# AXP

Contents

[AXP 1](#_Toc195449889)

[Technology Stack 1](#_Toc195449890)

[AXP Design 1](#_Toc195449891)

[Migrating On-prem Data to cloud data 4](#_Toc195449892)

[Resiliency 7](#_Toc195449893)

[Server Restarted 8](#_Toc195449894)

[Client Restarted 8](#_Toc195449895)

[CM OR SMGR Down 9](#_Toc195449896)

[Flow Control 10](#_Toc195449897)

[Incremental Sync 10](#_Toc195449898)

[AXP Admin followed Design Patterns 13](#_Toc195449899)

[Challenges in AXP Admin 14](#_Toc195449900)

# Technology Stack

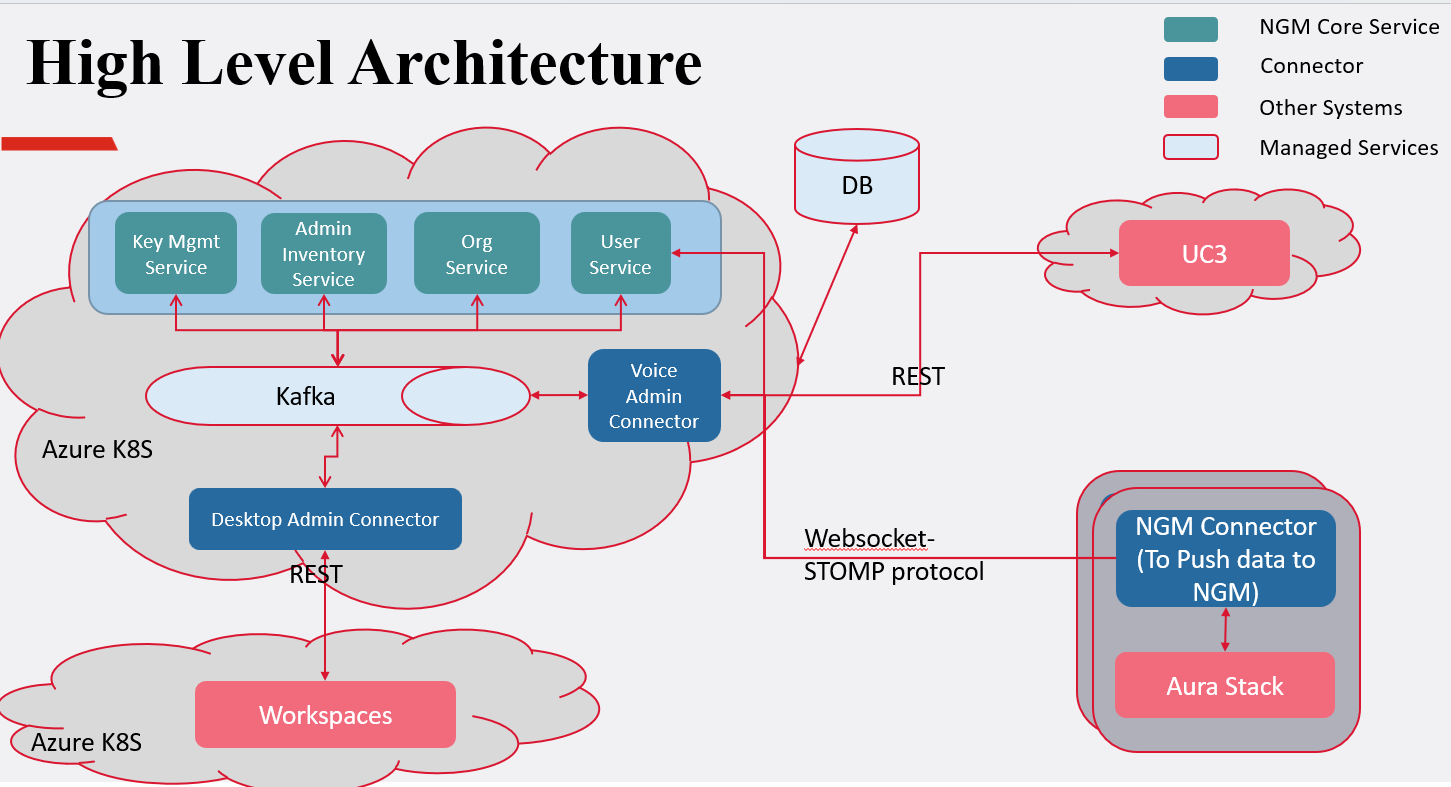
Java - Backend

ReactJS - Frontend

Postgrace DB

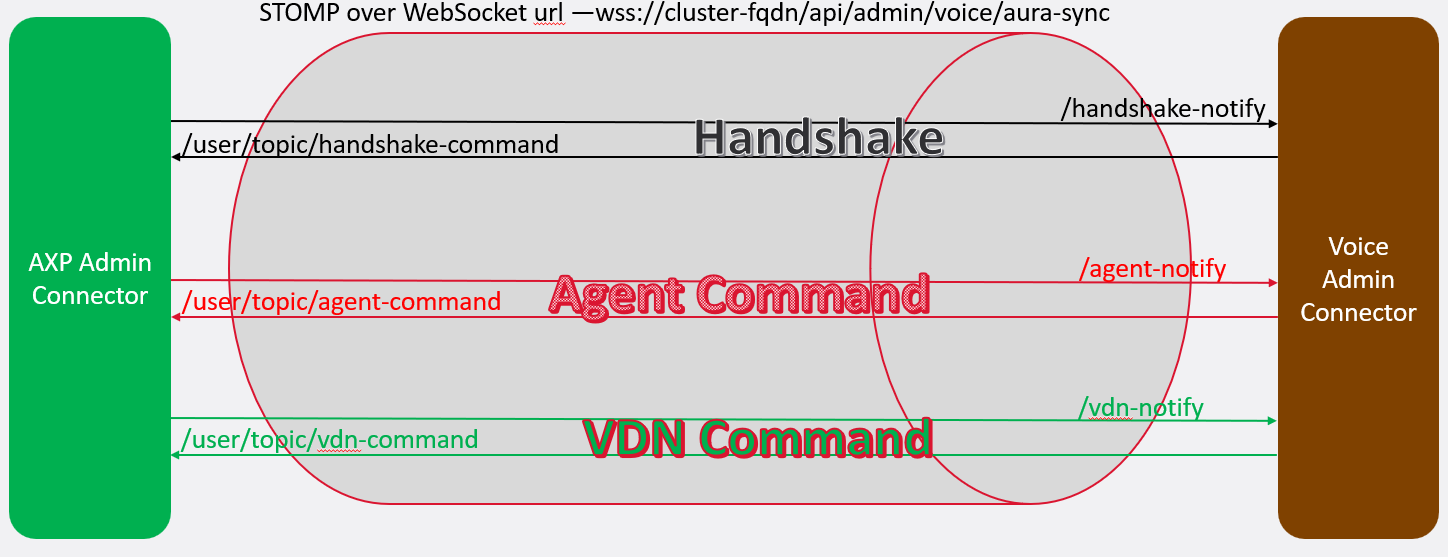
Go Lang-Connector

# AXP Design



**STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol**

|  |  |  |
| --- | --- | --- |
|  | **HTTP** | **WebSockets** |
| Communication | One-way, request-response | Two-way, full-duplex |
| Connection | Creates a new connection for each request | Maintains a persistent connection |
| Use cases | Good for serving static content | Good for real-time applications |



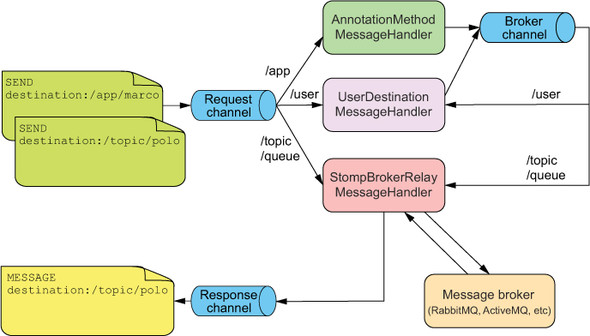
It’s a lightweight messaging protocol (like HTTP for messaging) that works over WebSockets or TCP.

it as a **standardized format** for messaging between clients and brokers.

While **WebSocket** provides a full-duplex communication channel, it doesn’t define any structure or messaging format. That’s where **STOMP** comes in.

STOMP gives:

* **Message structure** (e.g., headers, destinations)
* **Subscribe/publish** semantics (like pub/sub)
* **Interoperability** with message brokers (e.g., RabbitMQ, ActiveMQ)



**Example with Spring Boot**

**WebSocketConfig.java**

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/ws").withSockJS();

}

@Override

public void configureMessageBroker(MessageBrokerRegistry config) {

config.enableSimpleBroker("/topic"); // Server can publish to /topic/\*

config.setApplicationDestinationPrefixes("/app"); // Client sends to /app/\*

}

}

**Client sends message:**

stompClient.send("/app/chat", {}, JSON.stringify(

{ message: "Hi!" })

);

**Server method to receive:**

@MessageMapping("/chat")

@SendTo("/topic/messages")

public String handleMessage(String message) {

return "Received: " + message;

}

# Migrating On-prem Data to cloud data

Voice-service-: 60 +14

User-service -: 60

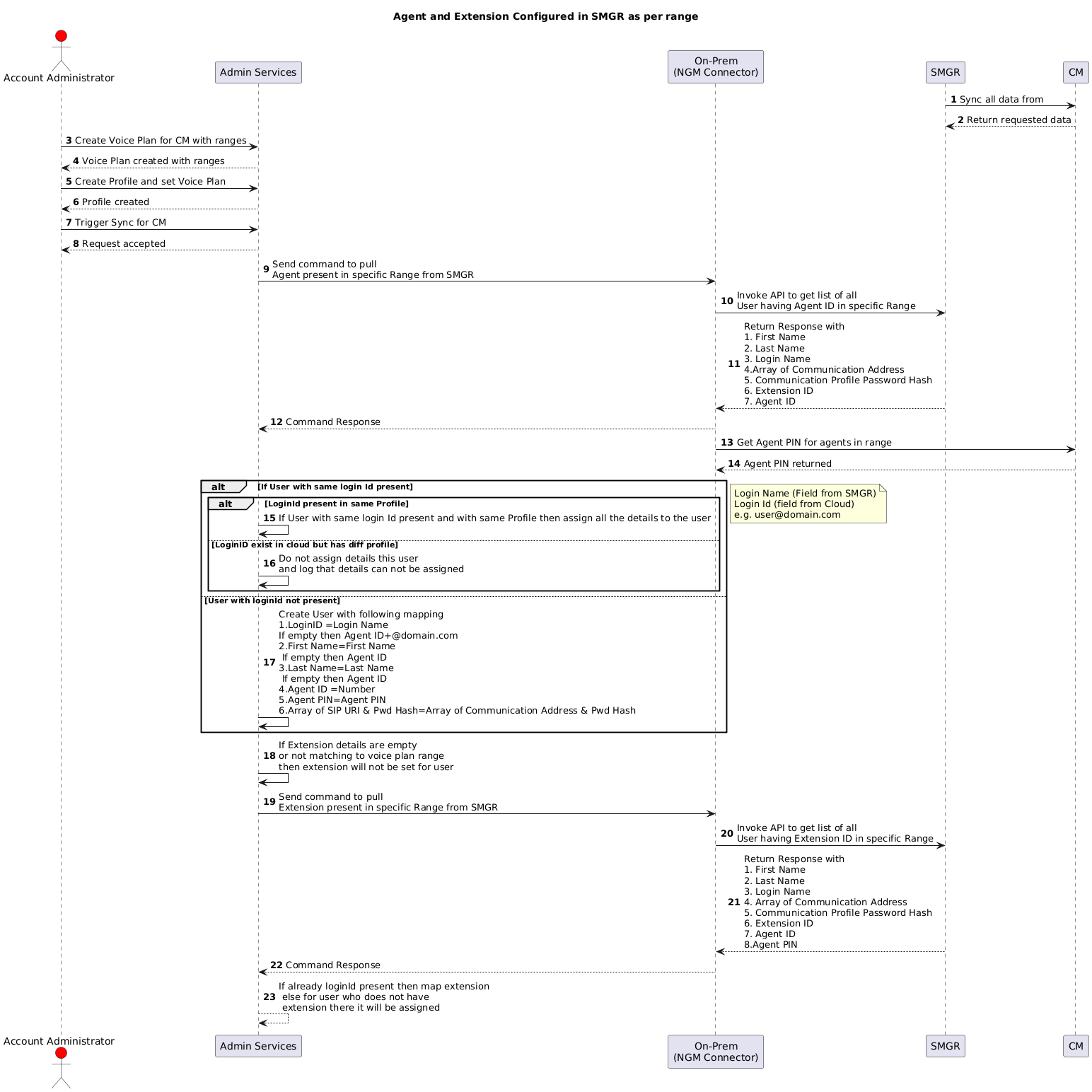
Admin-inventory & group –: 40

Org-service -: 40

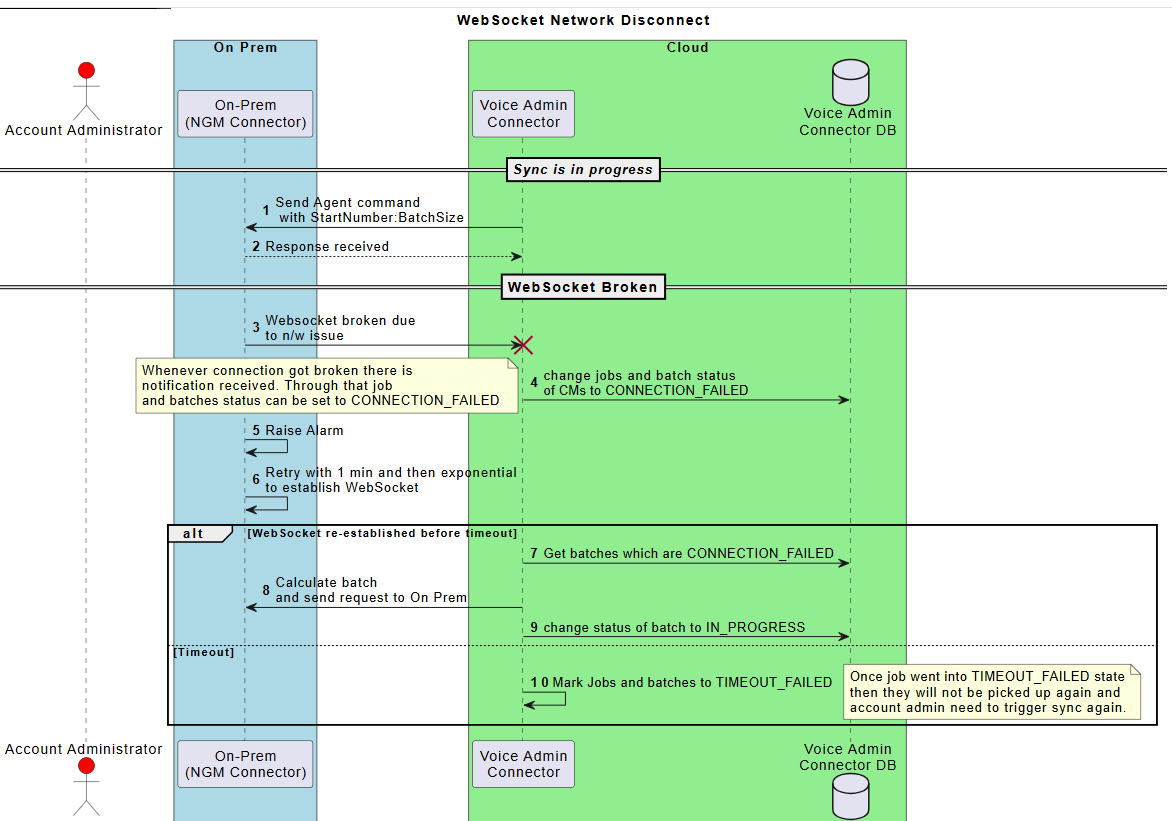
Total around => 220 tables

**Design Consideration**

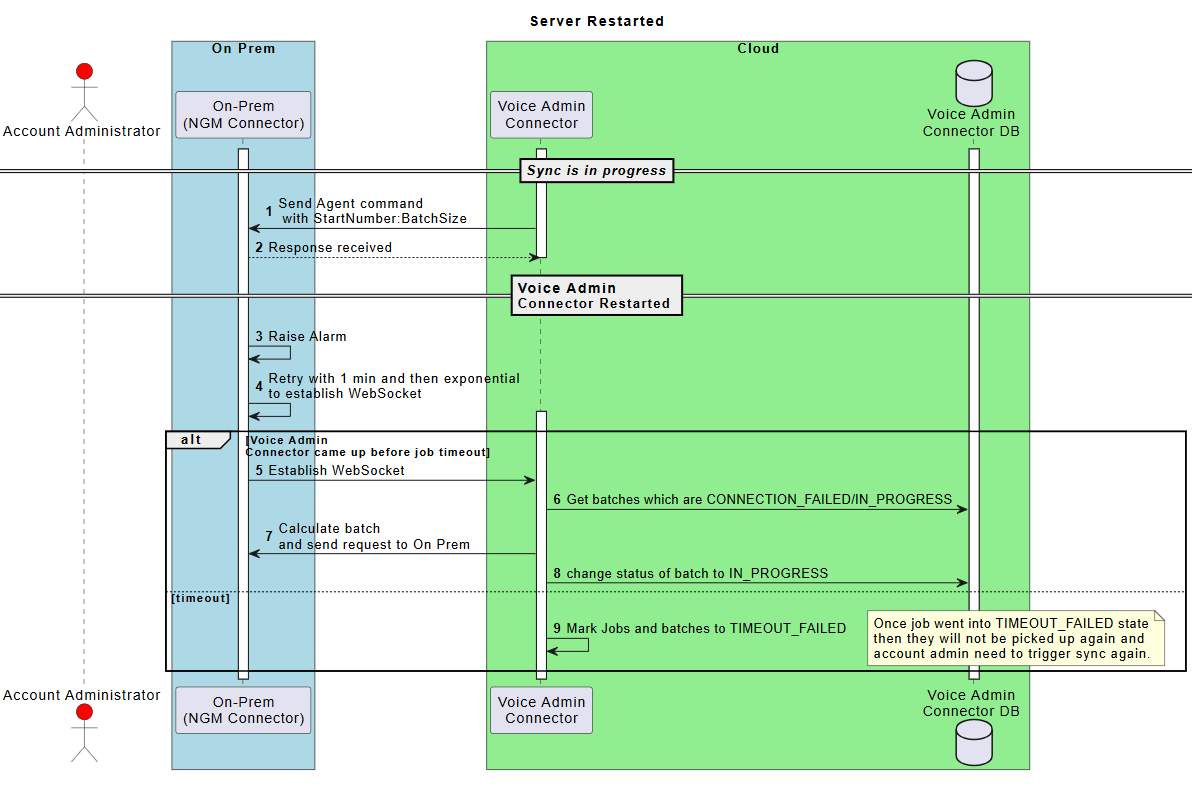
1. Minimal changes at on-prem side
2. Backward compatibility for message types
3. Resiliency
4. Duplicate data handling
5. In handshake message client used to send list of CM it is connected. Which will be helpful during sync resillency



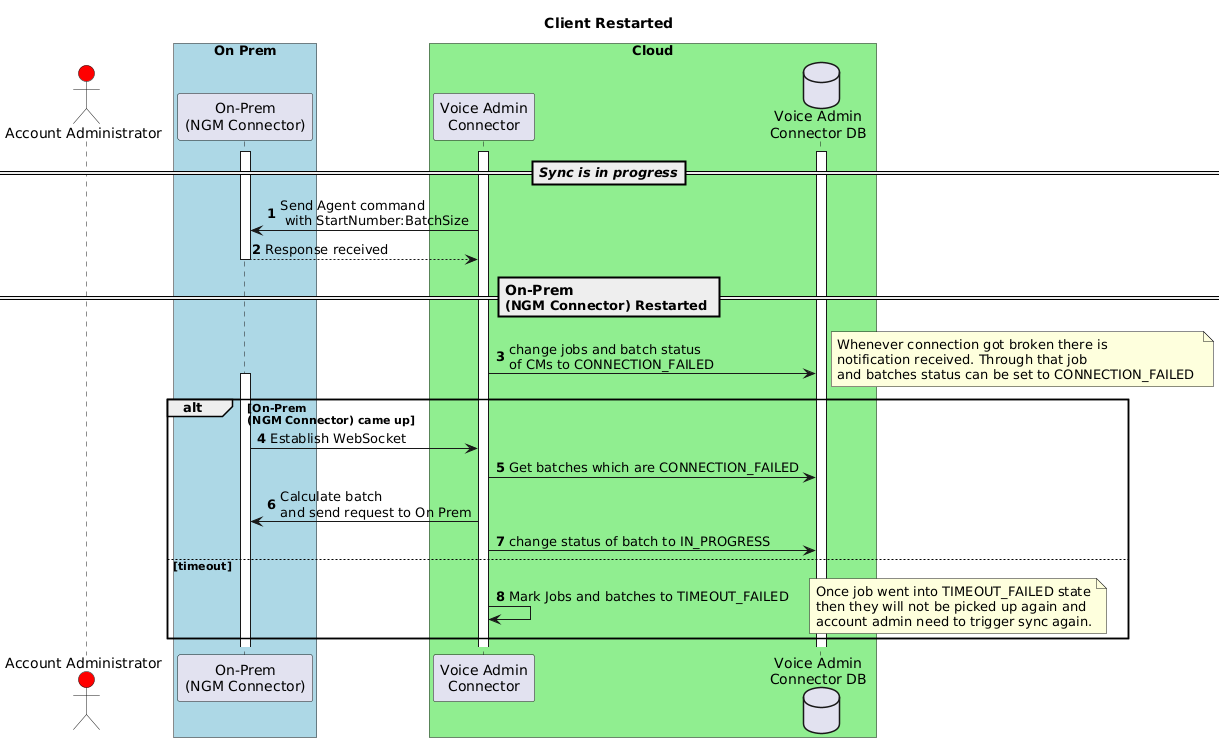
## Resiliency



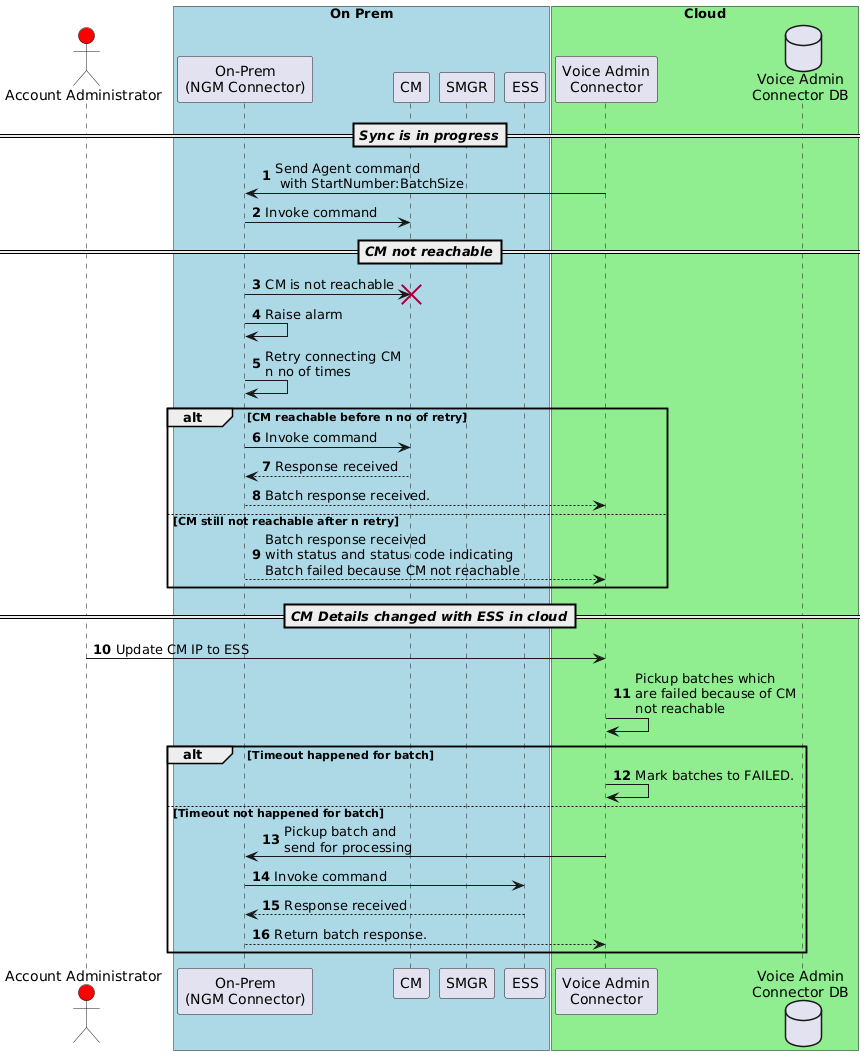
## Server Restarted



## Client Restarted



## CM OR SMGR Down



## Flow Control

**1. Client Side**

Server in cloud is going to make sure not more than x no of requests will be sent to client in on-prem. We are starting with single. Meaning at a time per entity only 1 request is going to sent over websocket. Once response is received then only next command will be sent

**2. Server Side**

Client is going to process the command which is received so there are no situation where client keep on sending same message again and again. Even if such situation arises then Server will detect based on response received (as this data is present in Server DB) ignores that message and send probably shutdown command.

## Incremental Sync

**High-Level Steps :**  
1. Receive Command for list History.  
2. Parse the command.    
3. From command payload, get Time-Stamp details. Get CM local time and then calculate the time difference. Get Start-Time and End-time from command and convert to CM time with the help of time difference calculated.. (Not able to get CM time zone, send failure). (Validation of start time and end time)  
4. Run the algorithm for figuring out point-in time details for all list history commands.   
5. Fire list history commands to CM and accumulate data.   
6. Accumulate and send details in single response only.

#1 : If response is received and corresponding list history message is in Time-out or failed state then don't process the response.

#2 : We will purge based on time or count for list history data in DB.

#3 : New entry should  be created for list history after marking the previous SENT command as Timeout\_failed.

#4. If some entities are not received due to size overflow - send the status in response as not Not\_Processed and persist same for entities.

#5. moreDataPresent value being true to be taken in second phase.

#6 Adding extra column for incremental sync status.

#7 Any incremental sync job will be consider as failed if any of its batch record will have rangeToProgess as non-empty.

#8 While creating incremental sync job, if previous inc job got failed (based on #7). Consider failed job time for next list history command.

#9 Response batch will always have entries for all entities irrespective of response received or not. in case of failure start time will retained as is from.

#10. Add transactional blocks in sequence diag

DB Schema

inc\_sync\_status

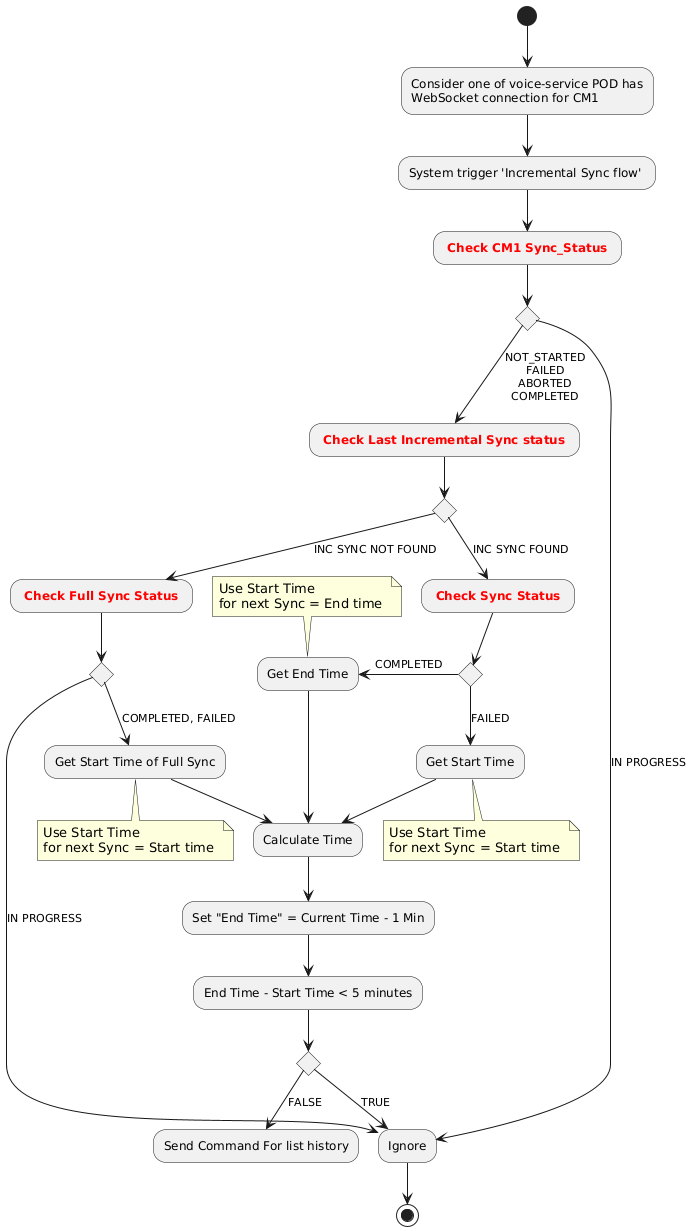
This will hold current request of "list history" which need to be sent.

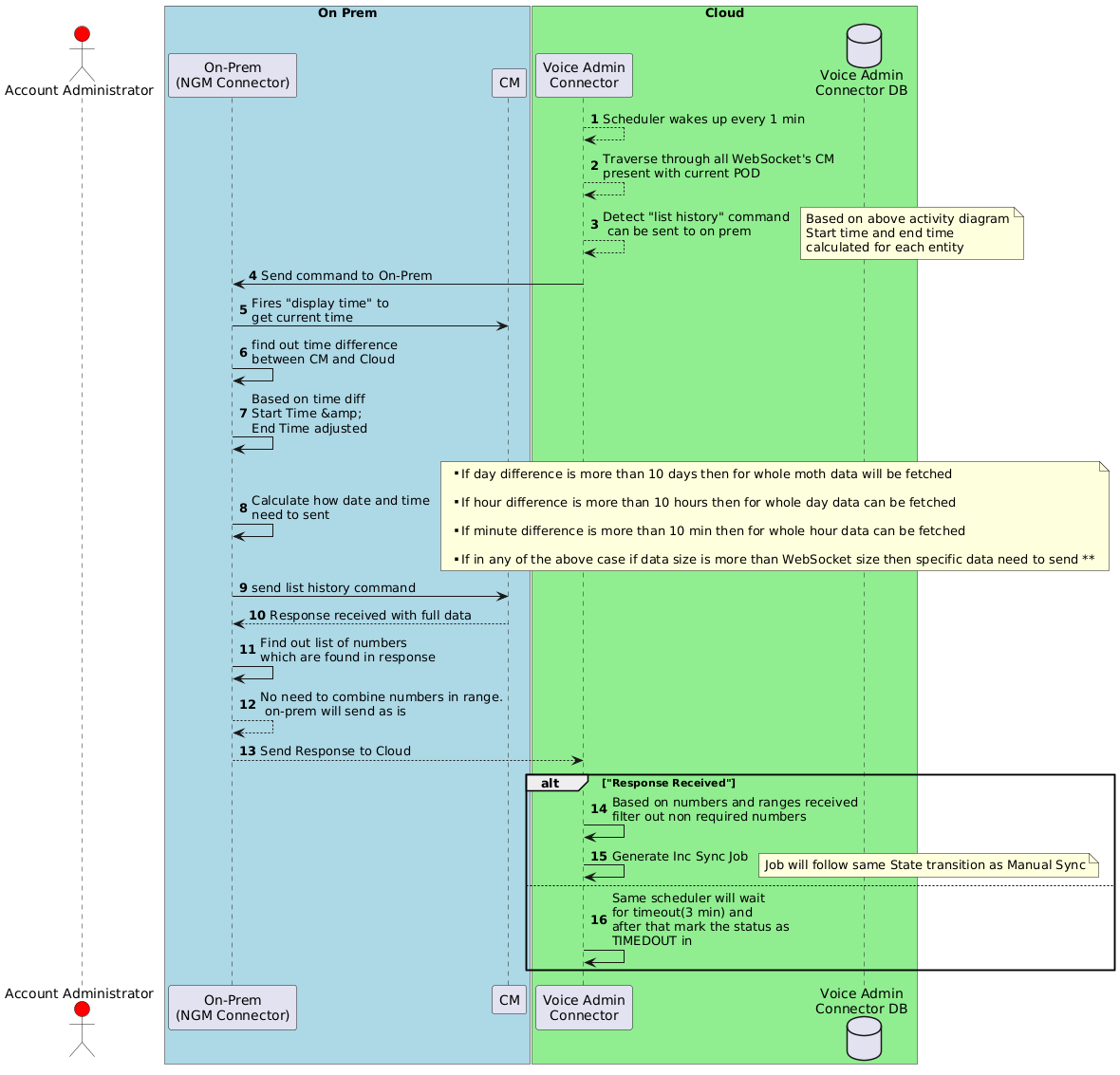
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Id (same as messageId)**  **PK** | **cm\_id**  **FK** | **status** | **job\_id** | **created\_at** | **updated\_at** |
|  |  |  | Once data is available then this can be filled with jobid. | TIMESTAMP | TIMESTAMP |

inc\_sync\_entity\_status

Once response is received and job is created then each of entity entry will be RESPONSE\_RECEIVED. If response not received in specific time as per above flow then this will be marked as TIMEDOUT and if WebSocket is still there then only add new set of entries

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **id**  **PK** | **inc\_sync\_status\_id**  **FK of above table** | **entity** | **start\_time** | **end\_time** | **response** | **status** |
| This will be unique ID of type uuid. | This will be same for multiple entries of Entity hence this can not be Primary Key. | AGENT  EXTENSION  REASON\_CODE  SYSTEM\_PARAMETER\_FEATURE | TIMESTAMP | TIMESTAMP | TEXT | SENT  RESPONSE\_RECEIVED  TIMEDOUT  FAILED  ABORTED |





# AXP Admin followed Design Patterns

1. **API Gateway pattern** -: It used ISTIO ingress gateway for all request to flow in…. Using Virtualservices and destination rules config of K8S requests were routed to specific service. There were plans to use other API gateways like APIgee for rate limiting, metering etc..
2. **Database Per Service Pattern** -: Different services has database per service.. Some of other services were following NoSQL DB.
3. **Service Discovery pattern** -: Using K8S services were able to discover each other through service name
4. **CQRS** -: Command Query Responsibility Segregation (CQRS) is a software design pattern that separates read and write operations in a system. Followed upto some extent using GlobalKTable but for other it was not there due to which load on DB was increased. Planning to use caching mechanism like Redis cache
5. **SAGA Pattern** -: Until user is not created and later until Voice data not available DAC was not triggering it’s update in Workspaces. Transactions were not getting rolledback but we used to maintain status

# Challenges in AXP Admin

1. Due to limited DB connection and limited partition in Kafka topic scalability was not achieved.   
   Solution -: Read must have been solved using CQRS pattern by using caching in Redis. Kafka topics partitions could be more.
2. Many number of topics for consumer of data and added complexity in DAC..   
   Solution -: This could have been solved by simplifying number of topics to single as UC reads data from admin directly
3. Bulk user has design issues like blocking thread due to which there was limit on number of bulk user processing otherwise they were getting stuck  
   **Issue**-: API accepts user data in xls. And publish batch of 1000 users on kafka. Then consumed by user-service and publish it on user-feature there each thread was keep on waiting for all users to get completed success failed there threads were getting stuck…  
   **Solution** -: Redesign with basic single thread which will wakeup every one minute to mark the progress.
4. Rate limiting was not there due to which in prod some automation was impacting performance.

Architecture problem you solved

1. RBAC Design  
   Lot of discussion and approaches proposed.
2. CPaaS Int…  
   BYOC-H support along with multiple CPaaS server support.