INTRODUCTION

This case is based on ***APIs*** and how our product MTD (Digital Transactional Engine) can develop and analyze through ***APIs*** in addition to other AWS services. MTD is a flexible, secure and transactional workflow that supports high volumes of complex operations (high transactionality) that involve the orchestration of multiple systems.

MTD is a flexible, secure and transactional workflow that supports high volumes of complex operations (high transactionality) that involve the orchestration of multiple systems.

As an additional definition, MTD allow:

- It allows to execute long-running flows (feasible to integrate with human tasks, through APIs) and short-running typically used for orchestration of complex processes.
- It provides an information model that facilitates the traceability and audit of historical processes, as well as online monitoring to have complete control of its processes.
- To facilitate the definition of the processes, it allows to implement the flows based on definitions made in Bizagui Modeler (Free Workflow Construction IDE).
- This solution can be acquired as a server license or in SaaS mode, allowing elasticity and rapid deployment of the processes.

During the development of this SDP, will be able to know the operation and the different components that the MTD possesses as such and how it is related to the different AWS instances and services.

In respect the problems that our solution comes to solve are relates to the following:

- Online control of orders in process.
- Manage orders with exception.
- Process all orders.
- Comply with dispatch times.
- Early warning.
- Timely information.
- Purchase order statistics.
- Capacity against high demand.
- Avoid crafts.
- Reduce operating costs.

For the business, MTD solves the life cycle process of a purchase order, handling communication and integration between the different legacy systems existing in Customer (Gift card, EOM, Paperless, Sales Support) and customer-side interactions, such as the generation of purchase tickets / invoices and the subsequent sending by email, and their operation and traceability is available to Customer executives through a BackOffice Web portal.

Being a cloud-mounted solution, MTD is a highly scalable platform which provides a high level of availability and adaptation to the operating requirements that Customer requires. Our proposal is based on a solution that allows to collect, manage and orchestrate purchase orders or transactions incorporating business rules, monitoring elements and others. An example of its implementation is in the management of Retail Purchase Orders where it manages its flow considering business rules, state management, exception control, mailings and integration with the different legacies among other things. This solution is our Digital Transactional Engine (MTD), mounted on an AWS environment.

For this solution, the services used by AWS Cloud are ***API Gateway*** (among others).

Its logical operation is developed as follows:

When entering a purchase order, it is stored in an S3 bucket in xml format. This activates a lambda that transforms the order to json format and sends the command to a SQS queue and this SQS sends the command to a queue reader. At this point there is a condition that, if the queue reader does not read the order in 10 attempts, it goes to an SQS where the order will be pending. In case the reader correctly reads the order, it will go to an API connected to the database engine.

When the order arrives at the database, it is verified in what state the order is and depending on this, it will be redirected to the corresponding API with the status of the order, launching a lambda and saving the order in the database.

In the event that an order fails when trying to enter the state of broadcast_dte, it will be sent to a step function that will trigger two lambdas where the error will be controlled.

To control errors that occur in the remaining stages, these will be directed to an API that controls these errors by triggering the corresponding lambda by storing and updating the order in the database.

For the second stage of handling the order status, the SES mail service is used, which, at the time of the order, passes through the issuance_dte successfully and is updated in the database, sends a legacy through of an email to the customer indicating that the order began to be processed.

The entire workflow is controlled with CloudWatch metrics and alarms for the different services indicated in the scheme.

AWS APIs VALIDATION

Implementation of REST-type ***APIs*** for the messaging flow between system components and lambda functions. The message format is JSON, through HTTPS secure protocol.

For the expected responses from the APIs, the requests that the client received in its high-flow period before implementing the AWS services were considered. By supplying AWS services, requests averaged between 6 and 8 per second which equals 500 orders per minute.

The CloudWatch metrics used for this solution are the number of calls the API receives, latency, integration latency, 4xx error, and 5xx error.

The implemented APIs are based on HTTP protocol with a regional endpoint, this because the calls that are made are within the same region. The API connected to the database engine, is connected to an Edge endpoint, in addition to having different control metrics with CloudWatch.

The APIs are integrated with lambdas functions located in the us-west-2 region, while the API connected to the database engine, both the GET and POST action, is integrated with HTTP.

Using a usage plan and API clients, client needs were identified to identify performance requirements.

As indicated in the previous point, tests were carried out before implementing AWS services in a period of high order flow, in order to calculate the average number of orders received per second and with this, when incorporating AWS services, having the application estimate.

From this, the so-called "API clients" were generated where the accesses to each API and the amounts of incoming requests in a specific time interval were defined, thus restricting the activation of the APIs by unauthorized users and protecting the operation of the backend.

The deployment of the APIs in the different environments was managed through the creation of three stages (one for each environment).

In the development stage (DEV), developers self-manage the deployment of APIs according to their needs. In later stages, deployments are performed once all unit tests have been run, according to an internal quality process and the authorization of the release manager.

The next stage is QA, where a version approved by the development area is delivered to the client. In this stage the integration and stress tests necessary to move to productive environments are carried out.

In the final production stage, the APIs were deployed in the production stage.

Starting from the QA stage, it is verified with CloudWatch metrics and alarms, that the operation was as expected, emphasizing response times and safe accesses.

DETAILS ON AUTOMATING THE CREATION OR UPDATING AMAZON API GATEWAY

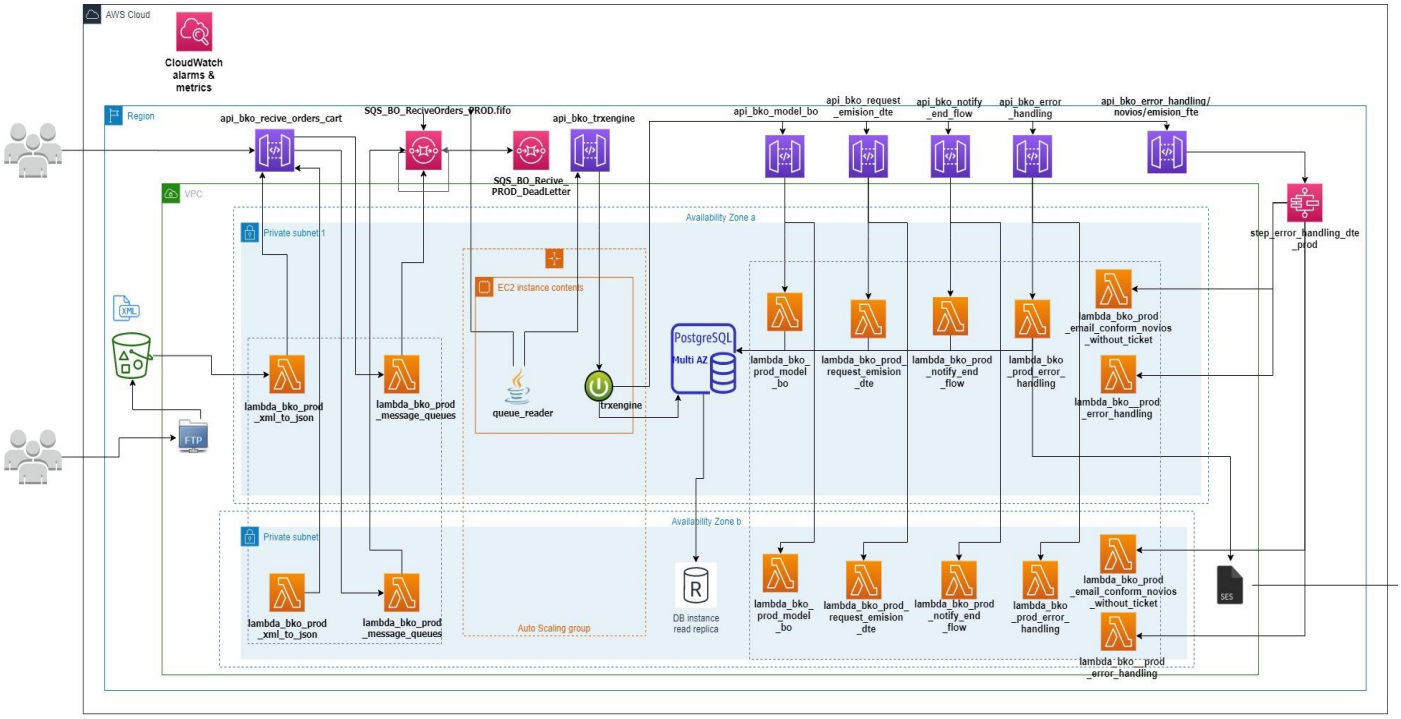
For our customers CloudFormation was used to perform updates or create API Gateway. For API invocation, it is controlled by the SAM template, who are authorized users to run the API, and who are not. It also automates events such as receiving SQS events by lambdas that connect to this. In addition, you also control the number of requests that you can receive in a defined time interval.

ACROSS ALL SUBMITTED CASE STUDIES SHOULD DEMONSTRATE PROFICIENCY HTTP Proxy Integrations
The HTTP proxy integration solution is used so that the methods that the client enters, whether it is POST, PUT, GET, DELETE, are sent directly to the application backend and in this way the same backend parses and returns a response. This allows the API Gateway to handle the authentications and the usage plan, in addition to being able to make mapping requests.

IAM Integration
For the security of AWS resources, each user was integrated with IAM roles specifying the policies of the services that they can see and execute. This controls who is authenticated and authorized to use the services.

AWS Service Proxy Integrations
APIs are built to be able to queue purchase orders in SQS queue. This API is triggered by a lambda function, and allows to deposit the message of the purchase orders in a SQS queue of the FIFO type directly.

DIAGRAMS.


AWS Cloud
CloudWatch alarms & metrics
Region
api_bko_recive_orders_cart
SQS_BO_ReciveOrders_PROD.fifo
SQS_BO_Recive_PROD_DeadLetter
api_bko_trxengine
api_bko_model_bo
api_bko_request_emision_dte
api_bko_notify_end_flow
api_bko_error_handling
api_bko_error_handling/novios/emision_fte
step_error_handling_dte_prod
VPC
Availability Zone a
Private subnet 1
EC2 instance contents
queue_reader
trxengine
PostgreSQL
Multi AZ
lambda_bko_prod_xml_to_json
lambda_bko_prod_message_queues
lambda_bko_prod_model_bo
lambda_bko_prod_request_emision_dte
lambda_bko_prod_notify_end_flow
lambda_bko_prod_error_handling
lambda_bko_prod_email_conform_novios_without_ticket
lambda_bko__prod_error_handling
XML
FTP
Availability Zone b
Private subnet
Auto Scaling group
DB instance read replica
SES