Haley Lifecycle State Machine — Plain Text Documentation

1. Purpose and scope
   This document explains the Haley Lifecycle State Machine end-to-end. It covers the intent, architecture, directory and file responsibilities, data model, repository and adapter gateway contracts, the JSON importer and required JSON shape, lifecycle flows, flag semantics, error handling and edge cases, performance and operational guidance, testing, and extension points. It is written for a new engineer who has not seen the codebase but needs to become productive quickly.

2. Core intent and principles
   The system provides a standalone, database-backed lifecycle tracker for any business entity. The lifecycle is defined once as a Definition and Version, and executed via Instances. Transitions between States are triggered by Events. Every change is audited via Transition Logs. External systems interact using a GUID-centric surface so callers never need internal numeric IDs. Flags are stored as bitwise integers to efficiently represent state attributes and instance lifecycle statuses. All SQL is single-line and parameterized via uppercase QueryFields placeholders. Boilerplate database execution is eliminated by helper methods implemented in the Adapter Gateway.

3. Naming, enums, and conventions
   All flag enums are prefixed with LifeCycle to avoid conflicts, for example LifeCycleStateFlag, LifeCycleTransitionFlag, LifeCycleTransitionLogFlag, LifeCycleInstanceFlag. Database queries are defined as single-line strings and interpolate placeholders like {ID}, {GUID}, {DEF_VERSION}, {FLAGS} that are provided by QueryFields. GUIDs are the external identity for instances; internal numeric IDs remain for joins and foreign keys. Inputs like state and event names are stored and looked up in a case-normalized fashion to ensure idempotency. For instance flags, the meanings are: Active = 1, Suspended = 2, Completed = 4, Failed = 8, Archived = 16. Bitwise operations are standard: OR to add a flag, AND with equality to test containment, and AND with complement to remove.

4. High-level architecture, how a client connects
   External applications call a LifeCycleStateMachine class that contains orchestration and domain-level flows, including a JSON definition importer. That class depends on ILifeCycleStateRepository, implemented by LifeCycleStateMariaDB. The repository uses an AdapterGateway abstraction to execute SQL against MariaDB. Client applications register the gateway, repository, and state machine in their dependency injection container. The gateway manages connection details and exposes both low-level operations (Read, Scalar, NonQuery) and standardized helpers (ReadAsync, ReadSingleAsync, ScalarAsync, NonQueryAsync) that return Feedback wrappers to unify success, message, trace, and result typing.

5. Directory and file responsibilities
   5.1 LifeCycleStateMariaDB.cs
   This is the root repository class (partial). It wires IAdapterGateway, ILogger, a connection key string, a ThrowExceptions behavior flag, and any shared caches (such as handler dictionaries) used across partials. It does not contain method bodies; those live in partial files grouped by domain (state, event, transition, etc.).

5.2 SM.Maria.Category.cs
Contains repository methods for category maintenance. Key methods: insert category by display name; fetch all categories; fetch a category by name. Methods call AdapterGateway helpers to return typed Feedback.

5.3 SM.Maria.Definition.cs
Contains repository methods for lifecycle definitions and versions. Key methods: register definition (name, description, env), register definition version (parent definition id, version integer, original JSON snapshot), get all definitions, get a definition by id, list versions for a definition, get latest version for a definition, check definition existence by name and environment, update definition description, delete definition. The version integer is policy driven: either an explicit version_code integer or a computed code from a dotted version string (for example X.Y.Z mapped to X*10000 + Y*100 + Z). The data column on def_version stores the original JSON for traceability.

5.4 SM.Maria.Event.cs
Contains repository methods for events per definition version. Key methods: register event with a display name, fetch events by version, get an event by name within a version, delete event by id.

5.5 SM.Maria.State.cs
Contains repository methods for states per definition version. Key methods: register state (display name, version id, state flags, optional category id), list states by version, get state by name (case-normalized), get initial state for version, get final state for version, update state flags, delete state. State flags combine bits like IsInitial, IsFinal, IsError, IsSystem. Categories are optional and used for grouping and reporting.

5.6 SM.Maria.Transition.cs
Contains repository methods for transitions. Key methods: register transition (from_state, to_state, event, definition version, transition flags, optional guard key), list transitions by version, get a specific transition by from_state and event for a given version, list outgoing transitions for a state in a version, delete transition by id. The guard key is a string token that the application can use to look up a guard strategy; the repository stores it as-is.

5.7 SM.Maria.TransitionLog.cs
Contains repository methods for transition logs (audit trail). Key methods: insert a log entry for a transition (instance id, from state, to state, event id, actor string, log flags, optional metadata), list logs for an instance, list logs by a particular state change, list logs by date range, get latest log for an instance. Transition logs are append-only and ordered by created timestamp.

5.8 SM.Maria.Ack.cs
Contains repository methods for message acknowledgment tracking (outbox-like reliability). Key methods: insert an ack row binding a message id to a transition log id, mark a message id as received, fetch pending acknowledgments that need retry after a configurable age, bump an acknowledgment’s attempt or timestamp. This supports reliable integrations with external systems.

5.9 SM.Maria.Instance.cs
Contains repository methods for runtime instances. Design is GUID-centric for all external operations, while preserving id-based methods for internal joins. Key methods: register instance (def_version, current_state, last_event, external_ref, instance flags) and return both id and guid; get instance by internal id; get instance by

guid; list instances by external_ref; list by current_state; list by flags using the bitwise filter pattern; update instance by id; update instance by guid; mark instance completed by id; mark instance completed by guid; delete instance by id or by guid. Insert uses an idempotent pattern: insert ignore and select by unique (def_version, external_ref), returning id and guid consistently whether the row existed or not.

5.10 SM.Maria.Maintenance.cs
Contains maintenance operations. Key methods: purge old logs older than a configured number of days (returns rows affected), count instances for a definition version with optional flags filter, rebuild indexes. The rows-affected semantics for purging are preserved.

5.11 LifeCycleStateMachine.cs (plus the import partial)
Contains orchestration and high-level flows used by client applications. The import partial provides two entry points: import from file and import from JSON string. It normalizes and validates the JSON, maps environment labels to integers, registers the definition and a new definition version, ensures categories, states, events, and transitions exist, and returns an ImportResult containing definition id, definition version id, environment, computed version code, and counts of created items.

5.12 AdapterGateway.* (Base, Configuration, CRUD, CRUDEx, Global, Old, IAdapterCrudHandler, IAdapterCrudHandlerEx, IAdapterGateway)
The AdapterGateway family provides the data access layer. It exposes low-level methods Read, Scalar, and NonQuery that accept AdapterArgs (including the connection key and query string) plus (key, value) parameter tuples. On top of those, the CRUDEx extension defines standardized helpers:
ReadAsync returns a list of dictionaries and fails gracefully when no rows are found.
ReadSingleAsync sets the FirstDictionary filter and returns one row as a dictionary.
ScalarAsync returns a typed scalar with special handling for long, int, and bool; it also understands boolean encodings such as bit(1), tinyint(1), single-byte arrays and strings.
NonQueryAsync returns a boolean success result after executing mutations.
These helpers return Feedback wrappers to unify success, message or trace, and results, and they eliminate per-method boilerplate from the repository partials.

6. Data model and schema overview
   The schema lives under the lifecycle_state database. Major tables and core columns are as follows:
   definition: id, display_name (or name), env (integer), description, created and modified. One row per high-level lifecycle blueprint.
   def_version: id, parent (definition id), version (integer), data (original JSON), created and modified. Immutable snapshots of a definition configuration.
   category: id, display_name (or name). Used to group states logically (for example business, system, error).
   state: id, def_version, display_name (or name), flags (bitwise LifeCycleStateFlag), category id, created and modified. Per version state nodes; one initial state and typically one final state.
   events: id, def_version, display_name (or name), created and modified. Triggers that cause transitions.
   transition: id, def_version, from_state, to_state, event, flags (LifeCycleTransitionFlag), guard_key string, created and modified. Edges in the state graph. A guard_key identifies optional guard logic.
   instance: id (bigint auto increment), guid (char 36 unique), def_version, current_state, last_event, flags

(bitwise LifeCycleInstanceFlag), external_ref (char 36 or string), created and modified. Runtime execution pointer for a specific entity. Unique constraint on (def_version, external_ref) guarantees idempotent instance creation. Indices on current_state, last_event, def_version, external_ref support lookups.
transition_log: id, instance_id, from_state, to_state, event_id, actor string, flags (LifeCycleTransitionLogFlag), metadata, created and modified. Append-only audit trail.
ack_log: id, message_id, transition_log_id, timestamps, counters, status fields to support reliable delivery and retries.
Flag storage is integer bitfields. Standard bitwise expressions let you filter by containment. The canonical SQL filter reads: select rows where (flags AND mask) equals mask. That returns rows containing all requested bits.

7. JSON importer shape and behavior
   The importer belongs to LifeCycleStateMachine. It accepts either a file path or a JSON string. The JSON must include:
   environment: a string such as Development, Test, or Production, or a numeric code; values are mapped to integers (0 for Development, 1 for Test or QA or Staging, 2 for Production).
   definition: an object containing name, optional version string, optional version_code integer, and optional description. If version_code is not provided, the dotted version string of the form X.Y.Z is converted to an integer using the policy X*10000 + Y*100 + Z. If neither is provided, a default version code is generated (for example 10000).
   states: an array of state objects containing name, optional is_initial boolean, optional is_final boolean, optional category string, and optional flags array of strings. If no initial state is marked, the importer marks the first state as initial.
   events: an optional array of event names. If omitted, the importer infers the event set from transitions.
   transitions: an array of transitions with from (state name), event (event name), to (state name), optional guard string, and optional flags array of strings.
   The importer performs these steps:
   Normalize and validate the JSON shape; ensure at least one state, ensure an initial state, and create an empty transitions list if missing.
   Resolve environment to an integer code.
   Register the definition by name and environment. If a matching definition exists, reuse it; otherwise create it.
   Register a new definition version with the computed integer version and store the original JSON in the data column.
   Ensure categories referred by states exist, creating any missing categories and mapping names to category IDs.
   For each state, attempt to fetch it by name for this definition version; if not found, register a new state with computed flags and optional category id.
   Assemble the event set, combining any explicit events with those referenced by transitions. For each event name, attempt to fetch it; otherwise register it.
   For each transition, resolve from_state id, to_state id, and event id. Attempt to fetch an existing transition using the from_state, event, and definition version key; if not found, register a new transition

with the provided flags and guard key.
Return an ImportResult containing definition id, definition version id, environment label, integer version code, and counts for states, events, and transitions created.

8. Adapter Gateway execution model
   The gateway receives AdapterArgs containing a database key and the SQL query string. Parameters are provided as tuples of (string key, object value) and the gateway is responsible for binding these into the execution. The low-level API mirrors common database operations:
   Scalar: returns a single scalar object.
   Read: returns either a list of dictionaries for multi-row results, or a single dictionary when Filter is FirstDictionary.
   NonQuery: executes mutations and returns implementation-specific results (often rows affected).
   The higher-level helpers standardize return types using Feedback wrappers and convert results to strong types. ScalarAsync supports long and int conversions and a permissive boolean conversion that handles bit(1), tinyint(1), numeric strings, and single-byte arrays, in addition to native booleans. ReadSingleAsync applies a FirstDictionary filter to Read and returns a single row as a dictionary. ReadAsync returns a list of rows. NonQueryAsync wraps a NonQuery call and returns a success boolean in a Feedback.

9. Repository surface (what clients call)
   Definitions: register definition, register definition version, list all definitions, get definition by id, list versions for a definition, get latest version for a definition, check definition existence by name and environment, update description, delete definition.
   States: register state, list states by version, get state by name, get initial state, get final state, update state flags, delete state.
   Events: register event, list events by version, get event by name, delete event.
   Transitions: register transition, list transitions by version, get a particular transition by from_state and event for a version, list outgoing transitions from a given state for a version, delete transition.
   Instances (GUID-centric): register instance returning id and guid, get by guid, get by id, list by external_ref, list by current_state, list by flags, update by id, update by guid, mark completed by id, mark completed by guid, delete by id, delete by guid. Insert is idempotent using the unique (def_version, external_ref) constraint and a follow-up select.
   Transition logs: log a transition and fetch logs by instance, by state change, by date range, and fetch the latest log for an instance.
   Acknowledgements: insert an ack record for a message id and a transition log, mark received by message id, get pending acks for retry (using an age threshold), and bump an ack.

10. Lifecycle flows and examples explained in prose
    Definition import: provide JSON with environment, definition details, states, events, and transitions. The system resolves or creates categories, states, and events, registers transitions, and returns identifiers to the caller. The version's JSON is preserved so you can always reconstruct the configuration used at the time of creation.
    Instance creation: when an external entity enters a lifecycle, fetch the initial state for the required definition version, then call the repository to create an instance providing the definition version id, the

initial state id, last_event set to none or zero, an external_ref to correlate with your domain record, and flags set to Active. The repository returns both a numeric id and a GUID. Persist that GUID in your system; all subsequent calls should use it.

Transition execution: to move an instance from its current state based on an event, read the instance by GUID to get the current_state id and numeric instance id. Resolve the event id (by name and version). Resolve the transition (by from_state, event, and version) to find the to_state id and any guard key. If application logic requires guard evaluation, perform it using the guard key as a lookup into a strategy registry. Log the transition with actor information and any metadata. Update the instance's current_state and last_event, maintaining the flags bitmask as needed (for example keeping Active set until completion).

Completion and failure: completion can be additive by setting the Completed bit without clearing other bits. Failure sets the Failed bit, optionally leaving historical bits intact. Archival is also additive; set Archived when the instance is no longer in active circulation.

Queries and filtering: filter by flags using the containment expression described earlier. To find all completed instances, match the Completed bit. To find instances that are both Suspended and Completed, build a mask containing both and require both bits to be set. To exclude a bit, use a negative check where the bitwise result equals zero.

11. Edge cases and correctness guarantees

    Missing initial state in JSON: the importer will mark the first state as initial if none is specified.

    Duplicate imports: the importer will reuse existing definition, states, events, and transitions where they already exist; it will only create what is missing.

    Case differences: state and event names are treated case-insensitively to avoid duplicates; storage and comparisons use normalized values.

    Guard keys: these are stored on transitions but not evaluated in the repository; application logic should map guard keys to strategies that accept contextual inputs and return allow or deny.

    Concurrent transitions: if multiple actors can move the same instance simultaneously, consider wrapping log and update in a database transaction and adding a row lock (for example select for update) around the instance row to serialize updates. Extend the repository to expose transactional wrappers if required by your platform.

    Flag overwrites: avoid assigning a raw flags integer when you only intend to add a bit. Always compute flags by OR-ing required bits with the existing mask or use repository helpers that are additive (for example a specific mark completed operation).

    Soft delete versus hard delete: prefer archiving instances by setting the Archived flag and retaining logs rather than deleting rows. Use the delete operations with care and only for cleanup tasks where referential integrity is guaranteed.

    External references: the unique constraint on (def_version, external_ref) ensures idempotency and provides a consistent upsert-like behavior via insert ignore and select; choose external_ref types that are stable and unique in your domain (GUIDs are a good choice).

    Boolean handling: Scalar conversions in the gateway cover bit(1), tinyint(1), byte arrays, numeric strings, and native booleans; this ensures consistent interpretation across MariaDB drivers.

12. Operational guidance

    Security: provision a least-privilege database user restricted to the lifecycle_state schema. Do not expose schema write permissions to untrusted services. Keep architectural diagrams and internal connection details confidential.

    Observability: log repository errors with query names or identifiers, not raw SQL strings. Track counts for transitions, instances by flag, ack retries, and importer activity. Include correlation ids where appropriate so a transition log entry can be tied to a client request id.

    Performance: the schema includes indices on current_state, last_event, def_version, and external_ref. Unique (def_version, external_ref) supports idempotent instance creation. If new hot paths emerge, add compound indices tailored to those queries. Keep all queries single-line and parameterized to reduce parsing overhead and avoid injection risks.

    Backups and retention: back up lifecycle_state independently of your application databases. Retain transition logs per your audit policy and use a purge process to remove aged logs beyond a retention threshold; measure the impact on analytics before enabling aggressive purging.

    Multi-tenancy: if you need tenant isolation in the same schema, add a tenant_id column in definition, def_version, state, event, transition, instance, and log tables; include tenant_id in unique constraints and all repository queries. Alternatively, create per-tenant schemas if that better suits your security boundaries.

13. Testing strategy and checklist

    Importer: test valid JSON, missing environment (ensure default), missing initial state (ensure first state becomes initial), unknown categories (ensure creation), duplicate imports (ensure idempotency).

    Definitions and versions: test version code derivation from dotted strings and explicit version_code, retrieval of latest version, scanning by name and environment.

    States: test lookups by name with case differences, correct flags mapping (IsInitial, IsFinal, IsError), updates to flags, and deletion constraints if any.

    Events: test registration and lookup by name, including duplicates across versions.

    Transitions: test registration with and without guard keys, retrieval by from and event, listing outgoing transitions, and deletion. Confirm that transitions cannot be created with unknown states or events.

    Instances: test idempotent creation by (def_version, external_ref), retrieval by guid and by id, updates by guid, marking completed by guid, deletion by guid, and filtering by flags. Confirm that flags are combined correctly and not overwritten inadvertently.

    Transition logs: test logging, listing by instance, listing by date range, listing by from-to pair, retrieving the latest log entry.

    Acknowledgements: test inserting an ack, marking received by message id, retrieving pending acks after a delay, and bumping an ack.

    Concurrency and transactions: test simultaneous transitions on the same instance, ensure no lost updates if you require transactional behavior, and verify behavior under retries from idempotent client calls.

    Failure paths: test malformed JSON, missing keys, unavailable database, missing foreign keys during registration, and permission errors. Ensure Feedback messages are actionable.

14. Extension points and roadmap

    Negative flag filters: introduce standardized queries for rows that do not contain certain flags using a pattern where the bitwise result equals zero.

    Compound filters: add repository methods that combine def_version and flags predicates in one SQL for performance.

    Transactional API: expose repository methods that start, commit, and roll back database transactions. Provide a unit-of-work pattern for combined operations like logging a transition and updating the instance atomically.

    Guard strategy registry: standardize a mechanism in LifeCycleStateMachine to resolve guard_key strings to strategies that can evaluate context and return allow or deny. Provide dependency injection points for guard providers.

    Bulk import: for very large graphs, add bulk registration endpoints for states, events, and transitions, and wrap them in a single transaction to optimize performance and ensure consistency.

    Tenant isolation: implement optional tenant_id plumbing as described in operational guidance, including changes to unique constraints and repository queries.

15. Summary of how everything fits together

    The LifeCycleStateMachine class orchestrates the lifecycle at a high level and provides the JSON importer so new lifecycles can be created from a declarative description that includes environment, definition, states, events, and transitions. The ILifeCycleStateRepository abstraction, implemented by LifeCycleStateMariaDB across partial files, is the single entry point to the database for all lifecycle operations, including definitions, versions, categories, states, events, transitions, instances, transition logs, and acknowledgements. The AdapterGateway removes boilerplate by exposing typed helpers and consistent Feedback results for scalar reads, list reads, single-row reads, and mutations. Instances are created and moved across the state graph using events; every move is logged. Flags are bitwise to support compact storage and efficient filtering. GUIDs are the external handle for instances so clients remain decoupled from the database’s internal numeric identifiers. The system is designed to be idempotent, auditable, and safe to call repeatedly, and it is ready to be extended with guard strategies, transactional flows, negative flag filters, and multi-tenant isolation as required.