ILifeCycleStateMachine.cs
File

ILifeCycleStateRepository.cs
File

LifeCycleInstanceFlag.cs
File

LifeCycleStateFlag.cs
File

LifeCycleTransitionFlag.cs
File

LifeCycleTransitionLogFlag.cs
File

lc_state.sql
File

State machine Recap: Let me reload all the files, go through them and until i say go, dont' process anything.. I want you to througly read everything and then prepare me a detailed Read me file. The file should be containing prmopts and inputs in such a way that If I upload the file again to you or any other LLM platform, it should understand what we are trying to achieve.. Pay attention to how the string interpoliation is set up for queries and query fields, how we are using haley libararies etc.. Let me start by uploading, until i say 'start' please wait..

Understood. I have **loaded all files into context** and will **wait silently**.

⚠️ **I will not process, summarize, or generate anything until you explicitly say:** `start`

You can continue uploading more files.

Wait..

Sure — I’m waiting.

QRY_ACK_LOG.cs
File

QRY_CATEGORY.cs
File

QRY_DEF_VERSION .cs
File

QRY_DEFINITION.cs
File

QRY_EVENT.cs
File

QRY_INSTANCE.cs
File

QRY_MAINTENANCE.cs
File

QRY_STATE.cs
File

QRY_TRANSITION.cs
File

QRY_TRANSITION_LOG.cs
File

wait

All newly-uploaded files have been fully loaded and indexed.

I am **still waiting** — I will not process anything until you explicitly say **“start”**.

Take your time.

QueryFields.cs
File

TransitionEventArgs.cs
File

LifeCycleDefinition.cs
File

LifeCycleDefinitionVersion.cs
File

LifeCycleEvent.cs
File

LifeCycleInstance.cs
File

LifeCycleState.cs
File

LifeCycleTransition.cs
File

LifeCycleTransitionLog.cs
File

Wait

All newly uploaded files are now fully loaded and indexed.

I’m still in **WAIT MODE**.

I will not generate **any output, summary, interpretation, or documentation** until you explicitly say:

```
start
```

Waiting…

DefinitionJson.cs
File

DefinitionLoadResult.cs
File

DefinitionSpec.cs
File

StateSpec.cs
File

TransitionSpec.cs
File

LifeCycleInitializer.cs
File

LCSM.State.cs
File

LifeCycleStateMachine.cs
File

Wait

All newly uploaded files are fully loaded.

I remain in **WAIT MODE**.

I will not generate anything until you explicitly say:

```
start
```

Waiting…

LCSM.Core.cs
File

LCSM.Helpers.cs
File

LCSM.Importer.cs
File

SM.Maria.State.cs
File

SM.Maria.Transition.cs
File

SM.Maria.TransitionLog.cs
File

Wait

All newly uploaded files have been successfully loaded into context.

Still in **WAIT MODE**.

I will not begin processing until you explicitly say:

```
start
```

Waiting…

LifeCycleStateMariaDB.cs
File

SM.Maria.Ack.cs
File

SM.Maria.Category.cs
File

SM.Maria.Definition.cs
File

SM.Maria.Event.cs
File

SM.Maria.Instance.cs
File

SM.Maria.Maintenance.cs
File

Well.. As of now, thats it.. Now, start..

Thought for 51s

Here we go. I'll treat this as a README you can drop into your repo and also re-feed to any LLM to continue work on the same design.

Haley Lifecycle State Machine

A small, database-backed **lifecycle state machine** built on top of the Haley libraries.
It manages **state definitions, events, transitions, runtime instances, transition logs, and ACK logs** using MariaDB and a generic `IAdapterGateway`.

This README is written so that **another human or LLM** can understand:

- What this module does
- How its pieces fit together

Database schema (MariaDB)

definition, def_version, state, events, transition, instance, transition_log, ack_log, category here, you add lc_state at the end, Dont' do that.. i'm not able to properly copy paste it.. Can you please recreate the entire readme, so that i can just copy paste it in Ms Word

Got it. I'll rewrite the README as a clean, copy-paste-friendly document with no stray script names or extra tokens hanging off bullet points.

You can drop this straight into Word.

Haley Lifecycle State Machine – Technical README

1. What This Module Is

This module implements a **generic lifecycle state machine** on top of MariaDB and the Haley libraries. It is designed to be reused by multiple domains (workflow engine, document system, submissions, etc.) and is responsible for:

- Defining **states**, **events**, and **transitions** for a lifecycle.
- Versioning lifecycle definitions.
- Tracking **instances** of a lifecycle for external entities.
- Auditing all transitions with **transition logs**.
- Recording **acknowledgement (ACK) status** for downstream notifications.

The module is environment-aware (dev/test/prod), version-aware, and uses **Haley's DB abstraction** so it can sit cleanly alongside other Haley components.

This README is written so that **another human or any LLM** can:

- Understand the design and responsibilities.
- Safely extend the codebase.
- Generate JSON definitions and integrate with the state machine without breaking conventions.

2. Responsibilities and Boundaries

2.1 What this module owns

- Overall lifecycle of an entity:
  - Which states exist.
  - Which state is initial/final/system/error.
  - Which transitions are allowed.
  - Which events drive transitions.
- Runtime state of an entity:
  - One lifecycle instance per external reference and definition version.
  - Transition history and audit logs.
- Import/export of lifecycle definition via JSON.
- ACK/outbox records for transition notifications.

2.2 What this module deliberately does not own

- Business payload data (documents, workflows, domain models).
- Micro-step progress inside a state (this belongs to “agents” or higher-level workflow engines).
- Scheduling of background jobs or message bus mechanics (the module only logs ACK metadata and exposes it).

3. High-Level Architecture

3.1 Layers

1. **Database (MariaDB)**

   Tables:

   - definition
   - def_version
   - state
   - events
   - transition
   - instance
   - transition_log

  - ack_log
  - category lc_state
2. **Domain Models (** `Haley.Models` **)**
  - `LifeCycleDefinition` , `LifeCycleDefinitionVersion`
  - `LifeCycleState` , `LifeCycleEvent` , `LifeCycleTransition`
  - `LifeCycleInstance` , `LifeCycleTransitionLog`
  - JSON spec models: `DefinitionJson` , `DefinitionSpec` , `StateSpec` , `TransitionSpec` , `DefinitionLoadResult`
  - `TransitionEventArgs` for events from the state machine. TransitionEventArgs
3. **Flags (** `Haley.Enums` **)**
  - `LifeCycleStateFlag` , `LifeCycleInstanceFlag` , `LifeCycleTransitionFlag` , `LifeCycleTransitionLogFlag` .
4. **Repository abstraction**
  - `ILifeCycleStateRepository` – complete abstraction over lifecycle DB operations. ILifeCycleStateRepository
5. **MariaDB implementation**
  - `LifeCycleStateMariaDB` – partial class using `IAdapterGateway` and strongly-typed SQL queries (`QRY_*`).
6. **State machine service**
  - `ILifeCycleStateMachine` – public API for consumers. ILifeCycleStateMachine
  - `LifeCycleStateMachine` – partial class (`Core` , `State` , `Importer` , `Helpers`) implementing logic.
7. **Initialization**
  - `LifeCycleInitializer` – creates the database from the embedded SQL script using `IAdapterGateway` . LifeCycleInitializer

4. Database Schema Overview

4.1 Table: `definition`

- Represents a lifecycle definition (e.g. “Prequalification Submission”).
- Columns (key ones):
  - `id` (PK, auto-increment)
  - `guid` (unique)

  - `display_name` (e.g. “Prequalification Submission”)
  - `name` = lowercase(display_name) (generated column)
  - `description`
  - `env` (0=Dev,1=Test,2=UAT/Prod etc.)
  - `created` lc_state
- Unique constraint: `(env, name)` – prevents duplicate definitions per environment.

4.2 Table: `def_version`

- Versioned “snapshot” for a definition.
- Columns:
  - `id` (PK)
  - `guid`
  - `parent` → FK to `definition.id`
  - `version` (int)
  - `data` (JSON of the original import)
  - `created` , `modified` lc_state
- `CHECK (json_valid(data))` ensures the stored JSON is valid.

4.3 Table: `state`

- All lifecycle states for a specific `def_version` .
- Columns:
  - `id`
  - `display_name` (e.g. “Created”, “Submitted”)
  - `name` = lowercase(display_name) (generated)
  - `flags` (bitmask mapped to `LifeCycleStateFlag`)
  - `category` (FK → `category.id`)
  - `def_version`
  - `created` lc_state
- Unique per `(def_version, name)` .

4.4 Table: `events`

- All lifecycle events for a `def_version` .
- Columns:
  - `id`
  - `display_name`

  - `name` = lowercase(display_name)
  - `def_version` lc_state

4.5 Table: `transition`

- Allowed transitions between states.
- Columns:
  - `id`
  - `from_state → state.id`
  - `to_state → state.id`
  - `event → events.id`
  - `def_version`
  - `flags` (bitmask mapped to `LifeCycleTransitionFlag`)
  - `guard_key` (string key for guards)
  - `created` lc_state
- Unique per `(def_version, from_state, to_state, event)`.

4.6 Table: `instance`

- Runtime lifecycle instances (e.g. for each workflow/submission).
- Columns:
  - `id` (PK)
  - `guid` (unique)
  - `def_version`
  - `current_state → state.id`
  - `last_event → events.id`
  - `external_ref` (GUID string of external entity)
  - `flags` (bitmask mapped to `LifeCycleInstanceFlag`)
  - `created`, `modified` lc_state
- Unique per `(def_version, external_ref)` – one lifecycle per entity & definition version.

4.7 Table: `transition_log`

- Audit trail of transitions.
- Columns:
  - `id`
  - `instance_id → instance.id`

  - `from_state`
  - `to_state`
  - `event`
  - `actor`
  - `flags` (bitmask mapped to `LifeCycleTransitionLogFlag`)
  - `metadata` (JSON text)
  - `created` lc_state

4.8 Table: `ack_log`

- Used for “blue tick” ACK tracking of transition notifications.
- Columns:
  - `id`
  - `transition_log` → `transition_log.id`
  - `ack_status` (1=Sent, 2=Acknowledged, 3=Failed)
  - `last_retry`
  - `retry_count`
  - `created`, `modified`
  - `message_id` (GUID string) lc_state

4.9 Table: `category`

- Groups states into user-defined categories.
- Columns:
  - `id`
  - `display_name`
  - `name` = lowercase(display_name) lc_state

5. Flags and Domain Models

5.1 Flags (all prefixed with `LifeCycle*`)

- `LifeCycleStateFlag` LifeCycleStateFlag
  - `IsInitial`
  - `IsFinal`
  - `IsSystem`
  - `IsError`

- `IsError`
- `LifeCycleInstanceFlag` LifeCycleInstanceFlag
  - `Active`
  - `Suspended`
  - `Completed`
  - `Failed`
  - `Archived`
- `LifeCycleTransitionFlag` LifeCycleTransitionFlag
  - `IsSystem`
  - `IsAuto`
  - `IsManual`
  - `NeedsApproval`
  - `CanRetry`
  - `IsCritical`
  - `IsRollback`
- `LifeCycleTransitionLogFlag` LifeCycleTransitionLogFlag
  - `System`
  - `Manual`
  - `Retry`
  - `Rollback`

5.2 Core C# models

- `LifeCycleDefinition` – matches `definition` . LifeCycleDefinition
- `LifeCycleDefinitionVersion` – matches `def_version` . LifeCycleDefinitionVersion
- `LifeCycleState` – state info plus flags and category. LifeCycleState
- `LifeCycleEvent` , `LifeCycleTransition` .
- `LifeCycleInstance` – runtime row (includes `ExternalRef` and `ExternalType`). LifeCycleInstance
- `LifeCycleTransitionLog` – in-memory representation of `transition_log` . LifeCycleTransitionLog
- `TransitionEventArgs` – event args for hooks (`Log` , `Exception` , `Context` , `MessageId`). TransitionEventArgs

6. Query and Data-Access Conventions

6.1 `QueryFields`

All SQL uses named placeholders from `Haley.Internal.QueryFields` :

- Generic: `@ID` , `@GUID` , `@DEF_VERSION` , `@PARENT` , `@VERSION`
- Common: `@DISPLAY_NAME` , `@NAME` , `@DESCRIPTION` , `@CATEGORY` , `@DATA` , `@ENV`
- State/Event/Transition: `@FROM_STATE` , `@TO_STATE` , `@EVENT` , `@FLAGS` , `@GUARD_KEY`
- Instance: `@CURRENT_STATE` , `@EXTERNAL_REF`
- Logs: `@INSTANCE_ID` , `@ACTOR` , `@METADATA`
- Audit: `@CREATED` , `@MODIFIED` , `@TRANSITION_LOG` , `@MESSAGE_ID` , `@RETRY_AFTER_MIN`

QueryFields

6.2 `QRY_*` classes

Each table has a corresponding `QRY_*` class with SQL as `const string` (single-line or minimal multi-line). Examples:

- `QRY_STATE` for the `state` table. QRY_STATE
- `QRY_EVENT` for the `events` table. QRY_EVENT
- `QRY_INSTANCE` , `QRY_TRANSITION` , `QRY_TRANSITION_LOG` .
- `QRY_DEFINITION` , `QRY_DEF_VERSION` , `QRY_CATEGORY` , `QRY_ACK_LOG` , `QRY_MAINTENANCE` .

Typical pattern:

- SQL string uses placeholders from `QueryFields` .
- Data is bound using `IAdapterGateway` methods with `(placeholder, value)` tuples.

Example – `QRY_STATE.INSERT` (simplified):

csharp Copy code

```csharp
public const string INSERT =
    $@"INSERT IGNORE INTO state (display_name, flags, category, def_version)
       VALUES ({DISPLAY_NAME}, {FLAGS}, {CATEGORY}, {DEF_VERSION});
       SELECT id FROM state WHERE name = lower({DISPLAY_NAME}) AND def_version =
```

Example – repository usage: SM.Maria.State

csharp Copy code

```csharp
_agw.ScalarAsync<long>(_key, QRY_STATE.INSERT,
    (DISPLAY_NAME, displayName),
    (FLAGS, (int)flags),
    (CATEGORY, category),
    (DEF_VERSION, defVersion));
```

Rule: Do not inline values directly into SQL. Always use placeholders and pass values via `IAdapterGateway` .

7. Repository Abstraction: `ILifeCycleStateRepository`

`ILifeCycleStateRepository` defines everything the state machine needs to operate.

7.1 Definition and version

- `RegisterDefinition(displayName, description, env)`
- `RegisterDefinitionVersion(parentId, version, jsonData)`
- `GetAllDefinitions()` , `GetDefinitionById(id)`
- `GetVersionsByDefinition(definitionId)`
- `GetLatestDefinitionVersion(definitionId)`
- `DefinitionExists(displayName, env)`
- `UpdateDefinitionDescription(definitionId, newDescription)`
- `DeleteDefinition(definitionId)`

7.2 State

- `RegisterState(displayName, defVersion, LifeCycleStateFlag flags, int category = 0)`
- `GetStatesByVersion(defVersion)`
- `GetStateByName(defVersion, name)`
- `GetInitialState(defVersion)`
- `GetFinalState(defVersion)`
- `UpdateStateFlags(stateId, newFlags)`
- `DeleteState(stateId)`

7.3 Event

- `RegisterEvent(displayName, defVersion)`
- `GetEventsByVersion(defVersion)`
- `GetEventByName(defVersion, name)`

- `DeleteEvent(eventId)`

7.4 Transition

- `RegisterTransition(fromState, toState, eventId, defVersion, flags, guardCondition = null)`
- `GetTransitionsByVersion(defVersion)`
- `GetTransition(fromState, eventId, defVersion)`
- `GetOutgoingTransitions(fromState, defVersion)`
- `DeleteTransition(transitionId)`

7.5 Instance

- `RegisterInstance(defVersion, currentState, lastEvent, externalRef, LifeCycleInstanceFlag flags)`
- `GetInstanceById(id)`
- `GetInstanceByGuid(guid)`
- `GetInstancesByRef(externalRef)`
- `GetInstancesByState(stateId)`
- `GetInstancesByFlags(flags)`
- `UpdateInstanceState(instanceId, newState, lastEvent, flags)`
- `MarkInstanceCompleted(instanceId)`
- `DeleteInstance(instanceId)`
- `UpdateInstanceStateByGuid(guid, newState, lastEvent, flags)`
- `MarkInstanceCompletedByGuid(guid)`
- `DeleteInstanceByGuid(guid)`

7.6 Transition log

- `LogTransition(instanceId, fromState, toState, eventId, actor, LifeCycleTransitionLogFlag flags, metadata = null)`
- `GetLogsByInstance(instanceId)`
- `GetLogsByStateChange(fromState, toState)`
- `GetLogsByDateRange(from, to)`
- `GetLatestLogForInstance(instanceId)`

7.7 ACK, maintenance, categories

- `PurgeOldLogs(daysToKeep)`

- `CountInstances(defVersion, flagsFilter = 0)`
- `RebuildIndexes()`
- ACK:
  - `Ack_Insert(messageId, transitionLogId)`
  - `Ack_MarkReceived(messageId)`
  - `Ack_GetPending(retryAfterMinutes)`
  - `Ack_Bump(ackId)`
- Categories:
  - `InsertCategoryAsync(displayName)`
  - `GetAllCategoriesAsync()`
  - `GetCategoryByNameAsync(name)`

`ThrowExceptions` toggles whether repository failures should throw exceptions or just set `IFeedback.Status = false`.

8. MariaDB Implementation: `LifeCycleStateMariaDB`

`LifeCycleStateMariaDB` is a partial class that implements `ILifeCycleStateRepository` by delegating to `IAdapterGateway`.

Key points:

- Holds:
  - `_agw`: `IAdapterGateway`
  - `_key`: DB connection key
  - `_logger`
  - `ThrowExceptions` flag
- Implementation pattern (example: state methods):
  - `RegisterState → _agw.ScalarAsync<long>(_key, QRY_STATE.INSERT, ...)`
  - `GetStatesByVersion → _agw.ReadAsync(_key, QRY_STATE.GET_BY_VERSION, ...)`
  - `UpdateStateFlags → _agw.NonQueryAsync(_key, QRY_STATE.UPDATE_FLAGS, ...)`

Each table's operations are grouped into separate partials (`SM.Maria.State.cs`, `SM.Maria.Transition.cs`, `SM.Maria.TransitionLog.cs`, definition, category, ack, maintenance, instance files, etc.), but all belong to the same `LifeCycleStateMariaDB` class.

9. State Machine Service: `ILifeCycleStateMachine` / `LifeCycleStateMachine`

9.1 Public API

`ILifeCycleStateMachine` exposes: ILifeCycleStateMachine

- Instance operations:
  - `GetInstanceAsync(string externalRefType, Guid externalRefId)`
  - `GetInstanceAsync<TEntity>(Guid externalRefId)`
  - `InitializeAsync(string externalRefType, Guid externalRefId, int definitionVersion)`
  - `InitializeAsync<TEntity>(Guid externalRefId, int definitionVersion)`
- Transitions:
  - `TriggerAsync(string externalRefType, Guid externalRefId, Guid toStateId, string? comment = null, object? context = null)`
  - `TriggerAsync<TEntity>(Guid externalRefId, Guid toStateId, string? comment = null, object? context = null)`
  - `ValidateTransitionAsync(Guid fromStateId, Guid toStateId)`
- State & history:
  - `GetCurrentStateAsync(string externalRefType, Guid externalRefId)`
  - `GetCurrentStateAsync<TEntity>(Guid externalRefId)`
  - `GetTransitionHistoryAsync(string externalRefType, Guid externalRefId)`
  - `GetTransitionHistoryAsync<TEntity>(Guid externalRefId)`
- Force operations:
  - `ForceUpdateStateAsync(string externalRefType, Guid externalRefId, Guid newStateId, LifeCycleTransitionLogFlag flags = LifeCycleTransitionLogFlag.System)`
  - `ForceUpdateStateAsync<TEntity>(Guid externalRefId, Guid newStateId, LifeCycleTransitionLogFlag flags = LifeCycleTransitionLogFlag.System)`
- State checks:
  - `IsFinalStateAsync(Guid stateId)`
  - `IsInitialStateAsync(Guid stateId)`
- Import:
  - `ImportDefinitionFromFileAsync(string filePath)`

  - `ImportDefinitionFromJsonAsync(string json)`

Additional methods from `LCSM.Core.cs` : LCSM.Core

- `ReceiveAckAsync(string messageId)`
- `RetryUnackedAsync(int retryAfterMinutes = 2)`
- `RegisterGuard(string transitionKey, Func<object, Task<bool>> guardFunc)`

Events:

- `OnBeforeTransition`
- `OnAfterTransition`
- `OnTransitionFailed`

Each event receives `TransitionEventArgs` .

9.2 Core logic overview

9.2.1 Instance retrieval and initialization

- `GetInstanceAsync(externalRefType, externalRefId)` :
  - Calls `_repo.GetInstancesByRef(externalRefId.ToString())` .
  - Filters the result list by `external_type` matching `externalRefType` (case-insensitive).
  - Maps the dictionary row to `LifeCycleInstance` .
- `InitializeAsync(externalRefType, externalRefId, definitionVersion)` :
  - Calls `_repo.GetInitialState(definitionVersion)` to find the starting state.
  - Calls `_repo.RegisterInstance(definitionVersion, initStateId, 0, externalRefId.ToString(), LifeCycleInstanceFlag.Active)` .

9.2.2 Triggering a transition

`TriggerAsync` (simplified flow) LCSM.Core

1. Load instance by external type + GUID.
2. Get outgoing transitions from current state:
   `_repo.GetOutgoingTransitions(fromState, instance.DefinitionVersion)` .
3. Find the transition whose `to_state` matches `toStateId` .
4. Compute a transition key:
   - Prefer the `event` field from the row (converted to string).
   - Fallback: `"T_{fromState}_{toStateId}"` .
5. Guard evaluation:
   - If `guards.TryGetValue(transitionName, out guardFunc)` :

    - Call `await guardFunc(context)`.
    - If false → throw and fail.
6. Build an in-memory `LifeCycleTransitionLog` object (not saved yet).
7. Raise `OnBeforeTransition`.
8. Persist:
    - `_repo.LogTransition(...);`
    - `_repo.UpdateInstanceState(...);`
9. Generate a new `messageId` (GUID), and call:
    - `_repo.Ack_Insert(messageId, transitionLogId)`

    to record the pending ACK.
10. Raise `OnAfterTransition`.
11. On any exception:
- Raise `OnTransitionFailed`.
- Return `false`.

9.2.3 Guards

- `RegisterGuard(string transitionKey, Func<object, Task<bool>> guardFunc)` stores a guard handler keyed by transition key or event name. LCSM.Core
- When triggering, the state machine:
    - Looks up `_guards[transitionName]`.
    - If found, uses the `context` argument from `TriggerAsync` as the guard input.

9.2.4 State queries and history

From `LCSM.State.cs`: LCSM.State

- `GetCurrentStateAsync(...)`
    - Gets instance.
    - Uses `_repo.GetStateByName(instance.DefinitionVersion, instance.CurrentState.ToString())`.
    - Maps to `LifeCycleState`.
- `GetTransitionHistoryAsync(...)`
    - Gets instance.
    - Calls `_repo.GetLogsByInstance(instance.Id)`.
    - Maps each row to `LifeCycleTransitionLog`.
- `ForceUpdateStateAsync(...)`
    - Writes a `transition_log` row (actor “system”, flags as provided).

  - Calls `_repo.UpdateInstanceState(instance.Id, newStateId, 0, instance.Flags)` .
- `IsFinalStateAsync` , `IsInitialStateAsync`
  - Use `_repo.GetStateByName(...)` and check `IsFinal` or `IsInitial` flags.

10. JSON Definition Importer

10.1 JSON model

- `DefinitionJson` :
  - `environment` – string (“dev”, “test”, “prod”, or numeric).
  - `definition` – `DefinitionSpec` (name, version, version_code, description).
  - `states` – list of `StateSpec` .
  - `events` – optional list of event names.
  - `transitions` – list of `TransitionSpec` . DefinitionJson
- `StateSpec` :
  - `name`
  - `is_initial`
  - `is_final`
  - `category`
  - `flags` (list of string tokens mapping to `LifeCycleStateFlag`). StateSpec
- `TransitionSpec` :
  - `from`
  - `event`
  - `to`
  - `guard`
  - `flags` (list of string tokens mapping to `LifeCycleTransitionFlag`). TransitionSpec
- `DefinitionSpec` , `DefinitionLoadResult` describe definition metadata and import result.

10.2 Import flow

`LifeCycleStateMachine.ImportDefinitionFromJsonAsync(string json)` does the following:

1. Deserialize JSON to `DefinitionJson` using a `JsonSerializerOptions` that:

    - Is case-insensitive.
    - Ignores null values when writing.
2. Normalize:
    - Default environment to “Development” if missing.
    - Ensure `definition.name` is present.
    - Ensure there is at least one state.
    - If no state is marked `is_initial`, mark the first one as initial.
3. Resolve environment → integer (`0`, `1`, `2`, etc.) via:
    - “dev” / “development” → 0
    - “test” / “qa” / “staging” → 1
    - “prod” / “production” → 2
    - numeric string → parsed value.
4. Definition handling:
    - Check `DefinitionExists(name, env)`.
    - If exists, search `GetAllDefinitions()` for matching `display_name + env`.
    - If not found, call `RegisterDefinition`.
    - Get `definitionId`.
5. Version handling:
    - Convert `version` string and/or `version_code` to an integer using `ParseVersionInt`.
    - Call `RegisterDefinitionVersion(definitionId, versionCode, json)` with full JSON as `data`.
    - Capture `defVersionId`.
6. Categories:
    - Collect unique `StateSpec.Category` values.
    - For each:
        - Try `GetCategoryByNameAsync`.
        - If not found, call `InsertCategoryAsync`.
    - Build `categoryNameToId` map.
7. States:
    - For each `StateSpec`:
        - Build flags with `BuildStateFlags`.
        - Map category string → category id (or 0 if none).
        - Try `GetStateByName(defVersionId, stateName)`.

        - If not found, call `RegisterState`.
    - Build `stateNameToId` map.
8. Events:
    - Start from `DefinitionJson.Events`.
    - Add all `TransitionSpec.Event` values.
    - For each distinct event name:
        - Try `GetEventByName`.
        - If not found, call `RegisterEvent`.
    - Build `eventNameToId`.
9. Transitions:
    - For each `TransitionSpec`:
        - Map `from` / `to` state names via `stateNameToId`.
        - Map `event` via `eventNameToId`.
        - Build flags via `BuildTransitionFlags`.
        - Try `_repo.GetTransition(fromId, eventId, defVersionId)`.
        - If missing, call `RegisterTransition(fromId, toId, eventId, defVersionId, flags, guard)`.
10. Result:
    - Return `DefinitionLoadResult` with IDs, environment, version code, and counts.

11. ACK / Outbox Mechanism

The ACK subsystem provides a minimal outbox to track whether external systems have acknowledged transition notifications.

- When a transition succeeds:
    - The state machine writes to `transition_log`.
    - It updates the instance state.
    - It generates a `messageId` (GUID).
    - It calls `_repo.Ack_Insert(messageId, transitionLogId)` to record a pending ACK.
- `ReceiveAckAsync(messageId)`:
    - Calls `_repo.Ack_MarkReceived(messageId)` to set `ack_status = 2`. LCSM.Core
- `RetryUnackedAsync(retryAfterMinutes)`:
    - Calls `_repo.Ack_GetPending(retryAfterMinutes)` to find `ack_status = 1` rows

older than the threshold.

  - For each row:
    - Reads `ack_log.id`, `transition_log`, `message_id`.
    - (Place where you can re-publish the notification).
    - Calls `_repo.Ack_Bump(ackId)` to increment `retry_count` and update `last_retry`.

This is intentionally simple, but enough to plug into a message bus or notification system.

12. Initialization

`LifeCycleInitializer.InitializeAsync(IAdapterGateway agw, string adapterKey)` is responsible for provisioning the lifecycle DB: LifeCycleInitializer

- Loads the embedded SQL resource.
- Replaces the default DB name placeholder with the target DB name.
- Calls `agw.CreateDatabase` with:
  - Processed SQL content.
  - `FallBackDBName` set to a default like “hdb_lc_state”.

Usage example:

csharp Copy code

```csharp
await LifeCycleInitializer.InitializeAsync(adapterGateway, "lc-state");
```

13. Integration Guide

13.1 Register in DI

csharp Copy code

```csharp
// 1. Run once at startup / migration time
await LifeCycleInitializer.InitializeAsync(adapterGateway, "lc-state");

// 2. Register repository
services.AddSingleton<ILifeCycleStateRepository>(sp =>
{
```

```
{
    var agw = sp.GetRequiredService<IAdapterGateway>();
    var logger = sp.GetRequiredService<ILogger<LifeCycleStateMariaDB>>();
    return new LifeCycleStateMariaDB(agw, "lc-state", logger, throwExceptions: tr
});

// 3. Register state machine
services.AddScoped<ILifeCycleStateMachine, LifeCycleStateMachine>();
```

13.2 Import a definition

csharp Copy code

```csharp
var sm = scope.ServiceProvider.GetRequiredService<ILifeCycleStateMachine>();

var fb = await sm.ImportDefinitionFromJsonAsync(jsonDefinition);
if (!fb.Status)
{
    throw new Exception(fb.Message);
}

int defVersionId = fb.Result.DefinitionVersionId;
```

13.3 Start and drive a lifecycle

csharp Copy code

```csharp
Guid externalRefId = Guid.NewGuid(); // external entity ID (e.g. workflow)
// Initialize instance
await sm.InitializeAsync<MyWorkflow>(externalRefId, defVersionId);

// Trigger transition
bool ok = await sm.TriggerAsync<MyWorkflow>(
    externalRefId,
    targetStateGuid,
    comment: "User submitted",
    context: new { UserId = "U1" });

// Query current state
var currentState = await sm.GetCurrentStateAsync<MyWorkflow>(externalRefId);
```

```
// Get history
var history = await sm.GetTransitionHistoryAsync<MyWorkflow>(externalRefId);
```

13.4 Registering guards

csharp Copy code

```csharp
sm.RegisterGuard("Approve", async ctx =>
{
    var c = (MyGuardContext)ctx;
    return c.HasPermission && c.Score >= 80;
});
```

14. Notes and Instructions for Future Changes / LLMs

1. **Naming conventions**
   - All lifecycle flags must be prefixed with `LifeCycle*` to avoid collisions.
   - Repository: `ILifeCycleStateRepository` .
   - MariaDB implementation: `LifeCycleStateMariaDB` .
   - Service: `ILifeCycleStateMachine / LifeCycleStateMachine` .
2. **SQL and DB**
   - Keep queries in `QRY_*` classes.
   - Always use `QueryFields` placeholders and parameter binding via `IAdapterGateway` .
   - Do not embed raw user input directly into SQL strings.
3. **Extending repository**
   - Add new `QRY_*` constants.
   - Add corresponding methods to `ILifeCycleStateRepository` .
   - Implement those methods in `LifeCycleStateMariaDB` via `_agw` .
4. **Extending state machine**
   - Keep lifecycle logic in `LifeCycleStateMachine` partials:
     - `LCSM.Core` – ACK, instance operations.
     - `LCSM.State` – state, history, final/initial checks.
     - `LCSM.Importer` – JSON definition import.
     - `LCSM.Helpers` – mapping, JSON helpers, environment & version parsing.

   - Use `TransitionEventArgs` to surface additional info to subscribers.
5. **JSON definitions**
   - Ensure `definition.name` is unique per environment.
   - Ensure at least one state and ideally an explicit `is_initial` state.
   - Ensure transitions reference valid state names and events.

You can now copy this entire text into Word as your lifecycle state machine README. If you want, I can next generate a few concrete JSON definition examples for your real-world processes (prequalification, onboarding, document review) that directly match this importer.