# Haley Lifecycle State Machine — Plain Text Documentation

## 1. Purpose and scope

This document explains how the State Machine Coordinator (SMC) works together with the existing Haley Lifecycle State Machine and your application.

The goals are:

- Keep the lifecycle state machine focused only on the state of entities.
- Keep the application focused on business logic.
- Introduce a small, configurable coordinator that reacts to lifecycle events and calls application methods, without tight coupling or hard-coded wiring.
- Avoid building a full workflow engine or extra workflow database.

## 2. High-level architecture

There are three main parts:

- **Application**
    - Owns business logic: validation, notifications, assignments, file operations, etc.
    - Calls the lifecycle state machine when it wants to change the state of an entity (for example, when the user clicks “Submit” or “Approve”).
    - Exposes certain methods as “actions” that the SMC can call.
- **Lifecycle state machine**
    - Owns definitions, versions, states, events, transitions, instances, and transition logs.
    - Stores everything in its own lifecycle tables.
    - Provides methods like Initialize and Trigger (TriggerAsync) to change state.
    - Emits a notification when a transition completes (for example, an OnAfterTransition event with context).
- **State Machine Coordinator (SMC)**
    - Runs inside the same process as the application.
    - Subscribes to the state machine’s transition notification event.
    - Loads a JSON configuration that says: “*When this event happens, run these actions in this order.*”
    - Discovers application actions using attributes and a dictionary of keys → handlers.
    - Does not have its own workflow state or history; it is just orchestration glue.

Optionally, you can also add a Lifecycle Monitor:

- **Lifecycle Monitor**
    - Periodically queries lifecycle instances to detect long-running states (for example, “PendingClarification longer than N days”).
    - Triggers reminders or timeout transitions based on rules.
    - Uses the same lifecycle database and repository.

## 3. How state changes are triggered

The flow for state changes is:

1. A user or an internal process does something in the application (for example, submits a form, requests clarifications, validates a company).
2. The application decides which lifecycle event should happen and calls the lifecycle state machine directly. For example:
    - Initialize an instance for a new entity.
    - Trigger a transition using TriggerAsync or similar, passing the event and instance reference.
3. The lifecycle state machine:
    - Validates the transition according to its definition and guards.
    - Updates the instance's current state.
    - Writes a transition_log row.
    - Optionally writes an ack_log row.
    - Raises a transition notification event (for example, OnAfterTransition).
4. The SMC listens to this notification event and reacts based on its configuration.

The important point is:

- The application remains responsible for calling the state machine to change states.
- The state machine does not know about the application or the SMC; it only raises events.

## 4. State Machine Coordinator concept

The SMC is a very small “lite workflow” layer. It does not know about the lifecycle definition, transitions, or database schema. It only knows:

- “An event with this key has happened.”
- “When that event happens, here are the actions (methods) to run.”

**Key characteristics:**

- It listens to a single event from the lifecycle layer (for example, OnAfterTransition).
- It builds a simple eventKey string from the notification (for example, “VendorPreQualificationState.Submit”).
- It looks into its JSON configuration to see what to do for that eventKey.
- It runs the configured actions in order, with optional conditions and retry behaviour.
- It does not store its own state; it is short-lived per event.

**This allows:**

- State machine to remain reusable and independent.
- Application to keep its methods simple.
- Orchestration logic to be configured in JSON instead of hard-coded in the app.

## 5. Event model and event keys

To avoid mixing state knowledge into SMC, SMC uses only event keys.

When the lifecycle state machine raises the transition notification, SMC reads:

- Definition name (for example, VendorPreQualificationState).
- Event name (for example, Submit, ClarificationsRequested, ValidCompany).
- Instance identifiers, external references, metadata, message id, etc.

From the definition and event name, SMC builds a simple eventKey, for example:

- VendorPreQualificationState.Submit
- VendorPreQualificationState.ClarificationsRequested
- VendorRegistrationState.ValidCompany

All SMC configuration is written against these eventKey strings.

So:

- State names and transitions are defined only in the lifecycle JSON and database.
- SMC JSON never needs from_state or to_state; it only cares about “which event fired”.

## 6. SMC configuration (JSON)

The SMC reads a JSON configuration file that tells it:

- Which events to subscribe to.
- Which actions (handlers) to run when those events occur.
- Optional conditions and retry policies per action.

Conceptually, the JSON has:

- A coordinator section with basic metadata (name, version).
- A list of subscriptions.

Each subscription contains:

- Id
    - A unique identifier for this subscription (for example, “VPQ_Submit”).
- source
    - Where the event comes from. For now, this is “Lifecycle” (later can be “Application” or “Bus” if you want).
- eventKey
    - The string SMC should match (for example, “VendorPreQualificationState.Submit”).
- phase
    - Optional label indicating which business phase this belongs to (for example, FormIntake, ClarificationLoop, Tiering, RegistrationIntake).
- steps
    - An ordered list of actions to run when this eventKey occurs.

Each step in a subscription has:

- key
    - The action key that identifies which handler to run. It must match the key in the handler’s attribute.
- if (optional)
    - A simple condition string, evaluated against a shared bag of values; if this evaluates to false, the step is skipped.
- maxAttempts
    - How many times to retry this step if it fails.

- retryDelaySeconds
    - How long to wait between retries for this step.
- stopOnFailure
    - If true, and this step ultimately fails after retries, SMC stops running the remaining steps in this subscription.

Because everything is driven by eventKey and action key, the JSON does not need any details about lifecycle internals.

## 7. Action handlers and attributes

The application exposes small, focused pieces of logic as handlers that the SMC can call.

An action handler:

- Is typically a class implementing a simple interface (for example, ExecuteAsync).
- Performs one logical step, such as:
    - Calculate auto tier from form data.
    - Assign evaluators based on configuration.
    - Send vendor notification.
    - Aggregate evaluation scores.
    - Store registration documents in storage.

To connect handlers with SMC without hard-coding, you use an attribute, such as:

- LifeCycleAction("vpq_form_auto_tier")
- LifeCycleAction("vrg_validate_registration_form")

During startup, SMC:

1. Scans assemblies for classes or methods decorated with LifeCycleAction.
2. For each found handler, reads the key from the attribute.
3. Builds a dictionary mapping key → handler instance.
4. Validates that every key referenced in SMC JSON exists in this dictionary.

At runtime, when SMC sees a step with key "vpq_form_auto_tier", it looks up the handler with that key and calls it.

Each handler receives:

- A transition context (constructed from the lifecycle notification) containing:
    - Definition name and version.
    - Event name.
    - Instance id/guid.
    - External reference and type.
    - Metadata and message id.
- A bag dictionary (shared within a single subscription execution):
    - Handlers can write values into the bag (for example, requires_hsse = true).

- Later steps can read these values and their conditions (“if bag.requires_hsse”) can depend on them.

## 8. SMC runtime flow (step-by-step)

The typical runtime sequence is:

Step 1: Application starts

- The app registers the adapter gateway, lifecycle repository, and lifecycle state machine.
- The app registers SMC and its dependencies.
- SMC scans for handlers, builds the action dictionary, and loads its JSON configuration.
- The app wires SMC to the lifecycle notification event:
    - lifecycleStateMachine.OnAfterTransition += smc.HandleNotification.

Step 2: Application triggers a lifecycle event

- A user or system action in the app decides that a lifecycle event should occur:
    - For example, vendor clicks Submit.
    - The app calls lifecycleStateMachine.TriggerAsync or similar for this event on the relevant instance.

Step 3: Lifecycle state machine performs the transition

- The state machine:
    - Validates the event and transition.
    - Updates the instance's current state.
    - Writes a transition_log entry (and, if applicable, ack_log).
    - Raises the notification event with transition context.

Step 4: SMC receives the notification

- SMC's subscribed handler is called with the transition context.
- SMC builds eventKey from the definition name and event name.

Step 5: SMC matches subscriptions

- SMC finds all subscriptions where:
    - source is "Lifecycle".
    - eventKey matches the computed eventKey.

Step 6: SMC executes subscriptions
For each matching subscription:

- SMC creates a new empty bag dictionary.

- For each step in the subscription's steps list:
    - If there is an if condition:
        - SMC evaluates it using the current bag:
            - For example, "bag.requires_hsse" or "bag.pending_clarifications == false".
        - If the condition is false, SMC skips this step.
    - If the step should run:
        - SMC looks up the handler by the step key in the dictionary built at startup.
        - SMC calls the handler, passing the transition context and the bag.
        - If the handler throws or reports failure:
            - SMC retries up to maxAttempts, waiting retryDelaySeconds between tries.
            - If it still fails and stopOnFailure is true:
                - SMC stops processing further steps in this subscription.
                - SMC logs the error.

Step 7: Logging and observability

- SMC logs:
    - The eventKey.
    - The subscription id.
    - Each step's key and attempts.
    - Any errors or exceptions.
- Long-term auditing of state remains in the lifecycle transition_log tables.
- For v1, SMC itself does not write to its own database tables; you can rely on logs and metrics.

## 9. Retry behaviour and idempotency

Because SMC supports retries for each step, handlers should be designed to be safe to call multiple times for the same event.

This means:

- If the same lifecycle event triggers the same SMC step again (due to a retry or a re-run), the handler should not cause incorrect duplication or side effects.
- Typical strategies:
    - Use messageId and instance guid as an idempotency key in your own application storage.
    - Before sending an email or creating an external record, check whether an entry with this key was already processed.
    - Overwrite computed data instead of appending (for example, recalculate tier vs adding another row).

In short:

- SMC takes care of short-lived retries according to maxAttempts and retryDelaySeconds.
- Handlers take care of business-level idempotency so that repeated calls are safe.

## 10. Long-running states and timeouts

Some states represent waiting periods (for example, PendingClarification, UnderEvaluation, PendingValidation). These are not SMC's responsibility.

Instead, you can introduce a Lifecycle Monitor:

- It runs periodically (for example, via a background service or cron).
- It queries the lifecycle repository for instances that:
    - Are in a specific state.
    - Have stayed in that state longer than a configured threshold.
- When it finds such instances, it can:
    - Trigger a reminder notification by raising an internal event that SMC can handle (for example, a "ReminderSent" event).
    - Or call the lifecycle state machine to perform a timeout transition (for example, move to Overdue or Expired).

The lifecycle monitor's logic and thresholds are independent of SMC. When it does trigger events, those events still pass through the same SMC mechanism (eventKey → subscriptions → steps).

## 11. Design boundaries and guarantees

To keep the design clean:

- The lifecycle state machine:
    - Remains the only component that changes lifecycle state, validates transitions, and writes lifecycle history.
    - Does not call application methods and does not know about SMC.
- The application:
    - Decides when to change state and calls the lifecycle state machine for that.
    - Implements business handlers and decorates them with the LifeCycleAction attribute.
    - Does not contain hard-coded “if event X then call handler Y and Z”; that mapping lives in JSON.
- The State Machine Coordinator:
    - Listens to a single event from the lifecycle layer.
    - Uses eventKey and JSON subscriptions to decide what actions to run.
    - Uses attributes to resolve handlers.
    - Handles conditions and retries.
    - Does not maintain a separate workflow state machine or history database.

This keeps responsibilities clear and lets you evolve each part independently:

- You can change lifecycle definitions without touching SMC JSON (as long as event names stay stable).
- You can change which actions run for which events by editing JSON, without recompiling the app.
- You can add or remove handlers by adding or removing attribute-decorated classes.

## 12. Summary

- State changes are still driven by the application calling the lifecycle state machine.
- The lifecycle state machine remains the authoritative source for lifecycle state and history.
- The SMC is a lightweight orchestrator that:
    - Listens to lifecycle transition events.
    - Uses a simple eventKey scheme to match configuration.
    - Executes application handlers in a configurable order, with conditions and retries.
    - Stays stateless and does not duplicate lifecycle logic.

This gives you a “lite workflow engine” that is:

- JSON-driven,
- attribute-driven,
- easy to adjust at runtime,
- and fully aligned with your existing Haley lifecycle state machine.