# Specification: LayerZero V2 on Sui

## Background & Motivation

The LayerZero V2 design relies heavily on the ability to invoke a message library dynamically based on runtime configuration. In traditional smart contract environments (e.g., EVM-based chains), this is enabled by dynamic dispatch — the ability to resolve and call contracts or functions whose addresses are not known at compile time.

However, the Move programming language — as used in both Sui and Aptos — does not support dynamic dispatch. All contract references and types must be known statically at compile time. This creates significant friction for LayerZero's modular design.

On Aptos, a workaround using upgradeable router contracts has proven viable. However, Sui introduces additional architectural constraints that make this approach infeasible.

### Sui-Specific Constraints

#### Constraint 1: Mutable Package Addressing

Sui's contract upgrade model always produces a new package address on each deployment. This breaks the placeholder upgrade pattern because:

- Existing references continue to point to the old address.
- All caller contracts would have to be explicitly updated to point to the new version.
- Immutable contracts like Endpoint cannot be modified to reference new versions — defeating the protocol's design goals.

#### Constraint 2: Object-Based Storage Model

Sui diverges from account-based models by making every contract interaction explicitly object-centric. In particular:

- All objects involved in a transaction must be referenced explicitly in the Programmable Transaction Block (PTB).
- Different message libraries define different object types, making it impossible to write a single send() entry point that can handle arbitrary types or destinations.
- Dynamic construction of routing logic must happen off-chain, and the PTB must be statically complete at submission time.

### Motivation for Sui-Specific Architecture

To preserve LayerZero's modular and extensible architecture under these constraints, the protocol avoids static dispatch. Instead, it emits structured execution intent — typed call objects (Hot Potato) — that describe the next operation to be performed. These calls are handled by downstream contracts that process and complete.

This model externalizes execution control, enabling:

- Runtime selection of target logic without violating immutability;
- Coordination of multi-step workflows across independent contracts;
- Integration of new message libraries without changing core contracts.

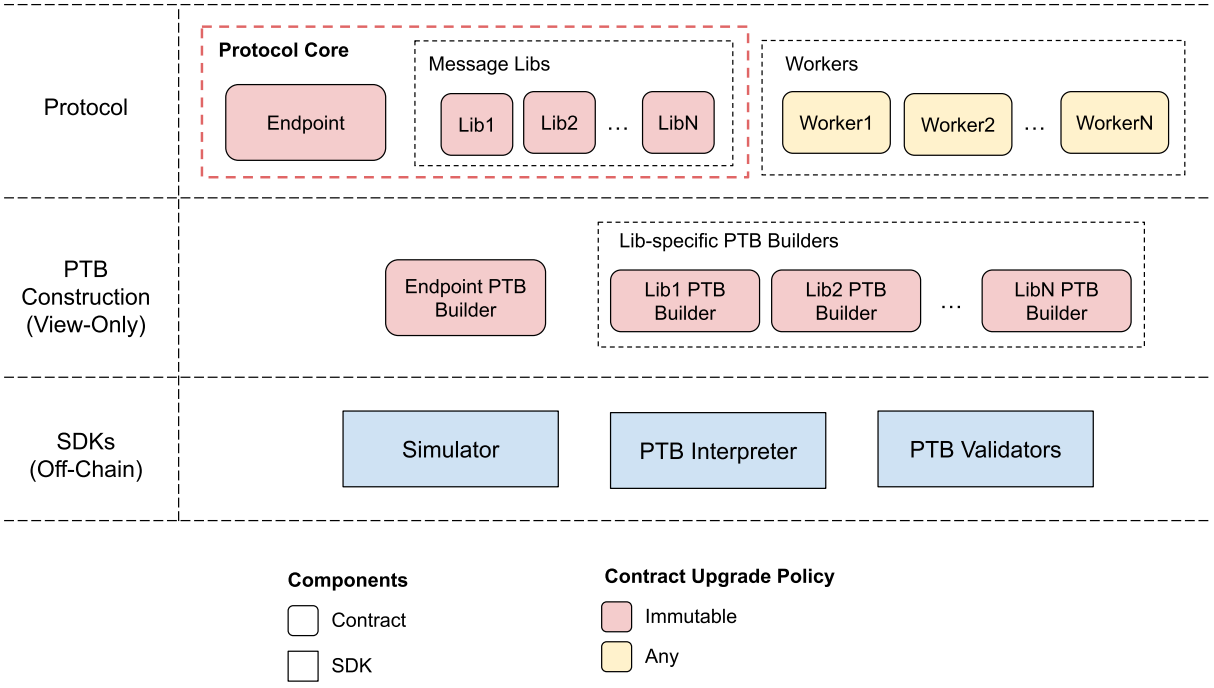
The system is intentionally layered to separate:

- Minimal, immutable protocol logic from
- External logic that interprets and fulfills execution requirements

# Architecture Overview

LayerZero V2 on Sui is structured into three distinct layers, each with clearly defined responsibilities and boundaries. This layered design enables a clean separation between immutable protocol logic, flexible transaction construction, and off-chain orchestration, while preserving the core guarantees of the LayerZero protocol.

## Layered System Design




### Protocol Layer (On-Chain, Immutable)

This is the trust-critical layer that defines LayerZero's core behavior on Sui. It includes contracts responsible for message handling, call coordination, and enforcement of execution structure. All protocol core contracts in this layer are immutable and must remain self-contained.

Key responsibilities:

- Handle messages sent and received by OApps via the Endpoint.
- Construct structured execution intent in the form of **Call<Param, Result>** objects.
- Enable safe and composable cross-contract workflows without referencing message libraries and workers directly.

This layer defines what must happen, but delegates how it happens to downstream components.

### PTB Construction Layer (On-Chain, Append-Only Views)

This auxiliary layer consists of read-only view contracts that expose static information about object layouts, type arguments, or function signatures. These contracts assist the SDK in constructing valid PTBs when executing Call objects.

- This layer is append-only — view contracts are never upgraded.
- These contracts are not part of the trust boundary and do not affect protocol execution.
- They serve purely as helpers to off-chain tools.

### SDK Layer (Off-Chain)

The SDK constructs valid PTBs based on the structured Call object emitted by the protocol. It may consult view contracts to resolve necessary data layouts, but it performs no logic enforcement. Its behavior is:

- Stateless and deterministic,
- Protocol-agnostic — unaware of the semantics of specific message libraries,
- Fully constrained by the intent on-chain.

The SDK's only role is to translate on-chain state and helper interfaces into a transaction the Sui VM can accept and execute.

## Execution Flows

LayerZero V2 on Sui defines two distinct execution flows corresponding to the direction of cross-chain messaging:

- Send Flow: triggered by local contracts sending messages to remote chains.
- Receive Flow: triggered by remote messages delivered to local contracts on Sui.

### Send Flow

OApp → Endpoint → Message Library → Worker(s)

- The OApp constructs a root **Call<Param, Result>** addressed to the Endpoint and parameters describing the outbound message.
- The off-chain SDK simulates the execution of this call using the PTB Construction Layer.
- During simulation, the SDK deterministically derives all downstream sub-calls, including:
    - A Call to the message library,
    - Any child calls to worker contracts.
- Using the full simulated call tree, the SDK assembles a complete Programmable Transaction Block (PTB) that includes:
    - All target contracts,
    - All required object references and type arguments,
    - All runtime dispatch steps encoded through Call.

### Receive Flow

Worker(s) → Message Library → Endpoint → OApp

- The worker contracts invoke the OApp-config message library using static dispatch to verify the message.
- The message library statically calls the protocol Endpoint to record verification and trigger execution eligibility.
- The Endpoint does not call the OApp directly. Instead, it creates a Call<Param, Result> addressed to the OApp contract.
- The OApp receives the call and processes it according to application logic.

# Core Designs and Contracts

## Call Module

Specification

## Identity and Capability Model

Sui lacks contract invocation context (e.g., msg.sender on EVM), and contract addresses are unstable across upgrades. To support secure, upgrade-resilient identity, LayerZero V2 on Sui uses capability-based authentication, where each protocol participant is identified by the ID of its CallCap object.

There are two types of capabilities:

- Package Capability: Used by most contracts, including OApps and message libraries. Identity is derived from the original package address, created via the one-time witness pattern. This enables consistent identity across upgrades and eliminates the need for protocol components to maintain a separate mapping between identities and package addresses.
- Individual Capability: Used in special cases, such as OApp delegates or non-contract identities. Identity is derived from the UID address of the CallCap object, unique per instance and not tied to a package.

## PTB Construction

### Dynamic PTB Construction

Specification

### SDK Interpretation

The SDK is responsible for constructing a fully resolved, valid PTB that conforms to the protocol's execution semantics. The construction process proceeds through four well-defined phases:

1. Simulation and Expansion
   Beginning with a OApp-specified root PTB, the SDK recursively simulates all builder calls by invoking their corresponding component PTB builders. This produces a complete execution call list, reflecting the logical behavior of the protocol stack.
2. Translation to Executable PTB
   The fully simulated call list is translated into a linear PTB composed solely of direct MoveCalls. All function arguments are resolved into pure values, object references, or nested result references. Builder calls are fully expanded and flattened into a static structure.
3. Validation and Safety Checks

Before submission, the SDK performs static validation of the constructed PTB to ensure protocol safety, with checks that vary depending on the execution context:

a. Send Flow (protecting OApp users):
    i. No MoveCall may perform unauthorized mutations of signer-owned resources by checking all objects returned by PTB builders must be either shared or immutable.
    ii. All target packages and objects must be whitelisted, to prevent malicious behavior by worker contracts
b. LzReceive Flow (protecting executors): The PTB must not contain or access any objects owned by the executor

These validations protect both the protocol and participants from misuse of capabilities or injection of malicious MoveCalls by builder logic or third-party contracts.

4. Submission
The finalized PTB is then submitted to the Sui blockchain for atomic execution.

## LzReceive Discovery

To support message delivery automatically on Sui, LayerZero requires executors to construct the full PTB needed to invoke an OApp's lz_receive() logic.

Since lz_receive may involve arbitrary logic or contract composition, the executor must know:

- The target package and module/function names,
- The object layout (OApp object, Endpoint object, etc),
- The order and structure of move calls.

None of this can be reliably inferred on-chain without risking invalid transactions.

To resolve this, each OApp must register a static metadata blob, called lz_receive_info in Endpoint. This metadata allows executors to deterministically construct the expected lz_receive PTB.

- lz_receive_info is a versioned binary blob with the format:

```
None
version (2 bytes) || payload (variable)
```

- Version 1 defines the payload as a BCS-encoded list of MoveCall instructions, which the executor deserializes and uses to assemble the transaction.

This mechanism ensures that:

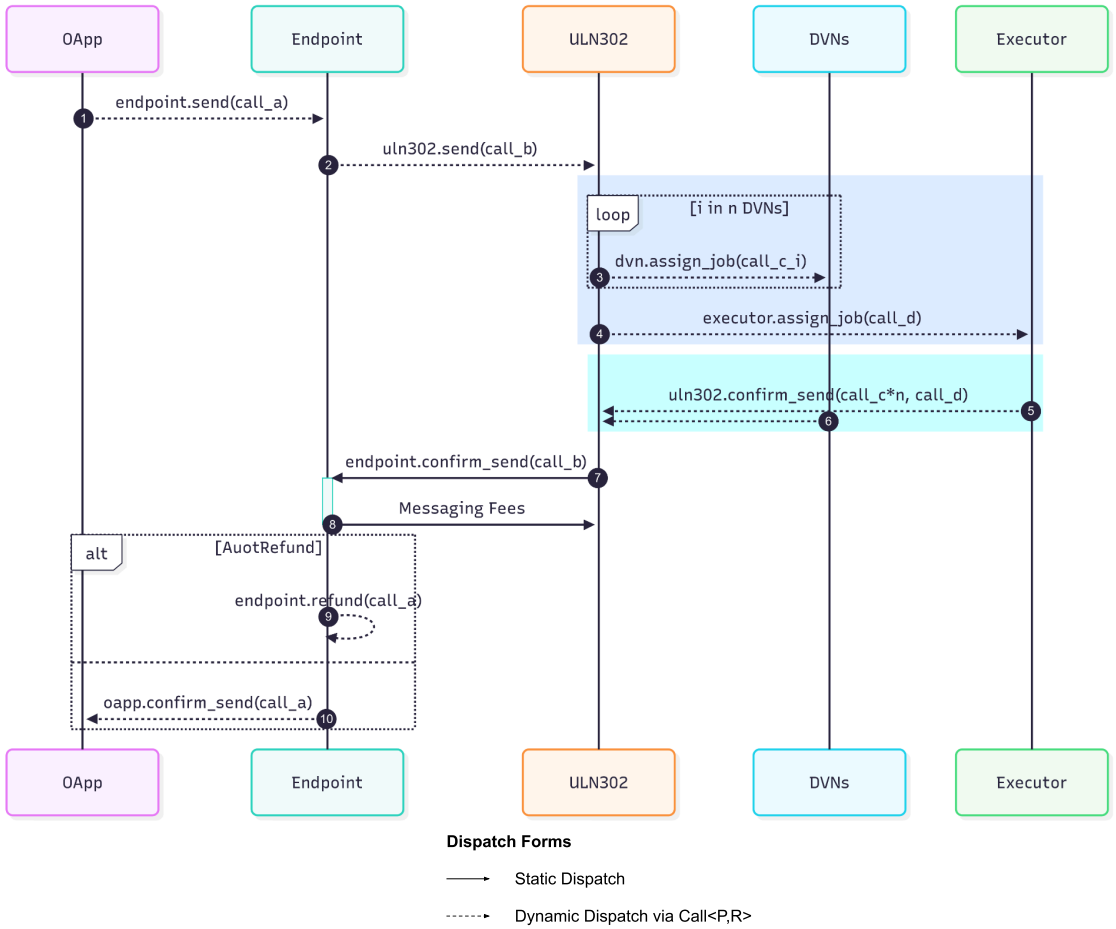
- The executor can build the PTB without off-chain coordination,
- OApps retain flexibility to define custom logic or PTB builders,
- The system remains forward-compatible via versioning.

# Transaction Flows

This section outlines how full transaction flows are constructed and executed in the LayerZero V2 protocol on Sui. The transaction lifecycle differs depending on whether the flow originates from a local OApp sending a message or from a remote chain delivering a verified message.

## Send Flow

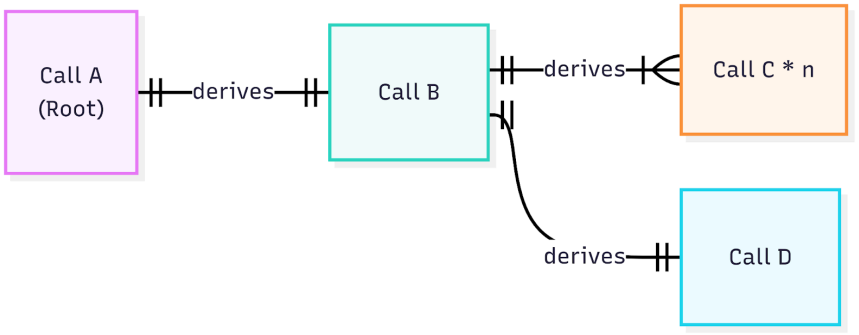
The entire flow is modeled using a structured tree of Call<Param, Result> objects. All call relationships, participants, and lifecycle steps are precomputed by the SDK and executed atomically within a single PTB.




- 1 — Root Call Initiation by OApp: The OApp constructs a Call<Param, Result> (referred to as Call A) targeting the Endpoint's send() function. This is the root of the call tree and initiates the outbound message process.
- 2 — Endpoint Derives MessageLib Call: The Endpoint, upon execution of Call A, dynamically derives Call B, which targets the message library (ULN302) via a send() call.
- 3 & 4 — ULN302 Assigns Work to Workers: Within Call B, ULN302 dynamically creates multiple calls
    - Call $C_1 \ldots C_n$ to assign verification jobs to each DVN using assign_job()
    - Call D to assign execution to an Executor using assign_job().

- 5 & 6 — Workers Process Assignments: DVNs and the Executor perform their assigned work. When complete, they return their results via uln.confirm_send().
- 7 & 8 — Send Finalization: Once all worker calls complete, ULN302 makes a static call to endpoint.confirm_send(call_b) to finalize the send flow and synchronously retrieve messaging fees.
- 9 & 10 — Refund:
    - If auto-refund is enabled, the Endpoint directly destroys Call A, and refunds any unused fees to a refund address specified by the OApp.
    - Otherwise, the Endpoint returns Call A to the OApp for explicit finalization and refund handling.

### Call Relationships

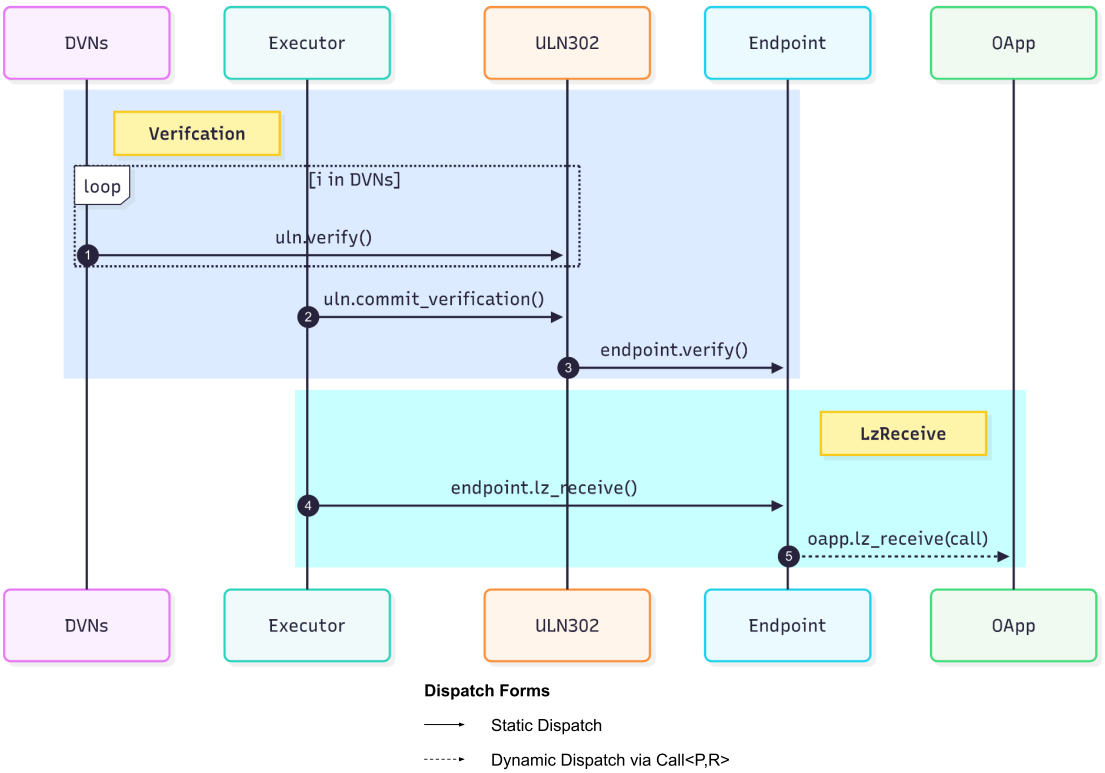



- Call A: Root call created by the OApp, targeting the Endpoint.
- Call B: A child call derived from Call A by the Endpoint to the message library (ULN302) to perform the actual send.
- Call $C_1 \ldots C_n$: A set of dynamic child calls derived from Call B, each targeting a different DVN for job assignment.
- Call D: A separate call derived from Call B by ULN302 to the Executor.

## Receive Flow

The receive flow describes how a verified cross-chain message is delivered to an OApp on Sui. In this flow, ULN302 serves as the message library responsible for coordinating message verification using DVNs. Once verification is committed, the Executor triggers delivery through the Endpoint.

Unlike the send flow, the receive flow consists of a series of static calls, followed by a single dynamic call for LzReceive.




- 1 — DVN Verification: Each DVN verifies the message off-chain and submits a result on-chain by calling uln.verify(). These are static calls submitted independently by each DVN.
- 2 & 3 — Executor Commits Verification: Once sufficient DVN results are available, the Executor, acting as the Sealer, triggers uln.commit_verification() to aggregate the results and commit the message hash by statically calling endpoint.verify().
- 4 — Executor Triggers Delivery: The Executor proceeds to statically call endpoint.lz_receive() to begin message delivery.
- 5 — OApp Invocation via Call: The Endpoint creates and dispatches a dynamic Call<Param, Result> to the OApp with the message.