Generative AI Application Patterns with AWS

Volume 1

Intelligent data processing - Chatbot - Agentic workflow

> *Building generative AI application is not only about LLM choice and prompt engineering, but also about the well-architected cloud solution*

By reading this book I declare that any knowledge I specifically obtained from this book is to be used according to justice

Yudho Ahmad Diponegoro

PATTERN 1

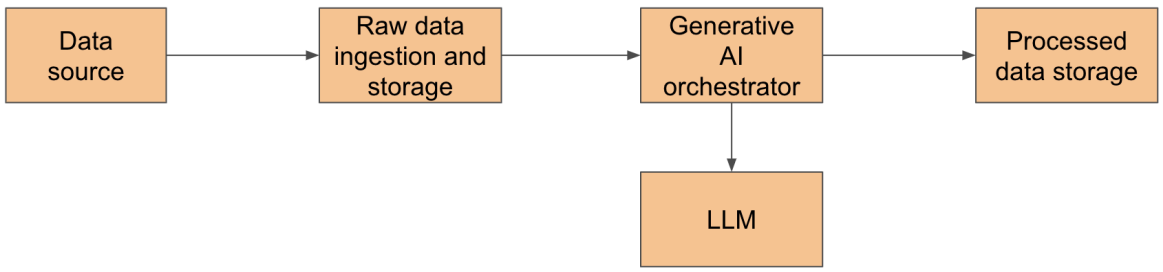
Intelligent Data Processing

Some applications involve heavy data ingestion with multiple possible sources, including but not limited to user's uploaded media files, Application Programming Interface (API) calls, webhooks, emails, clickstream, media streaming, and IoT data streaming. These data can be text, JSON data, images, videos, CSV file, PDF file, and much more.

Extracting meaningful information from these data can be crucial in improving customer experience, discovering important insights, or even supporting the core functionality of the application. In other cases, these data are used to produce output artifacts e.g., analysis of the inputted data. Generative AI can help to be the intelligence in automatically extracting the intended data from the raw data or in producing new artifacts.

While more traditional ways of extracting data such as query languages, field matching, or regular expression (regex) work for raw data that are semi-structured with certain formats, generative AI can leverage its intelligence to also understand the context of the data and extract it accordingly. For example, it can extract the intended product details and expected delivery date from a raw purchase order email.

A general architecture on how intelligent data processing can be implemented is depicted below.



The raw data is ingested from the data source into a storage which can be an object storage, a database, or even a queue or stream. The raw data is then processed by an orchestrator which runs on a compute technology such as a container, a function, or a virtual machine. This orchestrator can be scheduled to poll from the raw data storage. The orchestrator will leverage the intelligence of a large language model (LLM) which performs generative AI to

extract intended data or generate new artifacts from the raw data. The processed data output is then stored for further downstream processing.

The implementation examples of the workflow over several possible scenarios and requirements are detailed in the sections below.

Scenario 1: Asynchronous immediate document processing

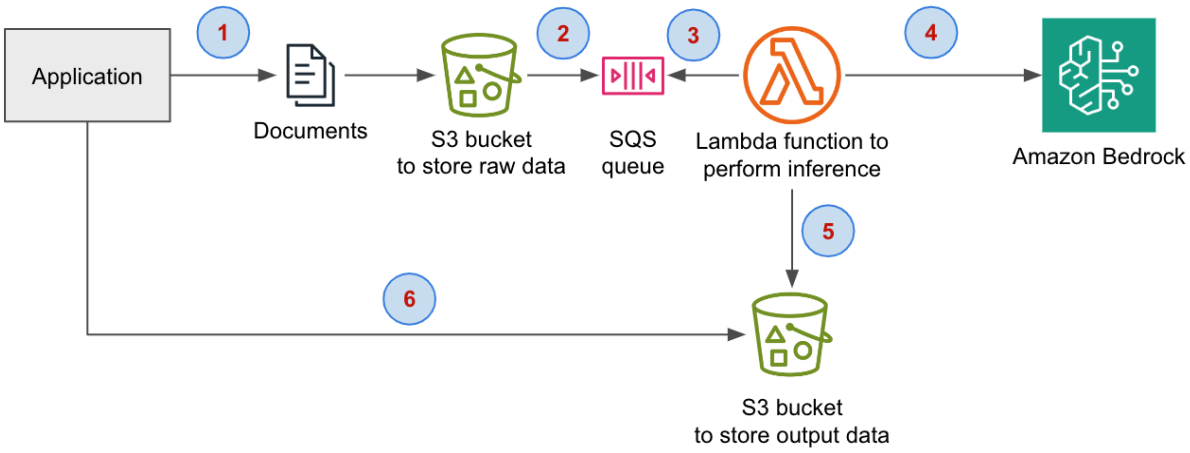
This scenario depicts the asynchronous, yet immediate, processing of documents being uploaded to an object storage such as Amazon Simple Storage Service (S3). An orchestrator will fetch the raw data, use generative AI to extract the intended data or generate new data, and then store the output.

While batch document processing will be addressed in a later scenario, this scenario focuses on the use case where each ingested data needs to be processed without much delay. We are talking about seconds or minutes of processing between an individual document being ingested until an output is generated.

These are the use case examples for this scenario:

1. Automated data extraction with these sub-use cases:
    a. Email data extraction, where incoming emails are triaged by the system which then extract information to a standard format for downstream tasks
    b. Purchase order (PO) data extraction, where potential buyers upload POs with their own format and the system will extract and normalize the data
2. Automated image & document verification with these sub-use cases:
    a. Product complaint triage system, where buyers upload photos of damaged delivered products and the system will do first triage to check for validity.
    b. Vehicle condition inspection & classification, where onsite inspectors upload car photos and the system helps classify and check the condition
    c. Document compliance checking, where end users upload documents to be checked against compliance before it is stored
3. Content moderation, such as video content moderation where the user uploaded videos will be checked first
4. Intelligent alerting system, where data, such as video frames or log is ingested and analyzed for potential issue that requires human's attention

A possible implementation architecture is depicted below.



The numbering in the diagram represents the processes with the following details.

1. An application uploads raw documents into an S3 bucket. The application can be a crawler, a backend of a user facing application, a scheduled job, or another type of application. The documents can be texts, images, videos, or files.
2. A queue in Amazon Simple Queue Service (SQS) is configured to receive a message when a document is uploaded into S3 through S3 event notification.
3. AWS Lambda is configured to poll the queue and invoke a Lambda function when there is a new message.
4. The Lambda function calls Amazon Bedrock, a fully managed service that offers a choice of foundation models (FMs), using API. The model will perform inference using the supplied input data and return an output.
5. Upon receiving the output response from Bedrock, the Lambda function uploads the output data into an S3 bucket.
6. The application consumes the output data from the S3 bucket.

> ⚠️ **Important**
>
> In this book, often the architecture diagrams are simplified for easier learning. They may not always contain all the required components and best practices within each diagram. Sometimes more components and best practices are added as you read further down.

While the diagram and steps explanation above describe the full process overview, let's dive deeper in the following subsections.

Documents upload, storage, and retrieval

When Amazon S3 is used as the object storage, you need to create the S3 bucket first. You can create the S3 bucket either using AWS Console, AWS SDK, AWS CLI, or even IaC (infrastructure as code) tools like AWS CloudFormation, AWS CDK, or Terraform.

A bucket is a container object. It resides in an AWS Region, a physical location around the world where AWS clusters their data centers, which you can pick to comply with data residency or other requirements. S3 is designed to provide 99.999999999% durability and 99.99% availability of objects over a given year. Most of its storage classes store objects redundantly over Availability Zones, which are defined as one or more discrete data centers with redundant power, networking, and connectivity in an AWS Region.

Your application can call S3 PutObject API using the AWS SDK to upload the documents programmatically to the intended S3 bucket. You can choose to upload the documents into a prefix like `/raw/2025/february/13/` so that each document's S3 path will be something like the following `s3://<bucket name>/raw/2025/february/13/file123.txt`. Organizing the prefix well can make it easier for you to use certain features in S3, such as analyzing usage with S3 Storage Lens and managing the objects lifecycle with S3 Lifecycle. It can also be easier for debugging.

When the document size is large, your application code can call different S3 APIs to perform multipart upload in order to speed up the upload process using a parallel upload mechanism.

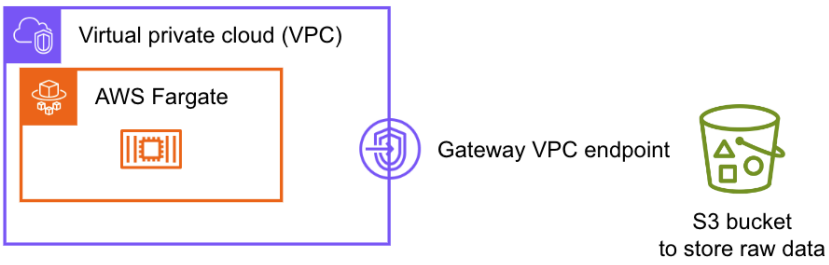
If you design your application in a way that the end users can upload documents, e.g. files, videos, images, straight to S3 bucket, you can use S3 presigned URL to allow the browser/app to upload directly to a private S3 bucket without having to go through your backend API. This can free up your application compute resource from having to allocate CPU clock cycles for upload operation, especially for big files like videos.

The application needs sufficient permissions to call S3 API. You can either configure an S3 bucket policy on the bucket to explicitly define who can access the resources in that bucket or configure the application's assumed AWS Identity and Access Management (IAM) principal with sufficient permissions. Since this topic is very important, I suggest having a proper read on this guide from AWS https://docs.aws.amazon.com/AmazonS3/latest/userguide/security-iam.html.

It is a best practice to apply the grant least privilege principle to improve security. Do not unnecessarily give permissions to your S3 resources to the unintended users/applications. Do not make your S3 bucket public for this use case.

If your application is in AWS, it is very likely that you can assign an IAM role into the application, which is more secure since the credentials are temporary. If your application is hosted in Kubernetes with Amazon Elastic Kubernetes Service (EKS), you can associate an IAM role with the Kubernetes service account. You can associate IAM policy, which contains the permissions you defined, into an IAM role.

From the networking perspective, S3 APIs can be called via the internet and S3 supports HTTPS endpoints to allow encryption in transit. However, when your application resides in a private subnet in Amazon Virtual Private Cloud (VPC) without internet egress access, you can use a gateway VPC endpoint for Amazon S3 to access S3 with no additional charge for using it. This is so that you can save some cost from having to route the documents upload traffic through a NAT (network address translation) component. The diagram below illustrates this setup with your application running in AWS Fargate, a serverless compute engine for containers, in an AWS VPC.



When your application resides in a private subnet outside AWS VPC, e.g. on premises, and it connects to AWS VPC through AWS Direct Connect or VPN, then you can use AWS PrivateLink for Amazon S3 instead to keep S3 reachable without traversing the internet.

In addition to access control and networking, there are other considerations on using S3, such as document versioning with S3 Versioning, custom encryption at rest, object locking to avoid accidental deletion, tiering down older documents with S3 Lifecycle or S3 Intelligent-Tiering for cost optimization, integrity check with checksums, and more. Evaluate your requirements carefully and leverage the relevant features in S3.

The output data or the inference result can be stored in a different S3 bucket or in the same bucket with a different prefix. The application can call ListObjectsV2 API to check whether

the output is ready and call GetObject API to obtain the data, given sufficient IAM permissions and correct network setup.

When the output data is intended to be accessible directly by your application's end users, such as in the case of video or image generation, you can make the data available in a more secure way by keeping the bucket private and using S3 presigned URL to access the data. When the user's browser/app sends a request to access the output data into your application's backend API, the backend API can create an S3 presigned URL using AWS SDK for that particular data with certain expiration configuration. The backend API then returns this URL to the frontend application. The browser/app can use the URL to retrieve the data from S3.