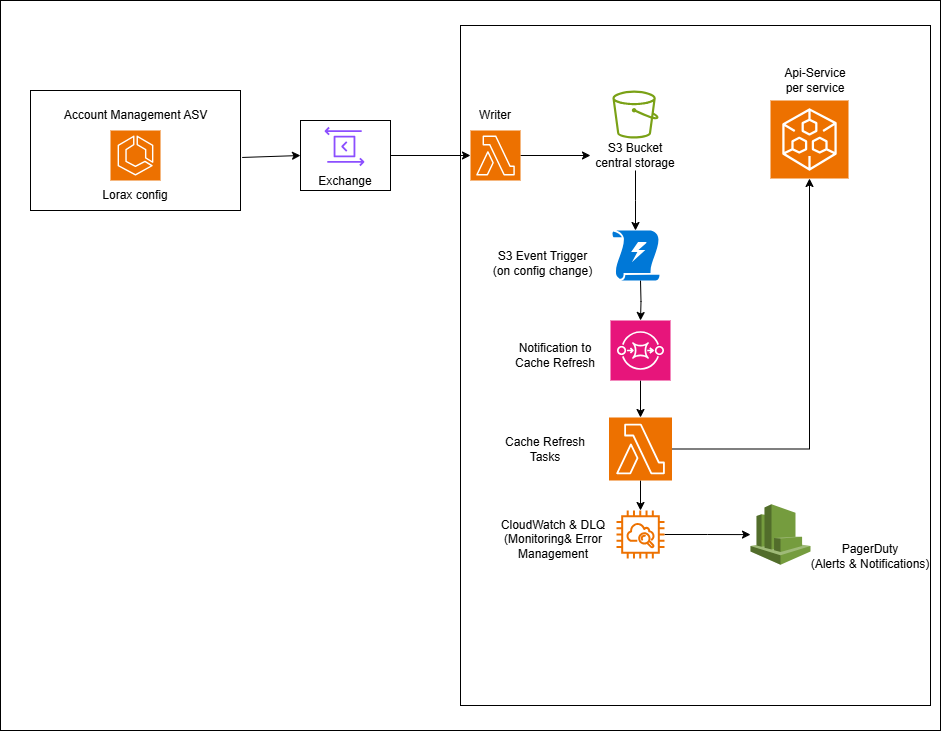
#### **Problem Statement**

The DocDB Reference database serves as a monolithic datastore relied upon across Loyalty ASVs for configuration data. However, this reliance on a single datastore creates risks, especially in terms of data resiliency and compliance with Enterprise standards. To address these issues, there is a need to transition this data into the target ASVs, ensuring it aligns with resiliency goals.

Currently, Level256 authors configuration data through the Lorax Config Tool, which is then written to the DocDB Reference. Consuming applications, including those across different ASVs and AWS accounts, connect directly to this database. This approach not only introduces potential delays but also raises concerns about cross-account dependencies and compliance with evolving standards. Therefore, a solution is required to decentralize the data storage and improve overall system resilience and compliance.

#### **Proposed Solution: S3 Event-Based Cache Refresh**

In this proposed solution, we will use Amazon S3 for centralized storage of configuration data, combined with an event-driven architecture to ensure that all services can refresh their cache with the latest configurations in near real-time.



#### 

#### **Flow:**

1. **Lorax Config (cof-card-dev)** initiates the configuration process by sending data to the Exchange.
2. **Exchange** forwards the data to the **S3 Bucket (Central Storage)** for centralized storage.
3. **S3 Event Trigger** detects changes in the S3 Bucket and sends notifications.
4. **SQS Queue** receives these notifications and prepares for cache refresh tasks.
5. **Lambda (Cache Refresh)** processes the notification and triggers cache updates in the API services.
6. **API-Service (Per Service)** fetches the updated configuration from the S3 Bucket and updates its cache.
7. **CloudWatch & DLQ** monitors the process and handles any errors, ensuring system .reliability.
8. **PagerDuty** alerts the responsible team if any critical issues arise, ensuring prompt action.

| **Pros** | **Cons** |
| --- | --- |
| **Centralized Configuration Management:** Central storage of configurations in S3 ensures a single source of truth. | **Network Latency:** The system relies on network connectivity, which could introduce latency when fetching data from S3. |
| **Scalability:** The architecture is highly scalable, leveraging AWS services that automatically scale with demand. | **Complexity:** Implementing and managing this architecture introduces complexity, particularly in ensuring all components communicate correctly and efficiently. |
| **Real-Time Updates:** S3 event triggers allow for near real-time updates to service caches, ensuring all services work with the latest data. | **Potential Inconsistency:** If a service fails to update its cache, there may be temporary inconsistencies across services until the issue is resolved. |
| **Cost-Effective:** AWS’s pay-as-you-go model helps manage costs effectively, only charging for actual usage. | **Cost Accumulation:** Although cost-effective, frequent updates and notifications can lead to increased costs, especially if the system scales significantly. |