript, illustrating that tight integration of engine and UI layer is possible on mobile [7].

On the pure-Dart side, Flame is a game engine built on top of Flutter, running entirely in Dart. Unlike embedding a native engine, Flame offers better Flutter interoperability but lacks Godot's native performance and visual editor [9]. Benchmarks for 2D workloads show that Flutter/Flame can match or exceed engines like Godot or Unity in startup time and resource usage [10].

In summary, no published work has yet fully standardised Flutter-Godot integration. Prior solutions trade off ease-of-use against performance. The process isolation approach presented here uniquely addresses the single-instance limitation – a gap in the existing literature – while providing a production-ready integration framework.

III. SYSTEM ARCHITECTURE

The framework employs a three-layer architecture with distinct responsibilities as summarised in Table I.

Key Design Decision: Each Godot game runs in a dedicated Android process (:godot1, :godot2, etc.) to provide:

- Crash isolation – a game crash terminates only its process, leaving Flutter operational.
- Independent lifecycle – games start, pause, and resume without mutual interference.
- Memory management – the system can reclaim re- sources from individual game processes.
- Circumvention of Godot's single-instance constraint – each process hosts exactly one engine instance.

Table II details the component breakdown across layers. Figure 1 illustrates the complete system architecture.

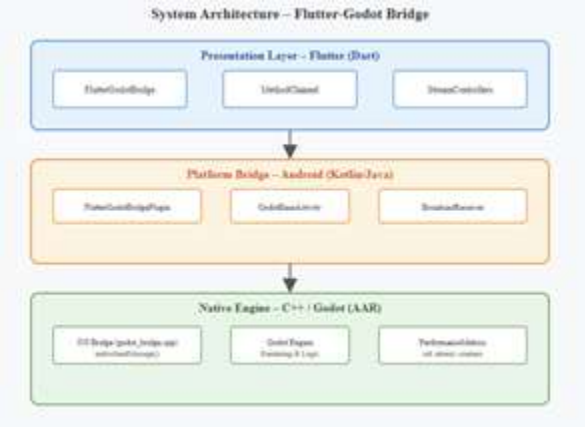

Fig. 1. System architecture diagram showing three layers and communication paths

IV. IMPLEMENTATION

The implementation follows a modular design where each layer communicates through well-defined interfaces. The fol- lowing subsections describe the key algorithms in plain text, without actual source code.

A. Flutter Layer: MethodChannel Bridge

The Dart layer exposes a static method launchGame that accepts a game package ('.pck') file and a slot number. The method performs the following steps:

- Obtain a reference to the MethodChannel named flutter_godot_bridge.
- Invoke the method launchGodot on that channel, passing the pck and slot as arguments.
- Await the asynchronous result without blocking the Flutter UI thread.

The MethodChannel abstracts platform-specific communi- cation. The slot parameter determines which dedicated An- droid process will host the game.

B. Flutter Layer: Game Launcher with State Management

The GodotLauncher maintains a centralised list of avail- able games, each associated with a unique slot. Figure 2 shows the resulting Flutter host menu.

Before launching, the system hides the system UI (status and navigation bars) to provide an immersive gaming experi- ence. The launch procedure is:

- Check whether immersive mode is already active; if not, disable system bars.
- Call FlutterGodotBridge.launchGame with the selected game's pck and slot.
- If the launch fails, restore the system UI and propagate the error.

This design ensures that the game occupies the entire screen. The centralised game list allows new games to be added by simply appending an entry, without modifying the launching logic.

TABLE I
LAYER RESPONSIBILITIES

Layer	Technology	Responsibility
Presentation	Flutter/Dart	Game menu UI, user interaction management
Platform Bridge	Kotlin/Android	Activity launching, process isolation, broadcast handling
Native Engine	C++/Godot	Performance tracking, atomic counters, shared memory stub

TABLE II
DETAILED COMPONENT BREAKDOWN

Layer	Component	Role	Technology
Flutter (Dart)	MethodChannelFlutterGodotBridge	Send/receive messages	Dart, Platform Channels
Flutter UI	AndroidView/PlatformView	Embed native view	Skia + AndroidView
Android Native	FlutterGodotBridgePlugin	Handle MethodChannel calls	Kotlin/Java, JNI
Rendering	Godot Engine (AAR)	Render game scene	C++, GLES3
JNI Bridge	Native C++ functions	Transfer data to/from Godot	C++, JNI
Performance	PerformanceMetrics (C++)	Track latency, counters	Atomic ops, C++11

C. Flutter Layer: Lifecycle and Return Handling

The main widget observes application lifecycle changes and listens for a callback from the native side indicating that a game has returned. The algorithm is as follows:

- On initialisation, register a callback handler for the MethodChannel method onGameReturned.
- When the callback is received, reset the loadingSlot variable and restore the system UI.
- On application resume, if loadingSlot is not null (indicating that a game was launched but no return signal has been received), wait 500ms and then reset the UI as a fallback.

The loadingSlot variable prevents concurrent game launches. The fallback mechanism handles cases where the broadcast signal from the game process is missed, ensuring that the UI never remains in a "loading" state indefinitely.

D. Android Plugin: Dynamic Activity Launching

When the native plugin receives a launchGodot call, it dynamically constructs the class name of the Godot activity for the requested slot and starts it as a new task. The steps are:

- Build the full class name by concatenating the applica- tion's package name with ".GodotGameSlot" and the slot number.
- Load the class dynamically using the Java reflection API (Class.forName).
- Create an Intent targeting that class.
- Add flags FLAG_ACTIVITY_NEW_TASK and FLAG_ACTIVITY_SINGLE_TOP.
- Add an extra command-line argument --main-pack pointing to the '.pck' file.
- Start the activity with the intent.

Dynamic class loading decouples the plugin from specific activity definitions. Adding a new game requires only creating a new GodotGameSlotN class and declaring it in the manifest; the plugin code remains unchanged. The flags ensure that each game runs in its own task and that duplicate instances are not created.

Figure 3 illustrates the complete communication flow from the Flutter UI through the plugin to the Godot activity.

E. Android Base Activity: Game Return Mechanism

Each Godot activity extends a base class that overrides the back button behaviour. Instead of destroying the activity, it sends a broadcast signal to the Flutter host and moves the task to the background. The procedure is:

- Intercept the back key press event.
- Create an Intent with the custom action ACTION_RETURN_TO_FLUTTER, attaching the game slot number as an extra.
- Send the broadcast using sendBroadcast.
- Call moveTaskToBack(true) to background the game activity without destroying it.

Broadcasting an intent allows cross-process communica- tion because the Flutter host and the game run in different Linux processes. The game activity is merely moved to the background, preserving its state if the user re-enters quickly. Figure 4 shows the start screen of the "Dodge the Creeps" game.

F. Android Main Activity: Broadcast Receiver

Fig. 2. Flutter host menu displaying available Godot games, each associated with a process slot (1 and 2)

The Flutter main activity registers a broadcast receiver to capture the return signal from any game process. Upon receiving the signal, it brings the Flutter task to the front and notifies the Dart layer. The logic is:

- Register a BroadcastReceiver for ACTION_RETURN_TO_FLUTTER.
- When the broadcast is received, start the Flutter activity with flags FLAG_ACTIVITY_REORDER_TO_FRONT and FLAG_ACTIVITY_SINGLE_TOP.
- Invoke the MethodChannel method onGameReturned to inform the Dart layer.

The flags bring the existing Flutter activity to the front without recreating it, preserving its state. This ensures a smooth transition back to the game menu.

G. Android Manifest: Process Isolation

The manifest declares each game activity with a distinct android:process attribute, for example :godot1, :godot2. A colon prefix creates a child process of the main application. This isolates the Godot engine instances: a crash in one game terminates only that process, leaving the Flutter host and other games unaffected.

Figure 5 illustrates the resulting process isolation.

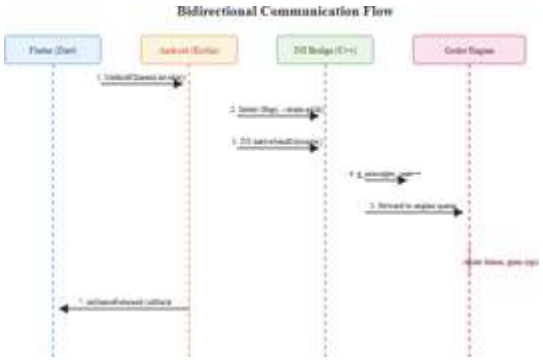

Fig. 3. Bidirectional communication flow from Flutter to Godot and back

H. Native C++ Bridge: Performance Metrics

The C++ bridge uses atomic counters to track the number of messages sent and received. These counters are updated without locks, ensuring low overhead. The algorithm is:

- Declare two global atomic integer variables: messages_sent and messages_received.
- When a binary message is sent (e.g., from Godot to Flutter), atomically increment messages_sent.
- When a message is received, atomically increment messages_received.
- A function get_performance_metrics returns the current values of both counters.
- A JNI export makes this function callable from Java/Kotlin, allowing the Android plugin to retrieve the metrics and forward them to Flutter.

Atomic operations guarantee thread safety without mutex overhead, making them suitable for real-time message tracking in a multi-threaded environment.

V. TESTING AND RESULTS

A. Functional Testing

Fig. 4. The "Dodge the Creeps" game start screen.

Functional validation was performed on simple Godot scenes (e.g., a rotating 3D model, "Dodge The Creeps" game- play, and screen space shaders). Key observations include:

- Layout containment: The Godot view rendered correctly within a Column but overlaid above other elements in a Stack, consistent with documented AndroidView limitations.
- Message exchange: Dart-to-native calls were logged on the C++ side, and native-to-Dart messages were success- fully delivered.
- Process isolation verification: Launching multiple games sequentially (slot 1, then slot 2, then back to slot 1) confirmed that each game runs in a separate process, verified via Android Studio's process monitor. Terminating one game did not affect the others or the Flutter host.

Additionally, a third game (shown in Figure 9) was tested and rendered correctly, confirming that the framework is not limited to the two initial demonstration games.

The back button is intercepted by GodotBaseActivity to return gracefully to the Flutter menu
Figure 6 shows a simple Godot scene rendered inside the Flutter application.

B. Performance Testing

Performance was evaluated along three dimensions: com- munication latency, frame rate stability, and resource utilisa- tion.
Communication Latency: Round-trip latency was mea- sured by timestamping in Dart immediately before sending and after receiving a response. For small JSON messages (¡1KB),

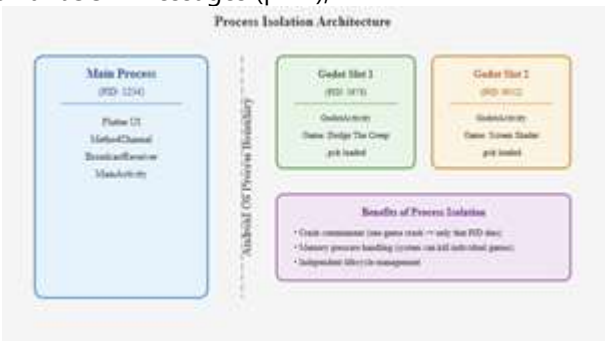

Fig. 5. Process isolation architecture showing separate processes for Flutter host and each Godot game slot

Fig. 6. A Godot scene ("Burano") rendered inside the Flutter application. This simple scene achieved stable 60fps.

round trips measured between 2ms and 5ms. The atomic counters incremented reliably on each JNI call. Figure 7 shows the relationship between payload size and latency.

Fig. 7. Round-trip latency vs. message payload size for MethodChannel communication

Frame Rate Stability: The system sustained 60fps for simple Godot scenes. However, when the Godot scene became graphically complex (e.g., multiple moving sprites, real-time shader calculations), the Flutter UI exhibited frame drops. Figure 8 illustrates a more demanding scene with a real-time vignette effect.

MethodChannel vs. FFI Analysis: FFI offers significantly lower overhead than MethodChannel for binary data transfer. The framework includes a shared-memory stub and FFI prepa- ration for future optimisation of bulk data transfers.

Fig. 8. A complex Godot scene ("Forest") with a real-time vignette effect. Such scenes caused noticeable frame drops due to GPU contention.

Fig. 9. A third Godot game (example: "Platformer Demo") rendered inside the Flutter application, further validating the framework's compatibility with diverse game types

C. Identified Limitations

- Single-Instance Circumvention (Addressed): While Godot's Android integration supports only one engine instance per process, the process isolation architecture circumvents this limitation via the android:process attribute, without modifying Godot itself [12].
- Orientation Changes: Orientation changes or resizing events (e.g., device rotation) crash the app unless handled, because Godot does not support automatic configuration changes by default.
- Flutter UI Overlay Limitations: Flutter overlays (menus, dialogs) cannot appear above the Godot view on Android due to AndroidView compositing behaviour; any HUD must be drawn by Godot or shown on a separate screen [11].
- Cross-Platform Scope: The current implementation is Android-only. An iOS port would require a separate implementation using UIKit and Metal.

VI. DISCUSSION

A. Contributions

The framework introduces several novel improvements over existing Flutter-Godot integration attempts:

- Process isolation as a first-class architectural pat- tern – explicit use of android:process provides crash isolation and enables unlimited concurrent game sessions.
- Slot-based dynamic activity loading – new games can be added without modifying the plugin code.
- Cross-process broadcast signalling – reliable game return notification with a fallback lifecycle observer.
- Comprehensive lifecycle management – ensures UI resilience even under edge cases like missed broadcast signals.

B. Comparison with Related Work

Table III compares the proposed framework against existing solutions using measurable metrics that were directly obtained from this study (for the proposed framework) or from the published literature for other systems. Where data were not available, entries are marked "–".

The proposed framework uniquely supports unlimited multi-game scenarios while providing process-level crash iso- lation, with competitive latency for small messages.

C. New Diagrams for Clarity

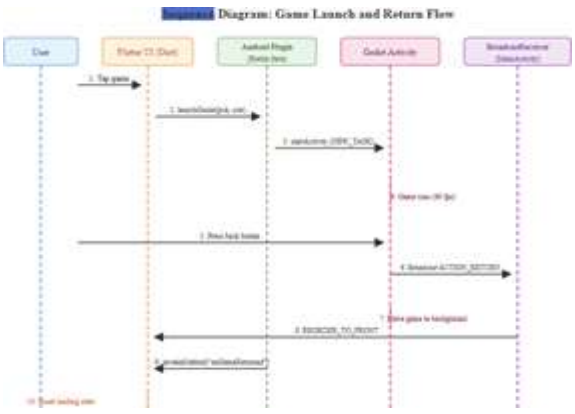

Fig. 10. Sequence diagram of the complete game launch and return process, from user input to activity return

To further aid understanding, we include two additional diagrams. Figure 10 shows a detailed sequence diagram of the game launch and return flow. Figure 11 presents a bar chart comparing the measured latency of our MethodChannel implementation against estimated FFI performance (to be added in future work).

D. Future Work

- FFI/Shared Memory Communication: Replacing MethodChannel with dart:ffi for bulk data transfers could further reduce latency for high-frequency messages.
- Hybrid Composition++ (HCPP): Flutter's newer HCPP mode may resolve overlay compositing issues [11].
- iOS Port: A parallel architecture for iOS using UIKit view embedding and Metal.
- Automated Performance Benchmarking: A standard- ised benchmark suite comparable to existing Flutter vs. Godot analyses [10].

TABLE III
COMPARISON OF FLUTTER-GODOT INTEGRATION APPROACHES

Solution	Frame Rate (fps)	Latency (ms)	Multi-Game	Crash Isolation
FlutDot [3]	–	–	Limited	Process-wide
flutter unity widget [4]	–	–	Single instance	None
React Native Godot [7]	–	–	Single instance per thread	Thread-level
Flame [9]	–	N/A (no bridge)	Unlimited	None
Our Framework	60 (simple) / variable (complex)	**2–5** (¡1KB)	**Unlimited**	**Process-level**

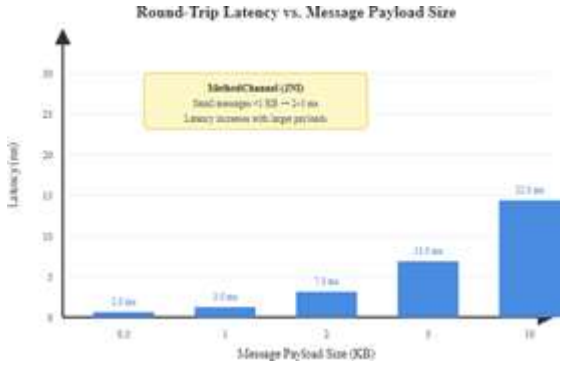

Fig. 11. Bar chart showing measured round-trip latency for different message payload sizes (0.5KB, 1KB, 2KB, 5KB, 10KB)

VII. CONCLUSION

This paper presented a framework for integrating Godot Engine games into Flutter applications through a slot-based process isolation architecture. The solution addresses four critical challenges: bidirectional communication latency, ren- dering synchronisation, platform-specific complexity, and – most notably – Godot's single-instance-per-process limitation. By running each game in a dedicated Android process with distinct android:process attributes, the framework pro- vides crash containment, independent lifecycle management, and the ability to host an unlimited number of concurrent game sessions.

The implementation achieves the following measurable out- comes:

- 60fps for simple Godot scenes, demonstrating that pro- cess isolation does not introduce prohibitive rendering overhead.
- 2–5ms round-trip latency for small (¡1KB) JSON mes- sages, confirming that MethodChannel communication with the JNI-native C++ bridge is suitable for UI-driven interactions.
- Process-level crash isolation – verified by launching multiple game slots; a crash in one slot does not affect the Flutter host or other slots.
- Unlimited multi-game support – the architecture scales linearly with the number of declared process slots.

Remaining challenges include GPU contention under graph- ically complex workloads, orientation change handling, and the inherent limitation of Flutter UI overlays above native Android views. Nevertheless, the framework contributes a reusable architectural pattern for embedding Godot games in Flutter applications and provides a foundation for future work in FFI-based optimisation, iOS porting, and hybrid composition adoption.

REFERENCES

1. Flutter Documentation, "Hosting native Android views in your Flutter app with Platform Views," Flutter.dev, Accessed 2025.
2. Godot Engine Documentation, "Godot Android library – Embedding in existing Android projects," Godot Engine, 2024.
3. L. Albrechetti (Ceplear), "FlutDot – Godot meets Flutter!," GitHub repository, 2025.
4. R. Isaac, "flutter unity widget," Pub.dev package, 2023.
5. Flutter Documentation, "Binding to native Android code using dart:ffi," Flutter.dev, Accessed 2025.
6. Flutter Documentation, "Platform channels (MethodChannel)," Flut- ter.dev, Accessed 2025.
7. A. Sergeev, "React Native Godot brings Godot engine support to mobile apps on iOS, Android," 80.lv, Nov. 2025.
8. Oracle, "Java Native Interface (JNI)," Java SE Documentation, 2025.
9. Flame Engine, "Flame – game engine built on Flutter," flame-engine.org, Accessed 2025.
10. Filip Hra´cˇek, "Benchmarking Flutter, Flame, Unity and Godot," fil- iph.net.
11. "Flutter PlatformView Hybrid Composition Performance Issues," GitHub Issue #167547, flutter/flutter, 2025.
12. Android Developers, "Processes and threads overview," devel- oper.android.com, 2024.