Technical documentation

LambdaFlow

**Document type**: Architecture

**Version**: 1.0.0

**Date**: 30/05/2025

Table of Contents

**Content**

[Table of Contents 2](#_Toc199499290)

[Version History 4](#_Toc199499291)

[1. Purpose 5](#_Toc199499292)

[2. Scope 5](#_Toc199499293)

[3. Content 5](#_Toc199499294)

[3.1. Introduction 5](#_Toc199499295)

[3.2. Goals 5](#_Toc199499296)

[4. Architecture 6](#_Toc199499297)

[4.1. Basic components 6](#_Toc199499298)

[4.2. Class Reference 6](#_Toc199499299)

[4.2.1. Class IPCBridge 6](#_Toc199499300)

[4.2.1.1. Members 8](#_Toc199499301)

[4.2.1.2. Workflow 8](#_Toc199499302)

[4.2.1.3. Design Justification 9](#_Toc199499303)

[4.2.2. Class WebViewHost 9](#_Toc199499304)

[4.2.2.1. Members **¡Error! Marcador no definido.**](#_Toc199499305)

[4.2.2.2. Workflow 9](#_Toc199499306)

[4.2.2.3. Design Justification 10](#_Toc199499307)

[4.3. Message Flow Diagrams 10](#_Toc199499308)

[4.3.1. Frontend to Backend 10](#_Toc199499309)

[4.3.2. Backend to Frontend 10](#_Toc199499310)

[4.3.3. Application Startup 11](#_Toc199499311)

[5. Glossary 11](#_Toc199499312)

[6. References 11](#_Toc199499313)

[7. Appendices 11](#_Toc199499314)

[7.1. Appendix A 11](#_Toc199499315)

[7.2. Appendix B 11](#_Toc199499316)

[7.3. Appendix C 11](#_Toc199499317)

[7.4. Appendix D 11](#_Toc199499318)

Version History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Author** | **Version** | **Date** | **Description** | **Approved by** |
| David L.M. | 1.0.0 | 30/05/2025 | Initial description of LambdaFlow functions and how is implemented. | David L.M. |

1. Purpose

This document provides a comprehensive overview and detailed reference for LambdaFlow. Its intended use is to guide new contributors and third‑party developers in understanding the system’s architecture, core classes, message flows, configuration, and design rationale.

1. Scope

This documentation applies to all software engineers, contributors, and integrators using or extending LambdaFlow. It covers the framework’s internal classes (IPCBridge, WebViewHost), the host application example, platform-specific WebView integrations (Windows, macOS, Linux), and the configuration JSON format. It does not cover organizational processes or project management procedures.

1. Content
   1. Introduction

LambdaFlow is a lightweight C++ library that enables seamless integration between a WebView-based HTML/CSS/JS frontend and an arbitrary backend (e.g., C#, Python, Java) via high-performance binary IPC pipes. It abstracts process creation, WebView hosting, and thread-safe message exchange into an easy-to-use API.

* 1. Goals
* **Simplicity**: minimal setup, clear APIs.
* **Performance**: binary IPC over pipes, worker threads.
* **Flexibility**: support any backend language/executable.
* **Cross-platform**: Windows (WebView2), macOS (WebKit), Linux (WebKit2GTK).

1. Architecture
   1. Basic components

The framework comprises three main components:

1. **IPCBridge**: manages IPC channels (reader & writer threads) between host and backend.
2. **WebViewHost**: initializes and controls the WebView window, injects JS-to-C++ hooks.
3. **Host Application** (main.cpp): orchestrates startup, configuration loading, launches backend process, ties IPCBridge and WebViewHost.
   1. Class Reference
      1. Class IPCBridge

Responsible for setting up two uni-directional pipes one for frontend → backend and another for backend → frontend and launching two dedicated threads, one for each uni-directional pipe.

* + - 1. Worker Threads

Before diving into members and workflows, two dedicated threads handle I/O:

* **Writer Thread** (\_writerThread):
  + **Role:** Delivers messages from the host application to the backend process (outbound direction).
  + **Lifecycle:** Starts during IPCBridge construction and continues running while \_running == true or until all queued messages have been sent. When stop() is called, it drains remaining messages, then exits cleanly.
  + **Behavior:**
    - Locks \_outMutex and waits on \_outCv if \_outQueue is empty.
    - When notified and \_running is still true, pops the next Message from \_outQueue.
    - Writes a 4‑byte length prefix, followed by raw message bytes to the backend stdin pipe (pipeToBackend\_[1]).
    - Repeats until shutdown.
* **Reader Thread** (\_readerThread):
  + **Role:** Receives messages from the backend process and forwards them to the host application.
  + **Lifecycle:** Starts during IPCBridge construction and runs until stop() signals shutdown by closing the read pipe or setting \_running = false. It exits immediately on EOF or read error.
  + **Behavior:**
    - Blocks on a read from the backend stdout pipe.
    - Reads a 4-byte length header, then reads exactly that many bytes of message data.
    - Wraps the raw bytes into a Message object.
    - Posts the Message to the main / UI thread by invoking the registered backend callback.
    - Loops until shutdown.
      1. Methods (Public API)

|  |  |
| --- | --- |
| Method Signature | Description |
| IPCBridge(const std::string& backendExePath) | **Constructor**. Initializes internal state and stores the path to the backend executable. |
| ~IPCBridge() | **Destructor**. Ensures stop() is called and resources (threads, pipes, child process…) are cleaned up. |
| bool start() | **Start communication**. Creates pipes, launches the backend process, and spawns reader/writer threads. Returns **True** on success pr **False** on failure. |
| void sendToBackend(const std::string& message) | **Enqueue message**. Pushes a binary message (byte buffer) onto the send queue and notifies the writer thread. |
| void stop() | **Shutdown**. Atomically signals threads to stop, closes the write pipe, notifies the writer, joins both threads, waits on the backend process and closes remaining descriptors |

* + - 1. Members

|  |  |
| --- | --- |
| Member | Purpose |
| std::thread \_renderThread | Executes the Reader Thread for inbound messages. |
| std::thread \_writerThread | Executes the Writer thread loop for outbound messages. |
| int pipeToBackend\_[2] | Unix/Win handles for writing into backend’s stdin. |
| int pipeFromBackend\_[2] | Handles for reading from backend’s stdout. |
| std::mutex \_outMutex | Guards the Writer Thread when new data is available. |
| std::condition\_variable \_outCv | Notifies the Writer Thread when new data is available. |
| std::queue<Message> \_outQueue | Buffer of pending outbound messages (host to backend) |
| backendCallback \_backendCallback | User-supplied function called on each inbound message. |
| std::atomic<bool> \_running | Flag to control the lifetime of the I/O loops. |
| ProcessHandle \_backendProcess | OS-specific handle or PID for the child backend process. |

* + - 1. Workflow

1. **Construction:** 
   1. Creates two anonymous pipes.
   2. Launches the backend process with its stdin/stdout redirected to the pipe ends.
2. **Enqueuing:**
   1. Locks \_outMutex, pushes message into \_outQueue and notifies \_outCv.
3. **Outbound I/O (WriterThread):**
   1. Waits on \_outCv.
   2. When notifies and \_running == true, pops each message and writes a 4-byte length + raw bytes into pipeToBackend\_[1].
4. **Inbound I/O (ReaderThread):**
   1. Blocks on pipeFromBackend\_[0].
   2. Reads the length header, then reads the specified number of bytes.
   3. Constructs a Message and invokes \_backendCallback(message) on the main / UI thread via a dispatch mechanism
5. **Shutdown:**
   1. Sets \_running = false, closes pipeToBackend\_[1] to unlock the writer.
   2. Notifies \_outCv to wake the writer thread.
   3. Joins both threads, closes remaining descriptors and waits for the backend process to exit.
      * 1. Design Justification

* **Dedicated Threads** ensure that read/write operations never block the main/UI thread, providing smooth responsiveness.
* **Length‑Prefixed Binary** avoids the complexity of parsing delimiters and maximizes raw throughput.
* **Callback Model** delivers inbound messages asynchronously, allowing event‑driven architectures on both frontend and backend.
* **Atomic Flag** (\_running) cleanly signals both loops to exit without race conditions.
  + 1. Class WebViewHost

Abstracts platform-specific WebView creation and JS bridge.

* + - 1. Methods (Public API)

|  |  |
| --- | --- |
| Method Signature | Description |
| WebViewHost(int w, int h, const std::string& title, const std::string& htmlFile) | **Constructor**. Creates a WebView instance of given dimensions, sets window title, loads and embeds the HTML file into the view. |
| void setJSCallback(const std::string& name, JSCallback cb) | **Register JS callback**. Binds a JavaScript function window.name(payload) to invoke the provided C++ JSCallback when called from the page. |
| void run() | **Run loop**. Enters the WebView event loop with webview\_run, blocking until the window is closed. |
| ~WebViewHost() | **Destructor**. Implicitly or explicitly destroys the WebView instance and frees associated resources. |

* + - 1. Members

|  |  |
| --- | --- |
| Member | Purpose |
| webview\_t \_w | Opaque handle representing the underlying WebView instance (cross-platform C API). |
| std::thread \_writerThread | Executes the Writer thread loop for outbound messages. |
| (in BindData) JSCallback userCb | Holds the user-provided callback to invoke when the corresponding JS function is called. |
| (in BindData) webview\_t w | Copies the \_w handle so that the JS-binding lambda call back into the correct WebView context. |

* + - 1. Workflow

1. **Construction**
   1. Calls webview\_create to instantiate the view.
   2. Sets window title and size.
   3. Reads the entire HTML file into a string and load it.
2. **JavaScript Binding**
   1. Allocates a heap-allocated struct containing C++ JSCallback and WebView handle.
   2. Calls webview\_bind, passing a lambda that:
      1. Casts userData back to BindData\*
      2. Invokes userCb(std::string(req)) with the raw request payload.
      3. Calls webview\_return to signal completion back to the JS caller.
3. **Event Loop**
   1. Starts the native GUI loop and blocks until the user closes the window.
4. **Destruction**
   1. When WebViewHost is destroyed the underlying webview\_t instance is torn down via webview\_destroy, freeing OS resources.
      * 1. Design Justification

* **Separation of Concerns**: UI hosting and IPC logic are decoupled—WebViewHost focuses solely on window management and JS bindings.
* **Event-Driven Model**: webview\_bind enables push-based JS→C++ communication, avoiding inefficient JS polling.
* **Cross-Platform Consistency**: The same C++ wrapper works across WebView2 (Windows), WebKit2GTK (Linux), and WebKit (macOS).
* **Minimal Overhead**: Small memory footprint and fast startup compared to full browser engines or Node.js runtimes.
  1. Message Flow Diagrams
     1. Frontend to Backend

1. Frontend->>WebViewHost: window.env.send(json)
2. WebViewHost->>IPCBridge: sendToBackend(json)
3. IPCBridge->WriterThread: enqueue message
4. WriterThread->>Backend process pipe: write(len+data)
   * 1. Backend to Frontend
5. Backend process->>IPCBridge pipe: write(len+data)
6. IPCBridge ReaderThread->>IPCBridge: read + decode
7. ReaderThread->>JS Thread: invoke backendCallback\_(data)
8. JS Thread->>WebViewHost: Dispatch JS callback
9. WebView->>Frontend: window.onBackendMessage(data)
   * 1. Application Startup
10. Start --> Load config.json
11. Load config --> IPCBridge.start()
12. IPCBridge.start --> launch backend process
13. launch backend --> init WebViewHost with config
14. WebViewHost.run --> enter UI event loop
15. Glossary

|  |  |
| --- | --- |
| Term | Definition |
| (Term 1) | (Definition 1) |
| (Term 2) | (Definition 2) |
| (Term 3) | (Definition 3) |

1. References

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Title | Type | Location |
| (ISO 27001) | (Security Framework) | [Internal / External]  [Document / Guide / Standard] | (URL / PATH) |

1. Appendices
   1. Appendix A
   2. Appendix B
   3. Appendix C
   4. Appendix D