What is State machine :

* Computation based on finite states.
* Usually, there are flows to go in sequence with the states or rules that should be followed, meaning that you can’t just go from any state to any other state.
* The transitions between these states are limited by the rules.
* State machines are a common way of describing a number of states in a business process.

When is the state machine needed?

* You are already trying to implement a state machine if you are trying to
* Using boolean flags or enums to handle different states in a flow or situations.
* If we are making multiple variables just to identify and switch flows in an application.
* Looping through if/else structure and checking if a particular flag or enum is set and then making further exceptions about what to do when certain combinations of your flags and enums exist or don’t exist together.
* States can be nested. Also, the states can be guarded with configured checks. and pseudo-states that allow defining a choice state, junction state etc.
* Events can be triggered based on the sequence of actions, or on a timer.
* The state machines can be persisted which will make them more performant and application can still be consistent when it faces a downtime.

Examples :

**Bots**: Bot usually has only a few states with different actions in between. For example, you have a bot who’s asking questions to book a hotel (a well-known example). You ask a few questions: location, number of guests, price range etc. Every question is a state. Every answer is an event that allows transitioning into the next state.

**IOT**: Here the state machine will have ON and OFF. but the complexity will be with a variety of devices, like a light, switch, there can be more states in between, and more events to make a state transition.

State Machine Major Flaws:

Case 1: State machines are simple in logic.

Suppose we are using a state machine for a lamp. In total, your lamp will have two states:

* Active, when the light is on,
* Inactive, when it is off.

To make the system work, it needs state transitions. In this case, it will be switch on and switch off commands which is a simple straight forward flow.

So writing a state machine for these flows is adding code/Complexity for no reason.

Case 2: Working with state machines seems to be quite simple if you have only two states and two types of state transitions. But developing a sophisticated system with numerous states or classes of states is very tedious. In each of the states, the behavior of a system is different and your major task is to characterize them all. You'll also need to define events that will make the system move from one state to another.

For instance, we are creating an online order system, to do that

**Maintenance issues** :

* you need to think about all the possible states of the system upfront, such as how it behaves when a customer places an order, what happens if it is not found on stock or if the customer decides to change or delete the chosen item.
* States could reach few 100’s very soon as you start specifying system behavior for each of the states and quite soon realize that your code becomes unreadable.
* Besides, managing states turns into a challenging process for any developer.

(i) what will you do when the system evolves and you need to add some new features?

(ii) You will have to fully reshape the machine.

(iii) Moreover, by adding new states and their transitions, the complexity of the code raises drastically making it harder to bring any new changes into the program. The system becomes unmanageable.

**Logging and Debugging :**

(iv) Complex state machines make maintenance and debugging rather problematic as well. You might have 1000 and more lines of asynchronous text log, that need to be analyzed.

(v) When you debug the code, you will soon find yourself working with “clumsy code”. If a certain issue occurs, it may take hours to finally realize where it comes from. As a result, system debugging turns into a nightmare.

Safeway Usecase:

As defined above state machine can be adopted in a scenario where there are a set of sequential steps that have to execute on specific rules defined.

In my view, we can consider using a state machine in the Order Management System,

Referring to the below order flow :



* In scope: Step 3 to Step 11 in order flow, which comes under the scope of Order management application can be considered to maintain under a single state machine.
* Each step can be persisted to the DB, By persisting the states into the DB the states of each order flow can be maintained even after an unanticipated downtime.
* Not In Scope: Step 1 and Step 2 in the data center scope do not give us any control to maintain/monitor their state.
* Not In Scope: Step 12 Is the place where VPOS is processing the Order and Proceeding payment which does not have any updates to OMS, SO this step is also not coming under the control of State machine.

Summary :

I am still analyzing other Cons and Pros for considering the usage of State Machines.

Will update this document even further and let you know the final summary.