Implementation of ReactiveUI

By Bani Dom

This is a work in progress and dated on March 16, 2024 6:49 PM

Table of Contents

Synopsis

I presented the idea during the Schinkels IT Team meeting on March 11, 2024. I see this concept as a way to remain competitive within our only current customer and increase market share from that customer. We are facing competition from new solution providers, and one has penetrated the Safwa ecosystem.

The purpose of this paper is to demonstrate a new methodology for improving both current and future applications.

In the industry, ReactiveUI has been introduced with various definitions. There is no finalization of the definition by any standards whatsoever. I coined it to our needs. My definition is adapted to our current needs and concentrates on what we are capable of doing now rather than in the future. It is also designed with the tools we currently use and have within our technology portfolio.

Technical Problem Statement:

The problem statement is: how can we make our developed apps better for users than what the Safwa team has produced using ReactiveUI methodology?

Business Problem Statement

My marketing strategy is to replace the current jPBT2u app with improved user interface components starting in the next 18 months, which will lead to a decrease in HR costs, including onboarding and quantum requirements. Two important technologies, including a user role-based system and a business process engine for Schinkels' apps, have already been implemented.

We anticipate a slower pace of upgrades or bug fixes at Safwa for all developed systems over the next 6 months. jPBT2u is an active development app. Changes are continuously made weekly to meet user demands. Starting in September 2024, there is a possibility of limited support availability within Safwa. Schinkels has an opportunity to offer support and suggest major upgrades due to this. However, on the flip side any major upgrade decision will diminished within 24 months as Safwa will undergo end of its 5-year contract with the government. There is a very strong possibility of continuation, as Safwa has a 5+5 contract in hand, but the uncertainty will reduce investment interest.

Schinkels must show a need to upgrade to show Safwa value creation for the second 5-year term for further continuation of service beyond the 10 years contract.

Introduction

ReactiveUI concept covers multitude of UI technology components. What I presented in our last meeting is just a simplification of only 1 component that is based on UI time-lapse stagnantTime methodology. We can define stagnantTime as the timelapse of last physical monitored event between user and interface to the current time.

$$\text{stagnantTime} = \text{lastMonitorEventTime} - \text{currentTime}()$$

where lastMonitorEventTime is the time of the last from a set of predefined monitor event inclusive of keyboard, pencil, mouse movement, or facial detection movement detected thru the OS APIs.

And

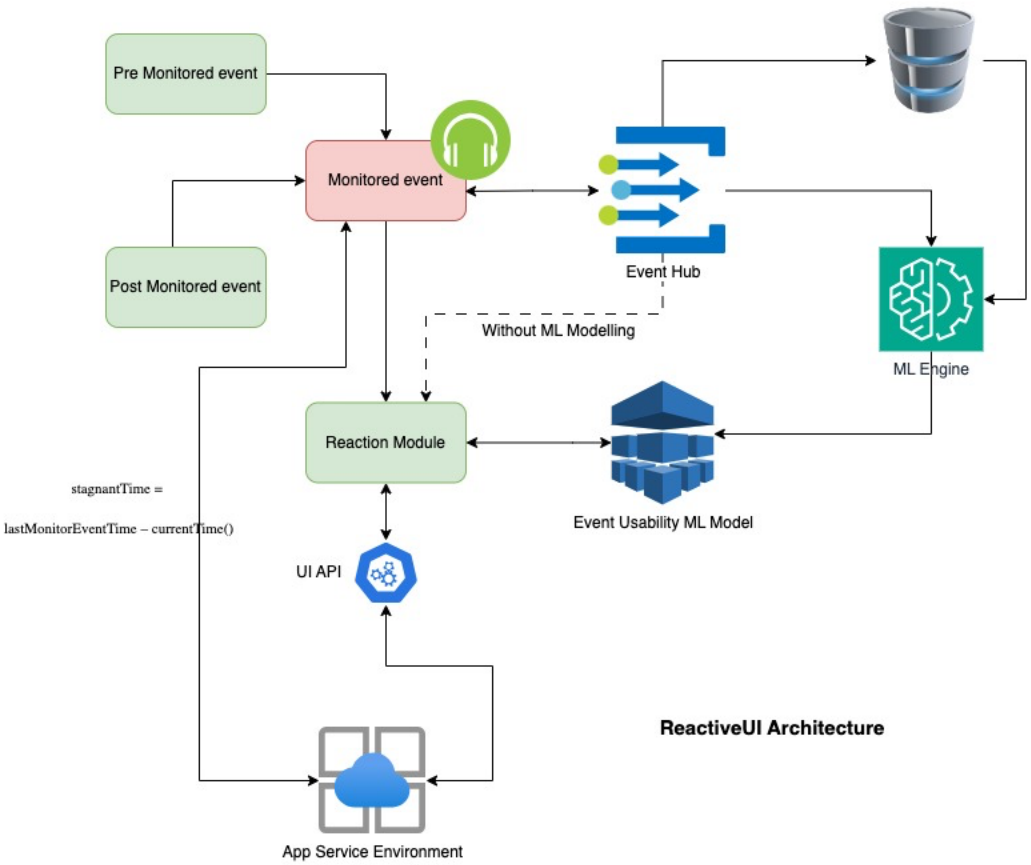
where currentTime is the current time

stagnantTime is therefore the lapse time since lastMonitorEventTime.

With the definition above is presented, we can continue our outline for our approach of ReactiveUI as follows:

1. Defined the set of monitored event.
2. Track usage pre and post monitor event.
3. Build a model of usability for each monitored event.
4. Defined a useable stagnantTime.
5. Produce reaction module for each of each stagnantTime.

ReactiveUI Architecture



The above represents the full-scale overview of the ReactiveUI architecture. Initial deployment can be achieved through a simpler architecture. But the overall design concept must in the direction of the presented architecture.

Solution Approach

The approach to solving the problem is straightforward.

The solution is to generate multiple options for the user when a stagnantTime is reached at a defined location in the UI, once the engine is running and the event model is defined.

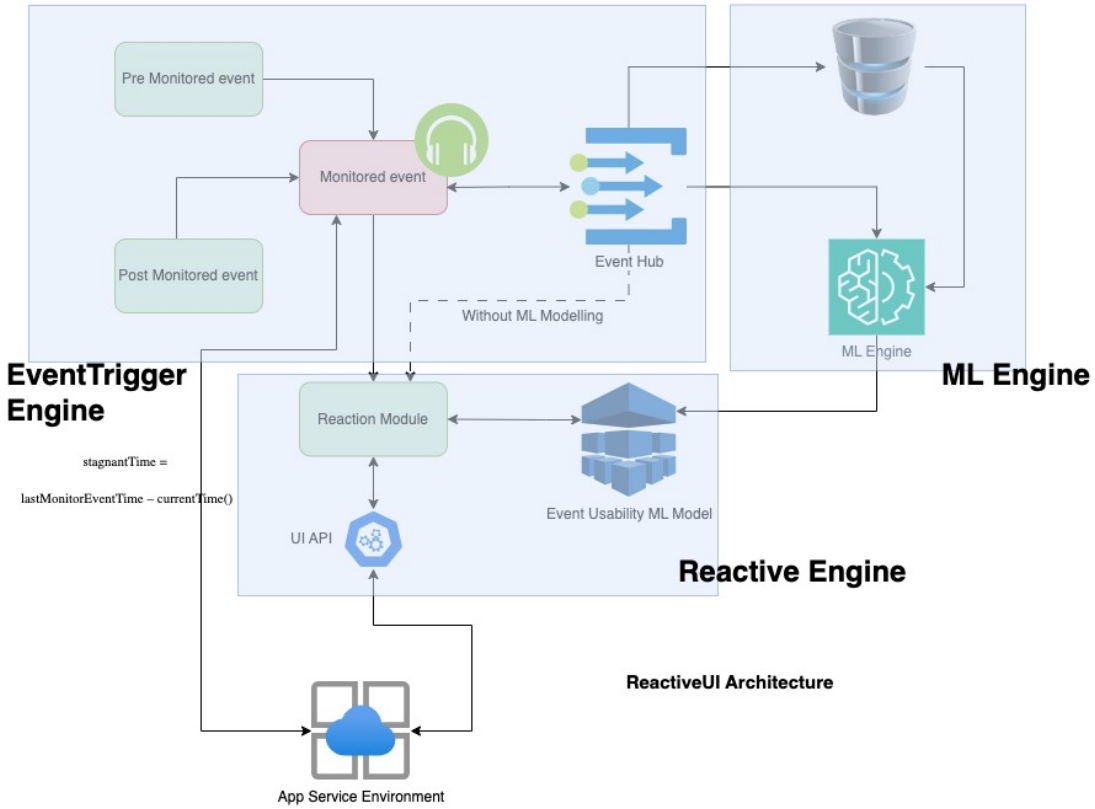
The method consists of onboarding users with the predictive ability. The predictive elements are based on the usage of the system and applications by various users.

The solution approach is targeted to reduce onboarding time by 90% and other personnel interruption to zero time.

Architecture Components

The reactive architecture can be broken down into three major components. The diagram below shows each of the working components that are addressed as engines. The engines work together as one integrated unit, but they can be customized to meet deployment needs.

Complete descriptions of each component can be found in the definitions below. However, every deployment will cater to some part of the described functionalities.



EventTrigger Engine

An event trigger engine is a component or system designed to detect and respond to specific events or triggers within a software application or a broader computing environment. These engines play a crucial role in event-driven architectures, where actions are initiated based on the occurrence of predefined events rather than through explicit user commands or scheduled tasks.

Here are some key aspects and functionalities of event trigger engines:

Event Detection: The primary function of an event trigger engine is to detect events or signals that indicate specific occurrences or changes in the system's state. These events could originate from various sources, including user interactions, system notifications, sensor data, or external services.

Event Filtering: Event trigger engines often include mechanisms for filtering and processing incoming events based on predefined criteria or rules. This allows developers to specify which events are relevant and should trigger a response, while disregarding irrelevant or redundant signals.

Event Handling: Once an event is detected and filtered, the event trigger engine initiates the appropriate response or action based on predefined logic or workflows. This response could involve executing specific functions or tasks, sending notifications, updating data, or triggering additional events in a cascading manner.

Asynchronous Processing: Event trigger engines typically operate asynchronously, allowing them to handle multiple events concurrently without blocking or waiting for each event to complete before processing the next. This enables efficient utilization of resources and ensures timely responses to incoming signals, even under high-load conditions.

Integration: Event trigger engines often integrate with other components or systems within the application architecture, such as message brokers, event buses, workflow orchestrators, or external APIs. This integration enables seamless communication and coordination between different parts of the system, facilitating complex event-driven workflows and interactions.

Scalability and Resilience: Event trigger engines are designed to be scalable and resilient, capable of handling large volumes of events and adapting to changes in workload or system conditions. They may employ distributed processing, load

balancing, and fault-tolerant mechanisms to ensure reliability and availability in distributed or cloud-based environments.

ML Engine

THE ML Engine is designed as a continuous learning ML engine.

Continuous machine learning, also known as continual learning or lifelong learning, refers to the capability of machine learning models to learn and adapt continuously over time as new data becomes available. Unlike traditional machine learning approaches where models are trained on fixed datasets and then deployed without further updates, continuous machine learning systems are designed to evolve and improve their performance incrementally as they are exposed to new data.

The key characteristic of continuous machine learning is its ability to adapt to changing environments and evolving data distributions. This is particularly important in dynamic domains where the underlying patterns and relationships may shift over time. By continuously updating their parameters and adjusting their predictions based on incoming data, these models can maintain relevance and accuracy over extended periods.

Continuous machine learning algorithms often incorporate techniques such as online learning, where models are updated in real-time as new data arrives, and incremental learning, where new knowledge is integrated gradually without discarding previous learning. Additionally, strategies such as model adaptation, transfer learning, and regularization are employed to prevent catastrophic forgetting and ensure that the model retains knowledge from past experiences while incorporating new information.

Reactive Engine

A reactive engine, in the context of computing or software engineering, refers to a type of system or component that reacts to stimuli or events in its environment. Unlike proactive systems that take initiative or predictive systems that anticipate future events, reactive engines respond to immediate inputs without necessarily considering long-term implications or planning.

Here are some key characteristics and considerations regarding reactive engines:

Event-Driven: Reactive engines typically operate in an event-driven manner, where they wait for specific events or signals to occur before triggering a response. These events could be user interactions, incoming data, or changes in the system state.

Asynchronous Processing: Reactive engines often handle events asynchronously, meaning they can process multiple events concurrently without blocking or waiting for one to complete before processing the next. This allows for efficient utilization of resources and responsiveness to high-throughput scenarios.

Statelessness: Reactive engines may operate in a stateless manner, where they don't maintain persistent state between interactions. Instead, they process each event independently, potentially updating their internal state or producing output based solely on the current event and context.

Scalability: Reactive engines are often designed to be scalable, capable of handling varying workloads and adapting to changes in demand. This scalability is achieved through techniques such as distributed processing, load balancing, and elastic resource allocation.

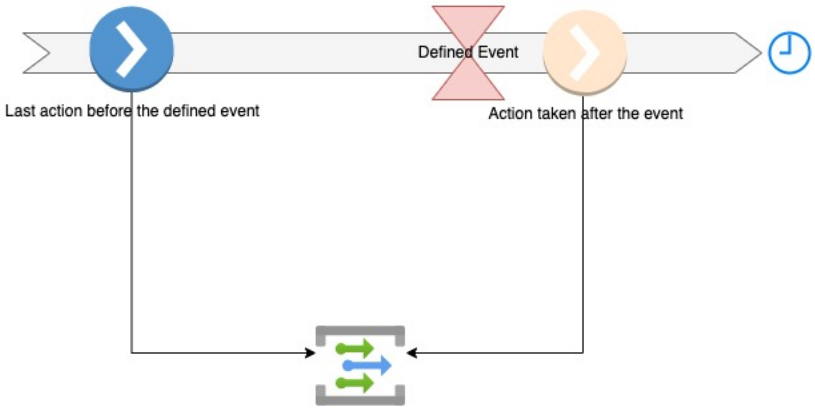
Fault Tolerance: Given their distributed and asynchronous nature, reactive engines often incorporate mechanisms for fault tolerance and resilience. This includes features such as redundancy, error handling, and automatic recovery to ensure continued operation in the face of failures or disruptions.

Reactive Programming: Reactive engines may leverage reactive programming paradigms and frameworks, which provide abstractions and tools for handling asynchronous events and data streams. Reactive programming enables developers to express complex event-driven logic concisely and manage asynchronous interactions effectively.

Implementation Approach

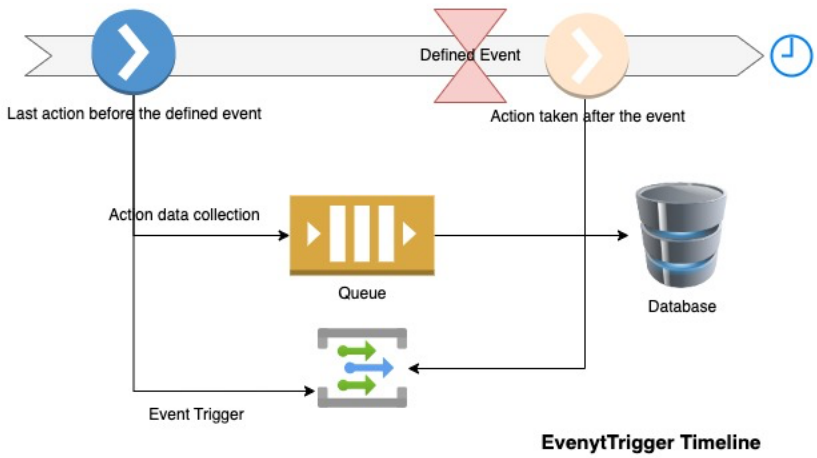
EventTrigger Engine

The EventTrigger Engine is designed to achieve two objectives. To begin, it is important to gather the list of events that have been defined. The second task is to produce the pre-event action and the post-event reaction of the defined event. If we look at an instance of time, we can visualize the engine taking action data at three instances of time. The purpose of extracting data sets is to provide a complete and accurate reaction to a predefined trigger for machine earning modeling.



EvenytTrigger Timeline

This user's interaction data is taken from every user. If need be, a queue system can be implemented to manage data consolidation into a database engine. The queue system is only needed for insertion into a database. The database is not a new element. It is



EvenytTrigger Timeline

part of the Event Hub components. It is further detailed to show the differences when a queue system is implemented.

To design a queue system that allows time for inserting data into a database, you'll want to consider a few key components: the queue itself, the database, and the processes for inserting data into the database from the queue. Here's a basic outline of how you might structure such a system:

Queue Implementation: You can use a variety of queue implementations based on your specific requirements and technology stack. Common options include:

Message Queues: Such as RabbitMQ, Kafka, or Amazon SQS.

Task Queues: Utilizing Celery (with RabbitMQ or Redis as a broker) or similar frameworks.

Custom Queues: If your system requirements are straightforward, you might opt for a custom implementation using data structures like arrays or linked lists.

Database: Choose a suitable database system that aligns with your application's requirements. Common choices include relational databases like MySQL, PostgreSQL, or NoSQL databases like MongoDB, Cassandra, etc.

Insertion Process: Define the process for inserting data from the queue into the database. This process should handle cases where the database might be unavailable, perform any necessary data transformation or validation, and ensure efficient and reliable insertion.

ML Engine

To start, it's important to create a reliable ReactiveUI machine learning model. It is necessary to initiate data collection from the EventTrigger machine. To obtain an ML model with reliable accuracy, it is necessary to gather a desirable volume size.

After collecting enough data, the modeling exercises are required.

Feature Engineering:

Identify relevant features that can aid in the model's learning of patterns from the data.

If required, use techniques like polynomial features, feature extraction, or dimensionality reduction to create new features.

Model Selection:

Select the machine learning algorithm(s) that are most suitable for the problem type (e.g., classification, regression, clustering), data characteristics, and constraints (e.g., computational resources, interpretability).

Try out a variety of algorithms to find the one that performs the best for your problem.

Model Training:

Divide the data into training, validation, and test sets to gauge the model's effectiveness.

Train the selected model(s) using the training data. Fine-tune hyperparameters using techniques like cross-validation or grid search to optimize performance.

Model Evaluation:

Use the validation set to evaluate the performance and generalization ability of the trained model(s).

Use evaluation metrics that are appropriate for the problem type (such as accuracy, precision, recall, F1-score for classification, MSE, MAE for regression).

Model Optimization:

Improving the model's performance can be achieved by iterating on feature engineering, hyperparameter tuning, or testing different algorithms.

Take into account methods like ensemble learning, regularization, or model stacking to improve performance and robustness.

The output to the above process is a machine learning model. However, this is the first phase of the development of the model. The next phase is to deploy a continuous machine learning model what is produce as output. Additional steps are required.

Here's an outline of how continuous machine learning typically works:

Initialize Model: Start with an initial model trained on an initial dataset or with random parameters.

Receive Data: Continuously receive new data instances or batches of data from a streaming source.

Update Model: Incrementally update the model parameters based on the new data without retraining the entire model from scratch. This can be achieved through techniques such as:

Online Learning: Update the model in real-time as new data arrives, adjusting model parameters to minimize a chosen loss function.

Mini-batch Learning: Periodically update the model using mini-batches of data, striking a balance between real-time updates and computational efficiency.

Stochastic Gradient Descent (SGD) or variants like mini-batch SGD or adaptive learning rate methods.

Model Evaluation: Periodically evaluate the performance of the updated model using a validation dataset or by monitoring its performance in a real-world environment.

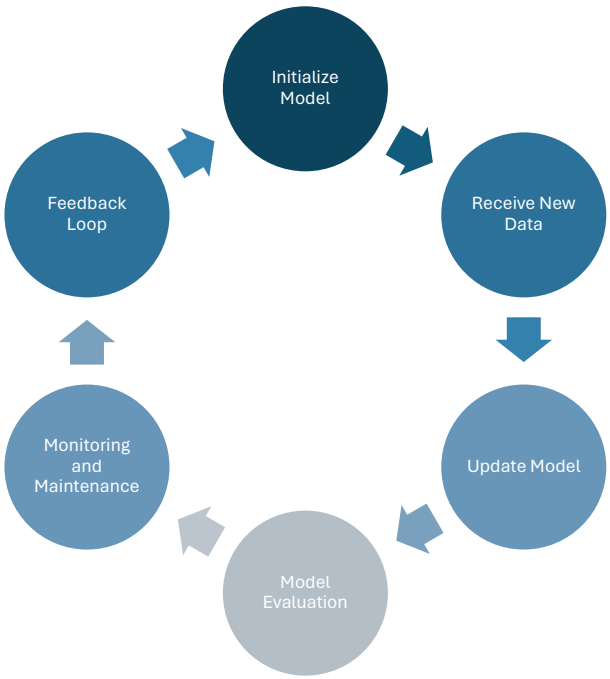
Model Persistence: Persist the updated model parameters to ensure that the model state is maintained across sessions or deployments.

Model Adaptation: Adapt the model architecture or hyperparameters over time to accommodate changes in the data distribution or requirements. This may involve techniques like model selection, hyperparameter tuning, or model ensembling.

Monitoring and Maintenance: Continuously monitor the model's performance and behavior and intervene if performance degradation or concept drift is detected. This may involve retraining the model on historical data or adjusting model parameters.

Feedback Loop: Incorporate feedback from users or domain experts to improve the model's performance or address specific challenges.

The new loop process can be presented as the following diagram.



Continuous machine learning is commonly used in various applications such as fraud detection, recommendation systems, predictive maintenance, and anomaly detection, where the underlying data distribution may evolve over time, and it's essential to adapt the model to these changes without discarding previously learned knowledge.

Reactive Engine

The reactive engine is an inference ml engine that will provide the possible actions from where the user monitored events is.

A machine learning (ML) inference engine is a crucial component of many modern ML systems, responsible for making predictions or decisions based on trained models. It's essentially the operational phase of the ML process, where the model is deployed and utilized to analyze new data and generate insights or actions.

An ML inference engine plays a critical role in operationalizing ML models, enabling them to make real-time predictions or decisions in production environments. Its design and implementation require careful consideration of factors such as performance, scalability, security, and integration with existing systems.

UX Changes

ON the UX changes, it is only limited to identification of every monitored event, a modal containing recommended action to the users.

Backend Macro Services Implementation

The macro services provide UI and API to the ML model and the DB recommended actions. Designing microservices for an inference engine involves breaking down the inference process into smaller, more manageable components that can be independently developed, deployed, and scaled. Here's a conceptual approach to designing microservices for an inference engine:

Model Service:

- This microservice is responsible for loading and managing ML models.
- It handles tasks such as model loading, initialization, and versioning.
- Provides APIs for making predictions using loaded models.

Feature Engineering Service:

- Handles feature preprocessing tasks required for model inference.
- This service may include functionalities for feature scaling, encoding categorical variables, or handling missing data.
- Provides APIs for transforming raw input data into features suitable for model inference.

Scoring Service:

- Performs the actual inference by applying ML models to input data.
- Utilizes the loaded models from the Model Service and the preprocessed features from the Feature Engineering Service.
- Provides APIs for making predictions on new data and returning the inference results.

Post-processing Service:

- Performs post-processing tasks on the inference results.
- May include tasks such as formatting the output, applying business logic rules, or filtering irrelevant predictions.
- Provides APIs for delivering the final processed inference results to downstream applications.

Monitoring and Logging Service:

- Monitors the performance of the inference engine in real-time.

- Logs relevant metrics and events for performance analysis, debugging, and auditing purposes.
- Provides APIs for accessing monitoring data and logs.

Feedback Loop Service:

- Collects feedback on the quality and accuracy of inference results.
- Incorporates feedback into the inference pipeline for continuous improvement of models or preprocessing techniques.
- Provides APIs for submitting feedback and triggering retraining or updates to the inference engine.

Model Deployment Service:

- Handles the deployment of ML models into production environments.
- Provides endpoints or APIs for serving predictions.
- Manages model lifecycle, including versioning, updating, and rollback.

Security and Authentication Service:

- Ensures the security and integrity of the inference engine and its components.
- Implements authentication and authorization mechanisms to control access to inference services and data.
- Provides APIs for user authentication and authorization checks.

Scalability and Resource Management Service:

- Manages computing resources required for inference tasks.
- Automatically scales resources based on workload demands.
- Optimizes resource allocation to maximize efficiency and cost-effectiveness.

Orchestration and Workflow Management Service:

- Orchestrates the execution of various microservices within the inference pipeline.
- Handles dependencies between services and ensures smooth workflow execution.
- Utilizes tools like workflow engines or Kubernetes for orchestrating microservices.

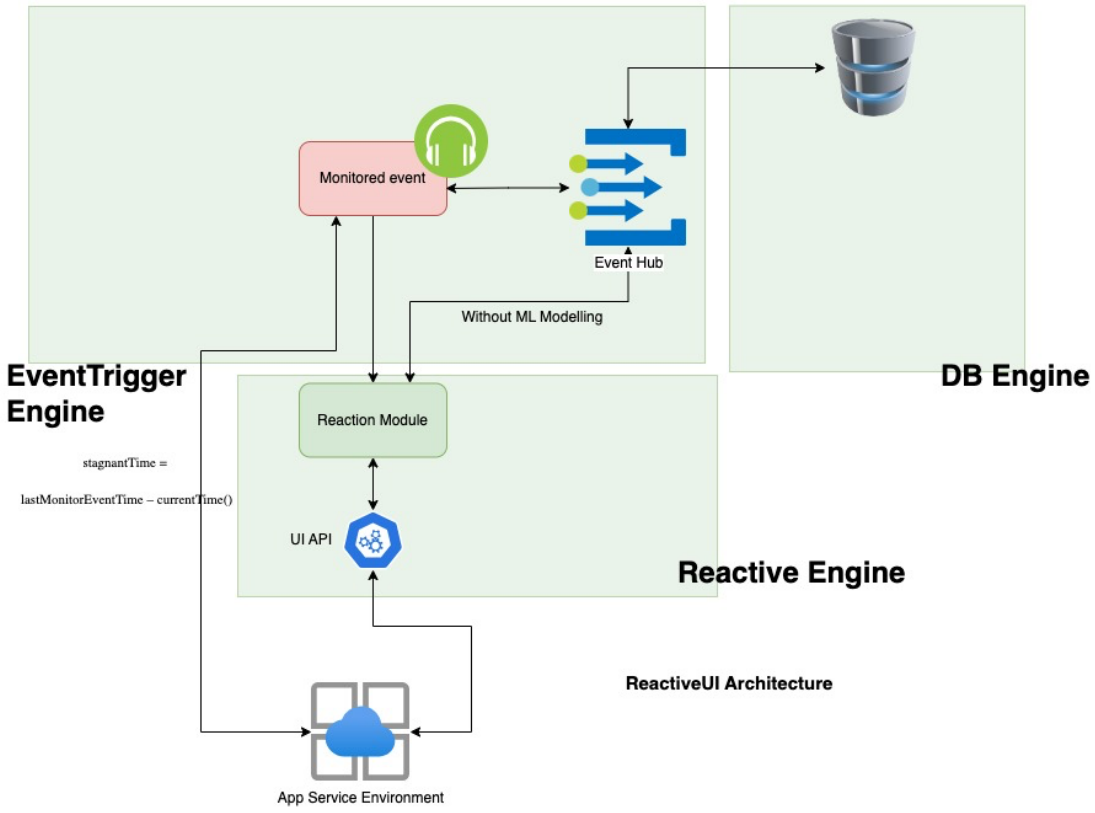
Database Requirements

The requirements for DB are straightforward. For full-scale implementation, there is a need to store pre-monitored and post-monitored events. This is connected to every event that is monitored. A continuous update from users' actions is needed. A SQL table may not be the best option for tracking such things.

Another database requirement is to keep track of all monitored events and provide a reference to the ML engine model inference identification.

That's all we need on the database side.

Simplification for SAFWA E-Tender Version 4.0 Implementation



Now comes the challenging part. How can we simplify the full-scale solution architecture above? Our objective is to demonstrate the reactive engine that can be activated by a set of monitored events.

First, we remove the complexity of processing and building the machine learning model. This does not diminish the importance of the model. Data is necessary to construct the initial stage of the reactive engine. For this scope, we build a reactive engine solely based on a defined monitored event.

A set of action items stored in the database will be used by the reactive engine to infer.

The next step is to create a list of events that are monitored and action plans for responding to them. All of these contents are to be stored within a connected database system.

The final stage involves intercepting monitored events and requesting a response through APIs for 'infer actions'.

Conclusion

Given that Safwa has requested that we carry out the third upgrade for SAFWA E-Tender, it is advisable to develop a simpler model of ReactiveUI during this project. The proposed approach is to initiate simpler implementation for the SAFWA E-Tender Sub-Con app. This will provide a taste of the implementation. Furthermore, onboarding cost is the highest for the 5000+ users within the Safwa contractor eco-system.