THALES AWS GENAI/ML RESOURCE KIT

PROTECTING SENSITIVE DATA

THALES

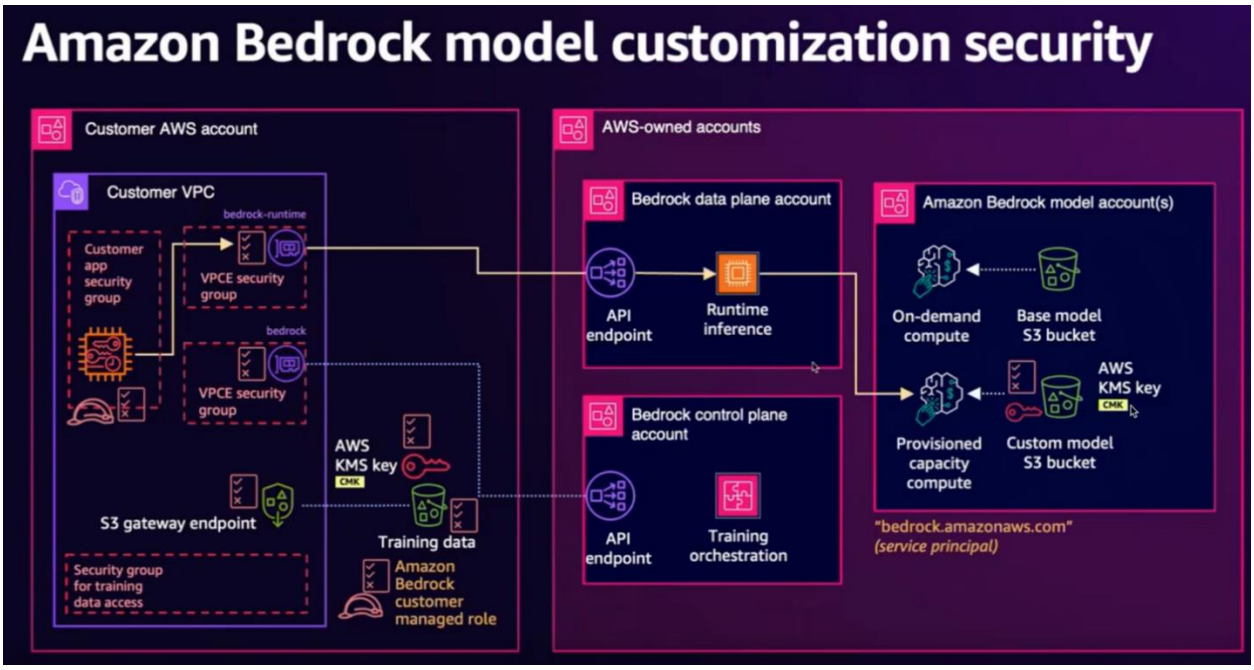
Contents

1 | Introduction

Businesses are increasingly turning to Generative AI (GenAI) to enhance operations and unlock new revenue streams. Utilizing proprietary, sensitive data is crucial for realizing the full potential of GenAI. However, this introduces significant security and compliance challenges. This document outlines solutions for protecting sensitive data within GenAI models, ensuring both innovation and regulatory adherence.

From a security perspective there are many potential security risks associated with the implementation of GenAi systems such as Data Poisoning Definition, Backdoor insertion Definition ,Model theft definition and Prompt injection definition attacks. ***The focus of this document will be on implementing Thales solutions with AWS capabilities to ensure sensitive data such as PII, PCI, etc will be protected.***

Listed below is the AWS Bedrock AI managed service for security. The diagram highlights the areas two separate scopes of control, customer AWS accounts and aws owned accounts. As you can see from the diagram both areas incorporate the AWS KMS for encryption keys to protect the data.



Thales provides customers the ability to populate the AWS KMS with key material created in an external key manager (CipherTrust Manager) using customer managed keys to achieve better control and compliance. See Appendix on Bring Your Own Encryption & Hold Your Own Key for more information.

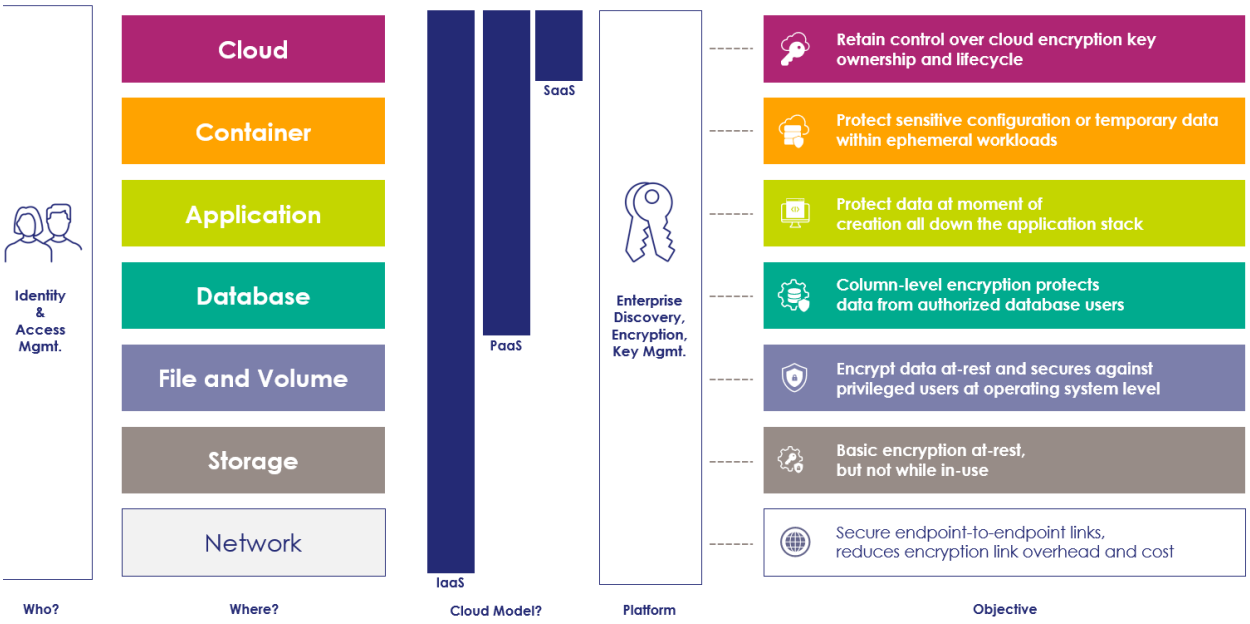
One thing to keep in mind is that using a strategy of Bring Your Own Key (BYOK) or Hold Your Own Key (HYOK) may be sufficient for many use cases, depending on the classification of your data, it

may not be appropriate for use cases where the data is highly sensitive. When implementing AWS native encryption, it is important to know what threats you are protected from with this kind of strategy. The diagram below highlights the different type of encryption and the threat you are protected from. AWS "Native" encryption is somewhat similar to Full Disk Encryption (FDE) that customers deploy for their on-premises disk. The difference with AWS KMS is that instead of having one key for the entire disk AWS provides customers the ability to have their own key for just their "service or portion of the disk" (whatever the key is protecting in the CSP). The other difference is that customers have the ability to disable the key to prevent access to the service which makes the entire service unavailable.

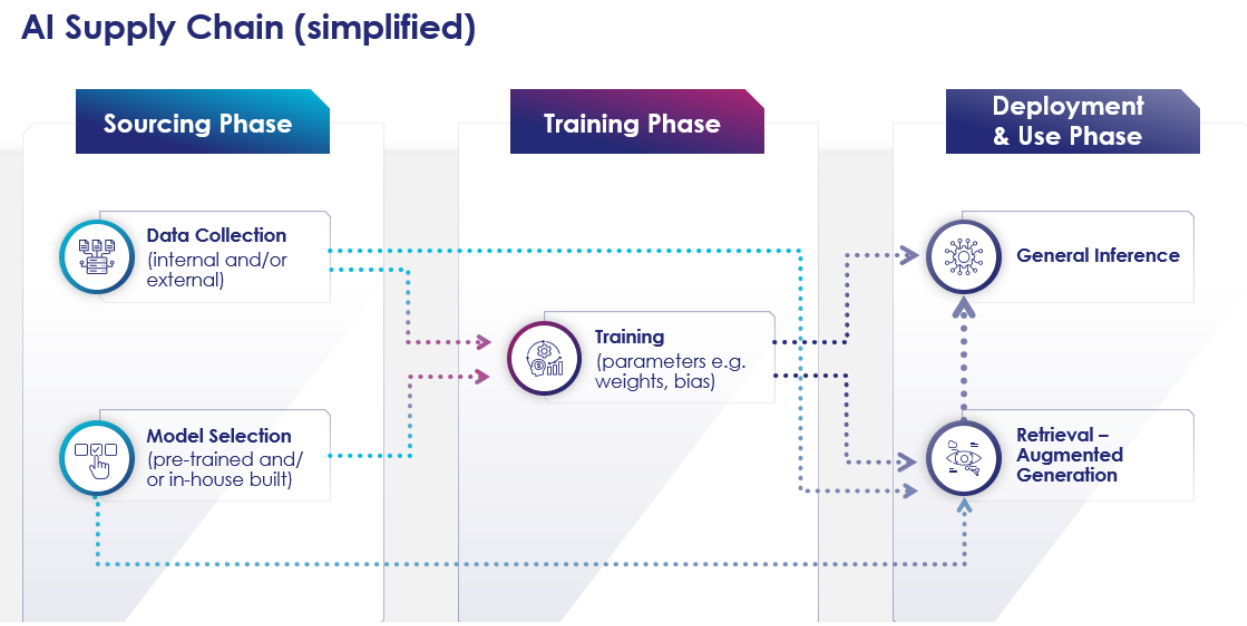
As seen from the diagram above when using the AWS GenAI capabilities some of the locations of data reside on S3 buckets in the customer's control. Looking at the encryption stack below it is necessary to deploy a different kind of encryption such as, bring your own encryption (BYOE) to prevent threats where a user's Operating System credentials have been compromised.

For more information on how this can be accomplished see the section titled Bring Your Own Encryption in the Appendix.

Data Protection Use Cases



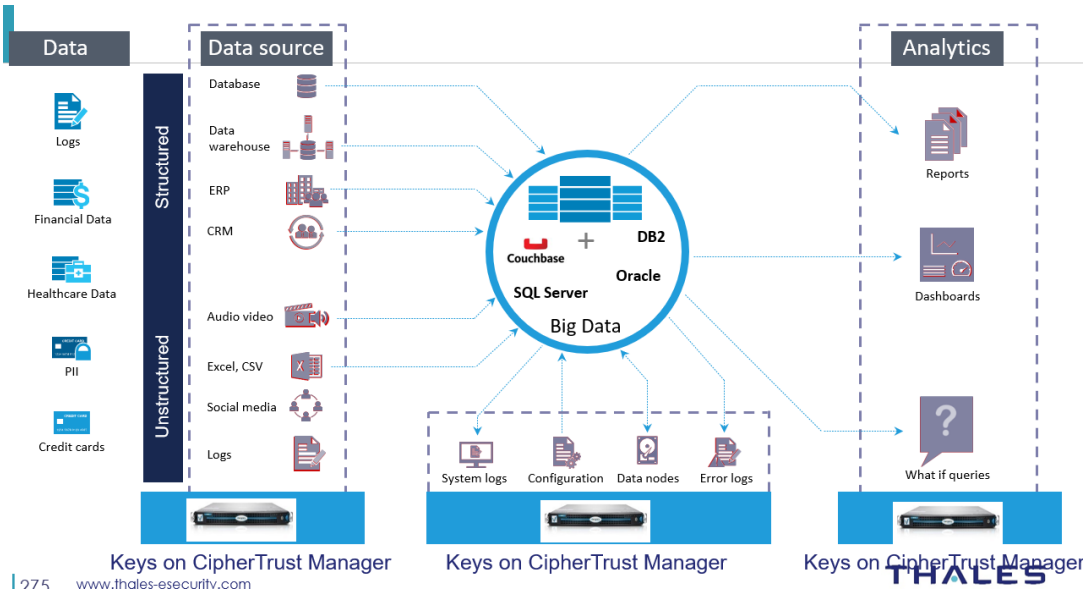
The Generative AI workflow can be broadly categorized into three distinct phases. First, the **sourcing phase** involves critical decisions regarding data collection, encompassing the gathering and preparation of relevant datasets, and model selection, where the appropriate pre-trained or custom model is chosen based on the desired application. Next, the **training phase** focuses on fine-tuning the selected model using the sourced data, optimizing its parameters to generate desired outputs. Finally, the **deployment phase** entails making the trained model accessible for practical use, encompassing both general inference, where the model generates outputs based on user prompts, and Retrieval Augmented Generation (RAG), which enhances responses by grounding them in external knowledge sources for improved accuracy and contextuality.



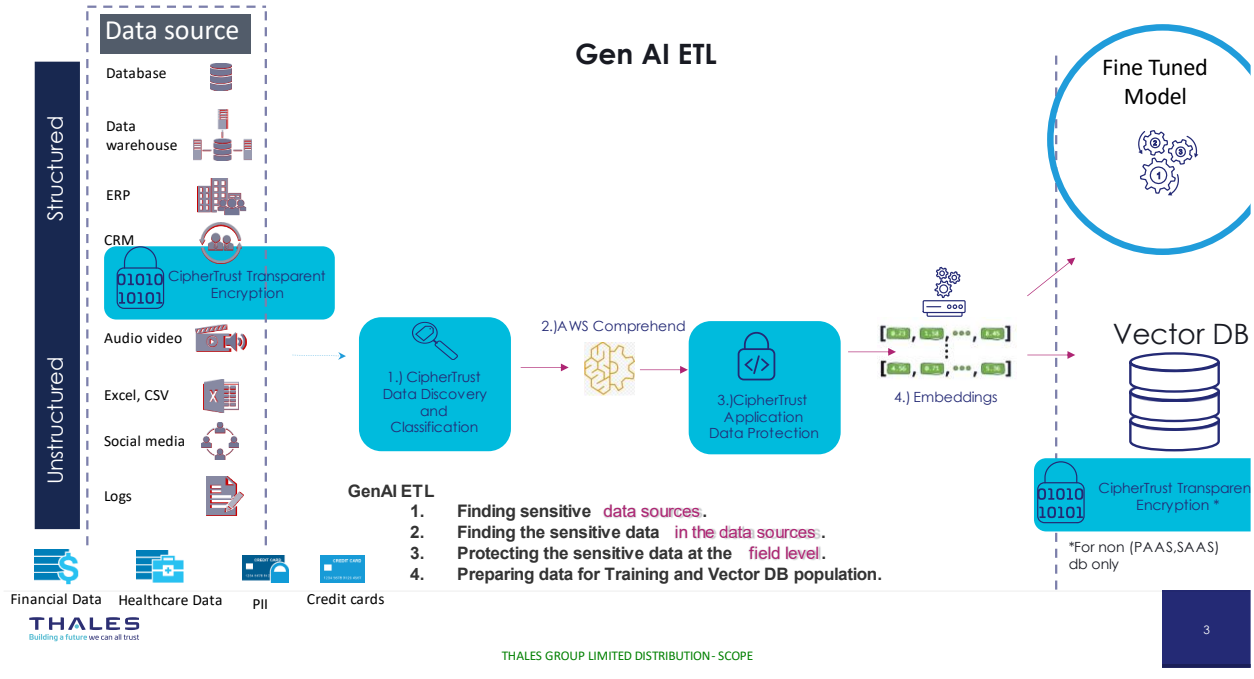
2 | Sourcing Phase.

The first phase is dedicated to sourcing, we want to define which data we will use for the training phase. Data appropriate for GenAI can come from several locations both structured and unstructured***. It is important to protect this data both at a file level and eventually the field or column level as well***. It is estimated that about 80% of organization data is in unstructured data sources which is something to be aware of as finding sensitive data can be a cumbersome and time-consuming process. Organizations should plan on using appropriate data discover and classification tools to identify where the sensitive data is and then once found protect it. Both Thales and AWS provide this capability which will be explained later in this document. See Appendix section "**Finding sensitive data sources and protecting it"** for more information on this topic.

This workflow is very similar to what customers are already implementing to build data warehouses and data marts. The Extract Transform and Load (ETL) process is to extract data from source systems, consolidate, clean and protect the data in a manner that can provide value to the business. The diagram below shows a traditional ETL environment.



As can be seen above there are many locations during this process where sensitive data may reside. Depending on the classification of the data much of this information should be protected with a strategy like (BYOE) encryption that is stronger than full disk encryption (FDE) as it is **vulnerable to anyone who has access to the operating system.** There are two non-mutually exclusive strategies to protect sensitive data during this process using BYOE protecting the entire file or database and or protecting the column in a database or field in a file. Please see BYOE section for more information. The same applies for the GenAI ETL process which is shown below.

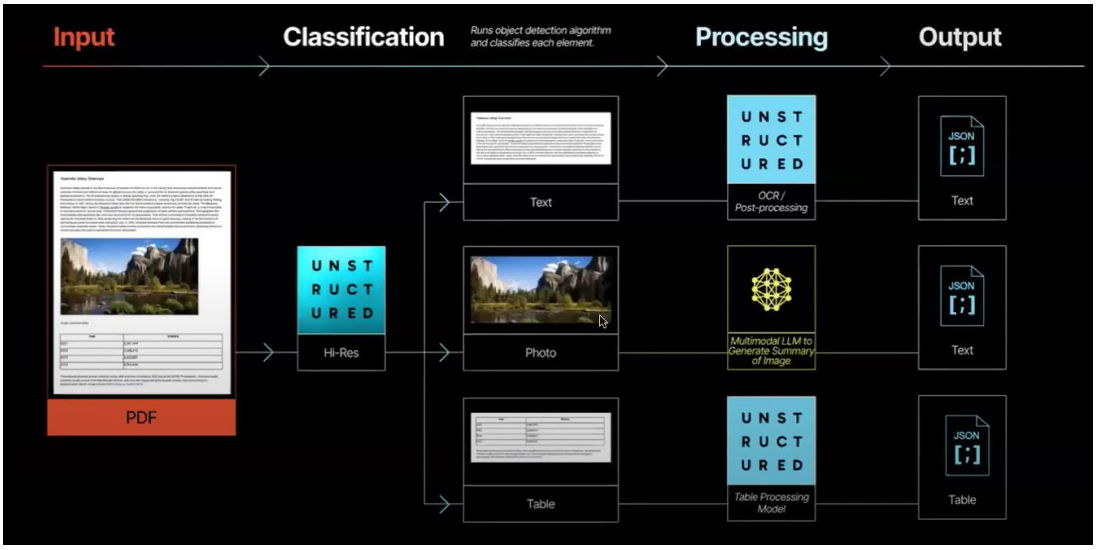


As noted earlier with GenAI and ML environments it is estimated that about 80% of the data to be used is unstructured which typically has **not** been verified to be PII, PCI etc safe. The diagram above shows how the GenAi ETL process has some unique steps that are necessary to prepare the

data for training or the RAG but conceptually the process to protect the data is the same except for the fact that a higher % of the sources will be unstructured data.

1. **Finding sensitive data sources.**
2. **Finding the sensitive data in the data sources.**
3. **Preparing data for Training and Vector DB population.**
4. **Protecting it before its loaded into the Fine Tunned Model**

A common example of unstructured data is pdf file that needs to be loaded into a fine tunned model or a vector database. As you can see below the first step is to extract the text from the document. Once this is done text scanning for sensitive data should be incorporated to the workflow converting any sensitive data to encrypted format and written back out in its original sequence replacing the sensitive **field** with the encrypted value. Once this has been accomplished then the data will be ready for output either for training the model or for the Vectordb to be used for the RAG.

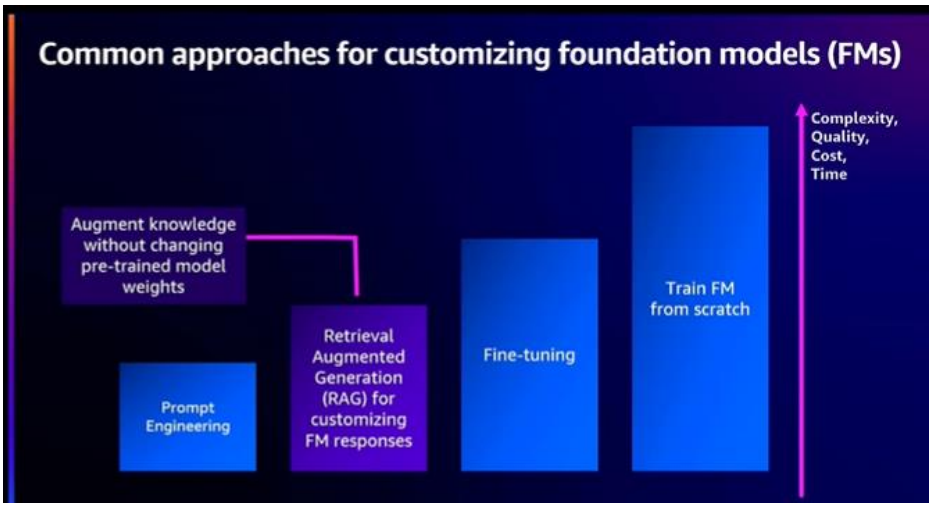


If the original sensitive data has not been protected the resulted model will inherit the same data classification as the original data set that was used to train/fine-tune it. This model can now be put under the various sophisticated jail-break, prompt injection attacks and can be forced to reveal the sensitive data it contains.

There are various methos (encryption, fine grain authorization) to protect the sensitive data outside of the GenAI model. For example, on how this can be accomplished see Appendix "**Protecting sensitive fields in in file"**

3 | Training Phase

Then we have the second phase, the training phase, which is optional depending on your use case. Amazon defines Foundation Models as : Foundation models (FMs) are LLMs that can perform a wide range of disparate tasks with a high degree of accuracy based on input prompts AWS provides this diagram which describes the various options to customizing the Foundation Models.



Here is a summary of the differences between the two different approaches when considering building models.

Training a Model (Training from Scratch):

- **Definition:**
  - This involves building an AI model from the ground up. It requires a massive dataset and significant computational resources.
  - The model's parameters are initialized randomly, and the training process involves adjusting these parameters iteratively until the model can perform the desired tasks.
- **Characteristics:**
  - Requires very large datasets.
  - Demands substantial computational power.
  - Allows for maximum customization.
  - Time-consuming and expensive.
  - This is how foundation models like the original versions of GPT were created.
- **Use Cases:**
  - Creating entirely new models for novel applications.
  - Developing foundational models with broad general capabilities.

Fine-Tuning a Model:

- **Definition:**
  - This involves taking a pre-trained model (a model that has already been trained on a large, general dataset) and further training it on a smaller, more specific dataset.
  - The goal is to adapt the pre-trained model to perform better on a particular task or within a specific domain.
- **Characteristics:**
  - Uses a smaller, task-specific dataset.
  - Requires less computational power than training from scratch.
  - Faster and more cost-effective.
  - Leverages the knowledge already learned by the pre-trained model.
  - Adapts a general model to a more specific purpose.

- **Use Cases:**
    - Adapting a general-purpose LLM for specific business applications.
    - Customizing a model's output style or tone.
    - Improving a model's performance on a particular type of data.

In essence:

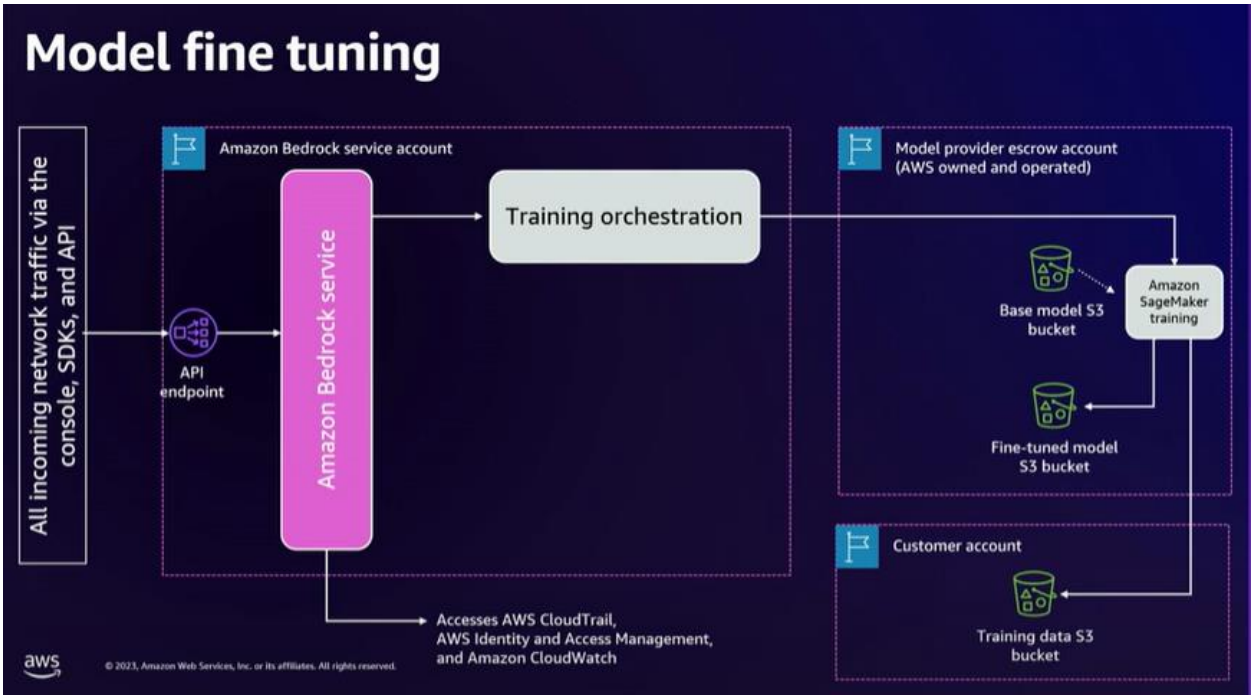
- Training is like building a house from the foundation.
- Fine-tuning is like renovating an existing house to better suit your needs.

For both use cases the training phase consists of 4 main tasks before the data can be loaded into the model. See Appendix for more information on each of these topics.

- **Finding sensitive data sources.**
- **Finding the sensitive data in the data sources.**
- **Preparing data for Training.**
- **Protecting it before its loaded into the Fine Tunned Model**

Weather you decide to protect the data at a field level or not with the process above you also may consider strong levels of protection for the data stored in S3 as there may be other data you do not want revealed to unintended audiences.

As you can see the model below fine tuning provided by AWS includes native encryption of the data in the S3 bucket which again may not be enough protection depending on your data sensitivity.



For fine-tuning text-to-text models, each JSON object is a sample containing structured fields designed to guide the model toward generating the desired textual output based on a provided textual prompt. The data format varies depending on the use case, broadly categorized into non-conversational and conversational use cases. Non-conversational tasks involve standalone prompts and outputs, while conversational tasks can be further divided into single-turn exchanges, where the model responds to a single user input, and multi-turn dialogues, where the model maintains context across multiple exchanges. As you can see in both formats below are potential to have sensitive data.

https://docs.aws.amazon.com/bedrock/latest/userguide/preparing-text-data.html

```
{"prompt": " Summarize the medical history of Bobby JacksOn?", "completion": "Bobby JacksOn, a 30-year-old male with blood type B-, was admitted on 2024-01-31 to Sons and Miller hospital for Cancer. He was under the care of Dr. Matthew Smith and his insurance provider is Blue Cross. His billing amount was $18856.28 and he was in room 328 for an urgent admission, discharged on 2024-02-02. He was prescribed Paracetamol and his test results were Normal"}
```

You can use the Amazon Bedrock Converse API to create conversational applications that send and receive messages to and from an Amazon Bedrock model. For example, you can create a chat bot that maintains a conversation over many turns and uses a persona or tone customization that is unique to your needs, such as a helpful technical support assistant. Here is an example of the format needed to support the Converse API.

Converse API format (Single turn and Multi turn)

```
{
    "schemaVersion": "bedrock-conversation-2024",
    "system": [
        {
            "text": "You are a digital assistant with a friendly personality"
        }
    ],
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "text": "What is the capital of Mars?"
                }
            ]
        },
        {
            "role": "assistant",
            "content": [
                {
                    "text": "Mars does not have a capital. Perhaps it will one day."
                }
            ]
        }
    ]
}
```

Amazon Bedrock also provides another method to train models called “Model Distilling”. *Model distillation* is the process of transferring knowledge from a larger more intelligent model (known as teacher) to a smaller, faster, cost-efficient model (known as student). In this process, the student model's performance improves for a specific use case. Amazon Bedrock Model Distillation uses the latest data synthesis techniques to generate diverse, high-quality responses (known as synthetic data) from the teacher model, and fine-tunes the student model. For more details see the two links below.

https://aws.amazon.com/blogs/machine-learning/amazon-bedrock-model-distillation-boost-function-calling-accuracy-while-reducing-cost-and-latency/

https://docs.aws.amazon.com/bedrock/latest/userguide/distillation-data-prep-option-2.html

https://github.com/aws-samples/amazon-bedrock-samples/blob/main/custom-models/model_distillation/dataset-validation/README.md

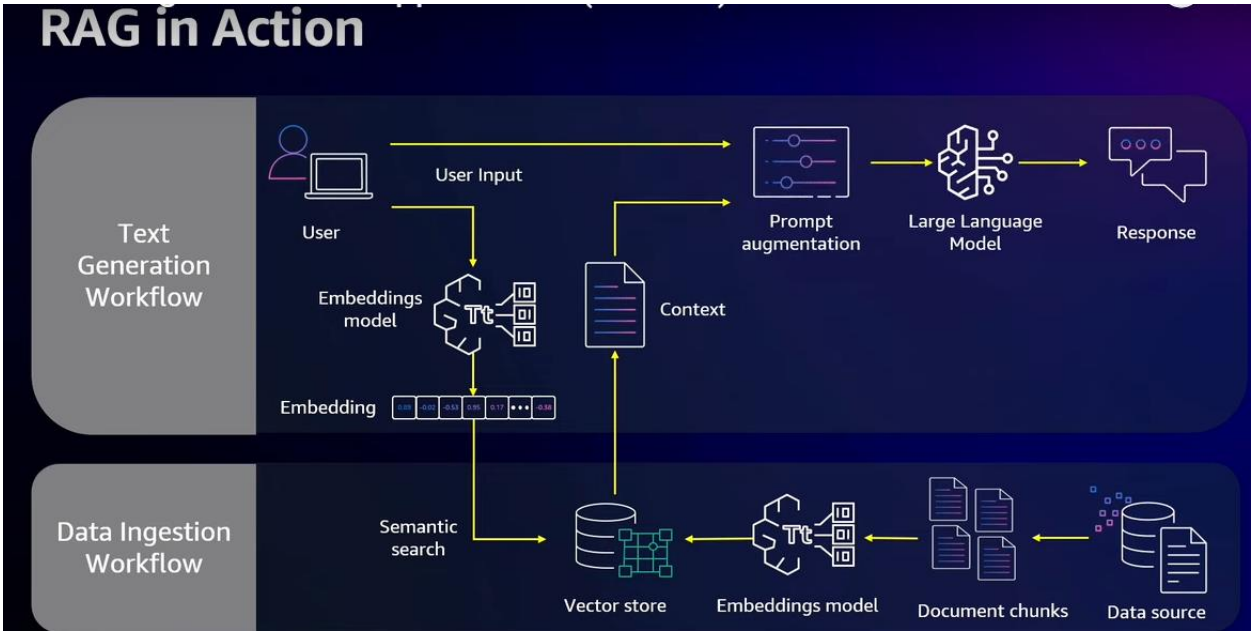
4 | Deployment & Use Phase.

And the last phase is the inference phase, where the decision makers will select a General Machine learning/General Inference technique or a RAG technique.

Retrieval Augmented Generation (RAG) addresses the limitations of Large Language Models (LLMs) by providing them with access to relevant, up-to-date, and domain-specific data. By grounding LLM responses in proprietary, private, or dynamic information, RAG significantly enhances accuracy and contextuality, overcoming the inherent static nature and knowledge gaps of LLMs. This is crucial for business applications requiring precise, context-aware outputs.

RAG also enables InfoSec team to apply the necessary security tools which are technically not possible within the model itself.

AWS uses something called Knowledge Bases for Amazon Bedrock. It is a fully managed RAG capability that allows you to customize FM responses with contextual and relevant company data. The diagram below shows the entire workflow that can be automated by using AWS Knowledge Bases.



Knowledge Bases for Amazon Bedrock automates the complete RAG workflow, including ingestion, retrieval, prompt augmentation, and citations, removing the need for you to write custom code to integrate data sources and manage queries. For unstructured data sources containing multimodal

data, you can configure Amazon Bedrock Knowledge Bases to parse, analyze, and extract meaningful insights. For seamless processing of complex multimodal data, you can choose between Bedrock Data Automation or foundation models as the parser. For structured data sources, Amazon Bedrock Knowledge Bases provides a built-in managed Natural Language to Structured Query Language for generating a query command, allowing users to retrieve data directly from the source without the need to move or preprocess the data. See links below for more information.

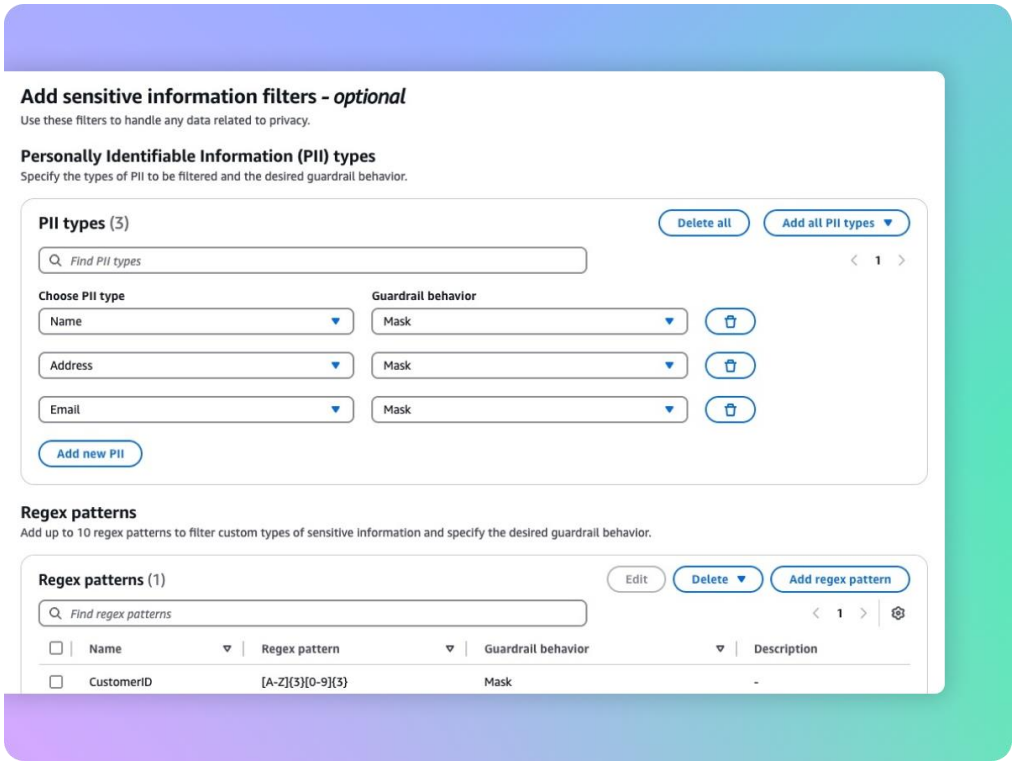
https://aws.amazon.com/bedrock/knowledge-bases/

https://www.youtube.com/watch?v=N0tlOXZwrSs

For use cases where you do need to modify the prompts there are a couple of ways to do this with either AWS Guardrails or using their Prompt Management API's.

AWS Guardrails

Guardrails helps evaluate user inputs and model responses based on use case–specific policies and provides an additional layer of safeguards on top of those natively provided by FMs. Guardrails helps you detect sensitive content such as personally identifiable information (PII) in user inputs and FM responses. You can select from a list of predefined PII or define a custom sensitive-information type using regular expressions (RegEx). Based on the use case, you can selectively reject inputs containing sensitive information or redact them in FM responses. For example, you can redact users' personal information while generating summaries from customer and agent conversation transcripts in a call center. Listed below is a screenshot showing the PII types and the behavior to implement if found.



AWS Guardrails leverages Amazon Comprehend API which is the same API that was used to find sensitive data in the other examples listed in the Appendix for finding sensitive data during the training phase and prompt management. In the appendix of this document is an example of how to use Thales encryption capabilities to protect sensitive data going to the model in the request and from the model response.

https://docs.aws.amazon.com/bedrock/latest/userguide/guardrails-use-converse-api.html

https://aws.amazon.com/bedrock/guardrails/

AWS Guardrails also provides the ability to create a lambda function that can be automatically invoked at the same time as the above screen and when a response is generated to provide custom functionality to protect sensitive data. As you can see from above the option is to mask the data but there may be situations during model verification or increase compliance requirements where masking is not good enough. See Appendix section titled **Examples Guardrails** for more information.

Prompt Management/Engineering

AWS provides a comprehensive UI to manage prompts. AWS also provides a robust API called converse to enhance your prompts. Prompt Management on Bedrock also enables you to store custom metadata with the prompts. You can store metadata such as author, team, department etc. to meet your enterprise prompt management needs. If some of this information is sensitive, it may need to be encrypted to protect it from unintended usage. Prompt engineering is about *how* to write good prompts and prompt management is about *how* to organize and deploy those prompts effectively in both use cases sensitive data could be contained from the original data entered by the user or data that has been obtained from the Vector database or SQL queries for user specific content. Listed below are a couple of links that provide more information.

https://aws.amazon.com/bedrock/prompt-management/

https://community.aws/content/2hUiEkO83hpoGF5nm3FWrdfYvPt/amazon-bedrock-converse-api-java-developer-guide

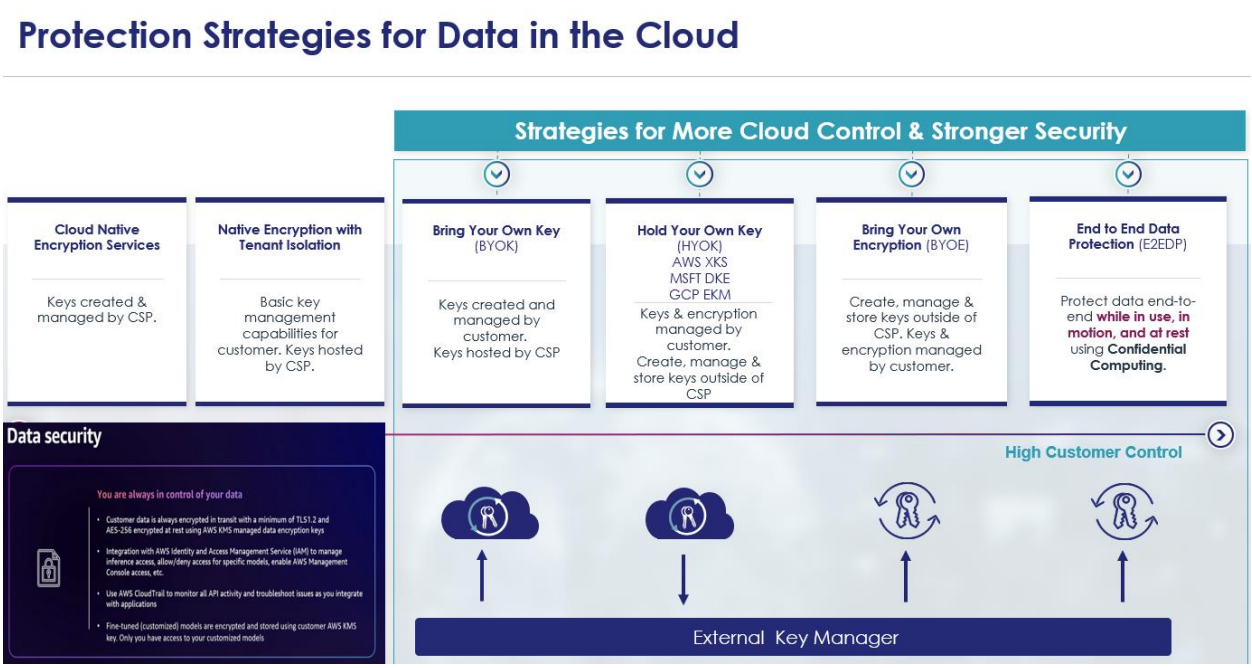
In the appendix of this document (**Prompt Protection**) is an example of how to use Thales encryption capabilities to protect sensitive data either going into the prompt or from the model response.

5 | Appendix.

Bring Your Own Keys and Hold Your Own Keys

Native Encryption Services: Can be deceptively simple to use, but very difficult to live with…The problem with native encryption & key mgmt. is that it is UNIQUE to the provider. These services are 100% managed by the CSPs. Also, customers have limited control and visibility over their cloud-encrypted data. There is no way to know who is accessing what and why; depending on the industry you are in, the sensitivity of data and the Cloud Service Provider, you may need to complement cloud

security with additional controls for compliance. The dark blue section on the lower left is a description of AWS Data Security for GenAI outlining the native KMS encryption.



Bring Your Own Key (BYOK): For customers that need greater control and visibility over the cloud-encrypted data, the providers offer variations of flexible key management such as Bring Your Own Key (BYOK) It is unique to the customer, depending on how they generate and manage their keys. Now, you have some control over your encrypted data because you can bring your own key material or master keys. Customers have the ability to generate and import the encryption keys or key material for their cloud-native encryption services. But still, there are gaps in visibility, control and usage of keys. Nobody know what happens after you bring your own key, if the cloud admins can access them, what happens in case of a breach or government subpoenas, etc.. Require additional tools for multi-cloud and hybrid data security & compliance.

Hold Your Own Key (HYOK) refers to a security model where organizations maintain full control over their encryption keys, even when using cloud services provided by a third party like **Google Cloud Platform (GCP)**, **Amazon Web Services (AWS)**, or **Microsoft Azure**. In this model, the organization uses its own key management infrastructure, often located on-premises or in a third-party environment, to generate, store, and manage encryption keys. The cloud provider does not have access to these keys, ensuring that even if the cloud infrastructure is compromised, the encrypted data remains secure and inaccessible.

Benefits of External Key Manger

For both Bring Your Own Keys and Hold Your Own Keys you get can obtain the following benefits from the Key Manager.

1. Key Lifecycle Management

    a. Detail: Native CSP key management services has limited ability to automate the lifecycle of keys especially across multiple subscriptions
       Impact: Customers have to implement expensive manual key management processes to meet internal key security requirements
2. Attaining Compliance
    a. Detail: Insufficient authorization control or DR services to ensure keys are not accidentally or intentionally deleted
       Impact: There is too high of risk that data will be lost
    b.
3. Encryption Key Visibility
    a. Detail: Internal and external regulations require that encryption keys do not permanently reside in the cloud
       Impact: Enterprise cannot move data to the cloud where data regulations require more control of keys that are locally stored
    b.
4. Assign permissions to keys
    a. Only allow certain accounts access and operations such as encrypt,decrypt. Prevent deletes unless have a quarum. Have dates expiration dates with automatic key roation schedules, Keys have states (active,destroyed,deactivated) as approved by nist. Should be able to search on keys by attributes like algorithums, key length etc.

For more information see link below:

https://www.thalestct.com/wp-content/uploads/2022/09/Best-Practices-Cloud-Data-Protection-and-Key-Management-TCT-WP.pdf

https://cpl.thalesgroup.com/blog/cloud-security/shared-responsibility-model-cloud

Bring Your Own Encryption (BYOE)

There are a couple of use cases to consider when using a BYOE strategy to protect sensitive data for GenAI platforms. The first is protecting the entire file or all files in a particular directory or S3 bucket and only allowing a certain process or user to access the content. The second is to protect the fields in a file that contain sensitive data. For working example on how to protect sensitive fields in a file see Appendix section on Examples.

Protecting the entire file.

Thales provides a capability called CipherTrust Transparent Encryption(CTE) to protect the entire file in a directory or S3 bucket. See links for more information on CTE.

https://cpl.thalesgroup.com/encryption/transparent-encryption

https://www.youtube.com/watch?v=VzRxrhUZfc0

https://www.thalestct.com/wp-content/uploads/2022/09/aws-s3-with-cte-sb.pdf

https://www.thalestct.com/wp-content/uploads/2022/09/avoiding-amazon-s3-leaks-wp.pdf

Protecting fields in a file.

Thales provides several different application encryption and tokenization capabilities.

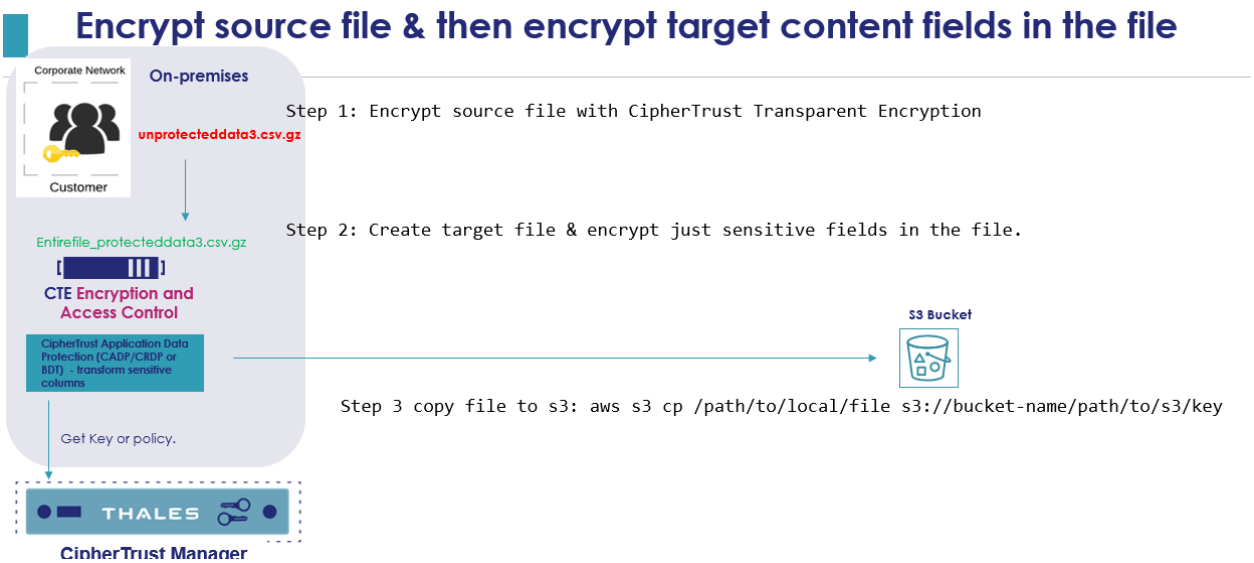
See links below for more information.

https://cpl.thalesgroup.com/encryption/ciphertrust-application-data-protection

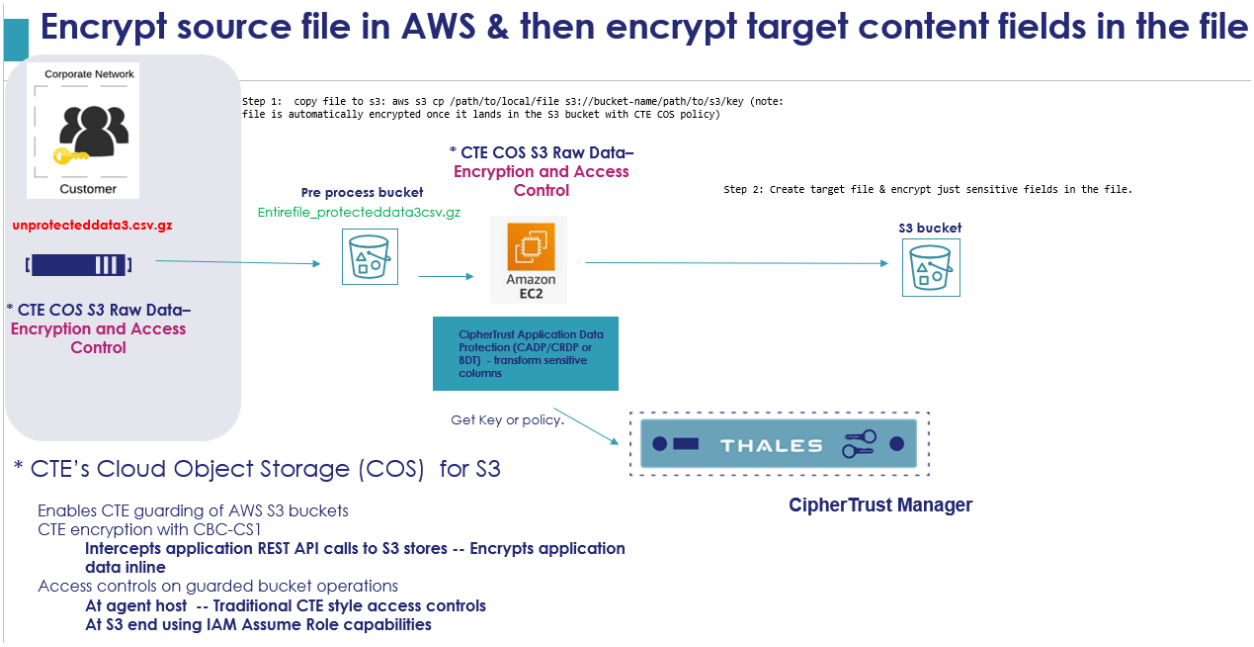
https://cpl.thalesgroup.com/encryption/tokenization

Note for some use cases it might be necessary to protect the file with both types of encryption depending on what stage of the process it is during the ETL process. Listed below are a couple of options on how this can be accomplished.

Option 1. Encryption on premises.



Option 2. Encryption on AWS.



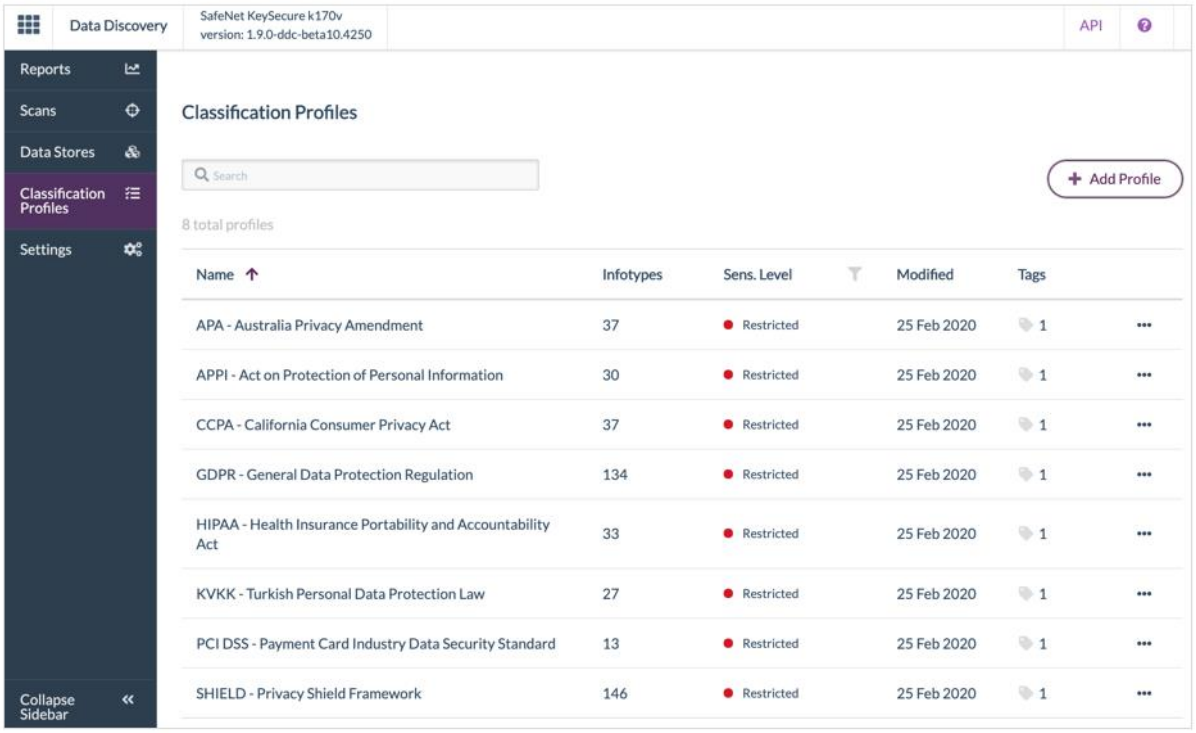
For working example see **Protecting sensitive fields in in file** in the Appendix Examples section.

Finding sensitive data sources and protecting it.

This content in this section focuses on encryption of sensitive data for AI and ML environments. It will explain how data needs to be protected at a field level in a file before the content is made available to a vector database or used for training an AI model. Since the content is specifically used by the models encrypting it at a field level is the most secure and appropriate for this use case.

When protecting sensitive data for general business use protecting sensitive data in a file at a field level may or may not be appropriate since decrypting on the fly and incorporating RBAC may not be available to the applications using the files. There are other methods such as Thales CipherTrust Manger Transparent Encryption (CTE) that may be more appropriate for these kinds of use cases.

As noted earlier it is critical that organizations find data sources that have sensitive information to avoid breaches. Thales provides a product that will find the datasources that contain sensitive data and also provide a report that shows the sensitivity and other metrics such as the information type (credit card, email, etc). Listed below is a screenshot showing the classification profiles and sensitivity levels.



Once Thales Data Discovery and Classification (DDC) has found the sensitive data at a **file level** it needs to be protected at **field level**. In order to find the actual attribute in the file AWS provides a very powerful API called comprehend that can find PII data in a file and then once found can be protected with Thales encryption. See Examples section below for detailed example.

https://cpl.thalesgroup.com/encryption/data-discovery-and-classification

https://docs.aws.amazon.com/comprehend/latest/dg/what-is.html

Examples

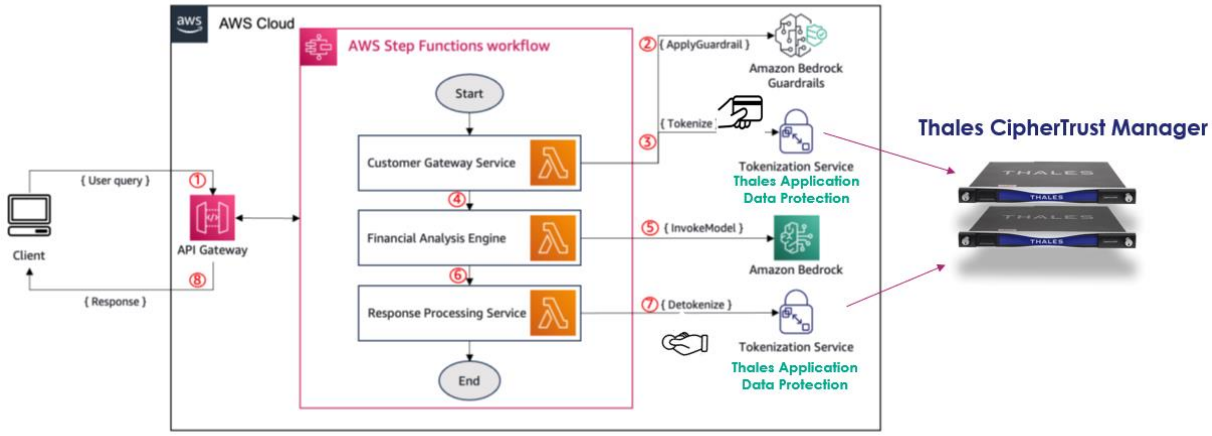
Guardrails

Amazon Bedrock Guardrails helps detect sensitive information, such as PII, in standard format in input prompts or model responses. Sensitive information filters give organizations control over how sensitive data is handled, with options to block requests containing PII or mask the sensitive information with generic placeholders like {NAME} or {EMAIL} . This capability helps organizations comply with data protection regulations while still using the power of large language models (LLMs).

Although masking effectively protects sensitive information, it creates a new challenge: the loss of data reversibility. When guardrails replace sensitive data with generic masks, the original information becomes inaccessible to downstream applications that might need it for legitimate business processes. This limitation can impact workflows where both security and functional data are required.

Tokenization or Format Preserve Encryption (FPE) offers a complementary approach to this challenge. Unlike masking, tokenization replaces sensitive data with format-preserving tokens that are mathematically unrelated to the original information but maintain its structure and usability. These tokens can be securely reversed back to their original values when needed by authorized systems, creating a path for secure data flows throughout an organization's environment.

The solution described below demonstrates how to combine Amazon Bedrock Guardrails with tokenization services (CRDP) from Thales CipherTrust Data Security Platform to create an architecture that protects sensitive data without sacrificing the ability to process that data securely when needed. This approach is particularly valuable for organizations in highly regulated industries that need to balance innovation with compliance requirements.




```
import json
import os
import requests

def process_guardrail_output(lambda_return_string, guardrail_json_string):
    """
    Processes the Lambda function's return string and the Guardrail's JSON output
    to replace PII placeholders with protected data.

    Args:
        lambda_return_string (str): The JSON string returned from the Lambda function.
                                    Expected format: {'statusCode': 200, 'body': '{"message":
..., "protection_results": [...]}'}
        guardrail_json_string (str): The JSON string representing the Guardrail's
                                     intervention output. Expected to contain
                                     "outputs" with a "text" field.

    Returns:
        list: The modified 'outputs' list from the Guardrail output, with placeholders replaced,
              or None if processing fails (e.g., status code not 200, parsing error).
    """
    try:
        # 1. Parse the Lambda function's return string
        lambda_return_dict = json.loads(lambda_return_string)

        if lambda_return_dict.get('statusCode') != 200:
            print(f"Lambda function returned non-200 status code:
{lambda_return_dict.get('statusCode')}")
            return None
```

```
        protection_results_body_str = lambda_return_dict.get('body')
        if not protection_results_body_str:
            print("Lambda return body is missing or empty.")
            return None

        protection_data = json.loads(protection_results_body_str)
        protection_results = protection_data.get('protection_results', [])

        protection_map = {}
        for res in protection_results:
            if 'type' in res and 'protection_response' in res and 'protected_data' in
res['protection_response']:
                protection_map[res['type'].upper()] =
res['protection_response']['protected_data']
            else:
                print(f"Warning: Malformed protection result entry: {res}")


        # 2. Parse the Guardrail's output JSON string
        guardrail_output_dict = json.loads(guardrail_json_string)

        # 3. Access and modify the outputs
        if 'outputs' in guardrail_output_dict and isinstance(guardrail_output_dict['outputs'],
list):
            modified_outputs = []
            for output_item in guardrail_output_dict['outputs']:
                if 'text' in output_item and isinstance(output_item['text'], str):
                    original_text = output_item['text']
                    modified_text = original_text

                    for pii_type, protected_value in protection_map.items():
                        placeholder = "{" + pii_type + "}"
                        modified_text = modified_text.replace(placeholder, protected_value)

                    # Create a new dictionary for the modified output item
                    modified_outputs.append({"text": modified_text})
                else:
                    # If an output item doesn't have 'text', include it as is or skip
                    modified_outputs.append(output_item) # Or handle as per your requirement
            return modified_outputs # Return the list of modified output items
        else:
            print("Guardrail output JSON does not contain 'outputs' or it's not a list.")
            return None

    except json.JSONDecodeError as e:
        print(f"JSON parsing error: {e}")
        return None
    except KeyError as e:
        print(f"Missing expected key: {e}")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None


def lambda_handler(event, context):
    lambda_event_for_thales = {
        'body': json.dumps(event)
    }

    results = thales_udf(lambda_event_for_thales, None)

    # process_guardrail_output now returns the list of modified output dictionaries
    processed_outputs_list = process_guardrail_output(json.dumps(results), json.dumps(event))

    if processed_outputs_list is None:
        return {
            'statusCode': 500,
            'body': json.dumps({'message': 'Failed to process guardrail output.'})
        }
```

```
    # Return the processed outputs list directly as the body
    return {
        'statusCode': 200,
        'body': json.dumps(processed_outputs_list)
    }


def thales_udf(event, context):
    """
    AWS Lambda function to extract PII entities from an input JSON
    and make a REST call to protect each sensitive value.

    Args:
        event (dict): The Lambda event object. Expected to contain the
                      input JSON data in 'body' (e.g., from API Gateway).
        context (object): The Lambda context object.

    Returns:
        dict: A dictionary containing the statusCode and a body with
              the results of the protection calls or an error message.
    """
    # --- Configuration (Ideally set as Lambda Environment Variables) ---
    CRDP_IP = os.environ.get("CRDPIP", "yourip")
    PROTECTION_PROFILE = os.environ.get("PROTECTION_PROFILE", "plain-alpha-internal")

    DATA_KEY = "data"
    MODE = os.environ.get("MODE", "protect")
    if MODE != "protect":
        DATA_KEY = "protected_data"

    USERNAME = os.environ.get("USER_NAME", "testingonly")
    KEY_METADATA_LOCATION = os.environ.get("KEY_METADATA_LOCATION", "internal")
    EXTERNAL_VERSION_FROM_EXT_SOURCE = os.environ.get("EXTERNAL_VERSION_FROM_EXT_SOURCE",
"1001000")
    # ------------------------------------------------------------------

    try:
        if 'body' not in event:
            return {
                'statusCode': 400,
                'body': json.dumps({'message': 'Missing "body" in event. Please provide the JSON
payload in the body.'})
            }

        input_data = json.loads(event['body'])
        #print(f"Parsed input data: {json.dumps(input_data)}")

        protected_results = []

        if "assessments" in input_data and isinstance(input_data["assessments"], list):
            for assessment in input_data["assessments"]:
                if "sensitiveInformationPolicy" in assessment and \
                   "piiEntities" in assessment["sensitiveInformationPolicy"] and \
                   isinstance(assessment["sensitiveInformationPolicy"]["piiEntities"], list):

                    pii_entities = assessment["sensitiveInformationPolicy"]["piiEntities"]

                    for entity in pii_entities:
                        sensitive_value = entity.get("match")
                        entity_type = entity.get("type")

                        if sensitive_value:
                            crdp_payload = {
                                "protection_policy_name": PROTECTION_PROFILE,
                                DATA_KEY: sensitive_value,
                            }

                            url_str = f"http://{CRDP_IP}:8090/v1/{MODE}"
                            headers = {"Content-Type": "application/json"}
```

```
                            try:
                                response = requests.post(
                                    url_str, headers=headers, data=json.dumps(crdp_payload),
timeout=10
                                )
                                response.raise_for_status()
                                response_json = response.json()
                                protected_results.append({
                                    "original_value": sensitive_value,
                                    "type": entity_type,
                                    "protection_response": response_json
                                })
                            except requests.exceptions.Timeout:
                                error_message = f"Request to CRDP timed out for value:
'{sensitive_value}'"
                                print(f"Error: {error_message}")
                                protected_results.append({
                                    "original_value": sensitive_value,
                                    "type": entity_type,
                                    "error": error_message
                                })
                            except requests.exceptions.RequestException as e:
                                error_message = f"Error making request to CRDP for value
'{sensitive_value}': {e}"
                                print(f"Error: {error_message}")
                                protected_results.append({
                                    "original_value": sensitive_value,
                                    "type": entity_type,
                                    "error": error_message
                                })
                        else:
                            print(f"Skipping entity due to missing 'match' field:
{json.dumps(entity)}")
                else:
                    print("sensitiveInformationPolicy or piiEntities not found or not a list in
assessment.")
        else:
            print("assessments not found or not a list in input data.")

        return {
            'statusCode': 200,
            'body': json.dumps({
                'message': 'PII protection process completed.',
                'protection_results': protected_results
            })
        }

    except json.JSONDecodeError as e:
        print(f"JSON decoding error: {e}")
        return {
            'statusCode': 400,
            'body': json.dumps({'message': f'Invalid JSON format in request body: {e}'})
        }
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return {
            'statusCode': 500,
            'body': json.dumps({'message': f'An internal server error occurred: {e}'})
        }
```

For more information on how to leverage AWS guardrails with Thales CipherTrust Manager see blog post at: ?????

Protecting sensitive fields in in file

As mentioned earlier it is estimated about 80% of data in an organization is unstructured files such as pdf text and csv files. It is important to convert the sensitive data in those files before the content is used in a vector database or used to fine train a model. Here is an example of how this can be accomplished using the Thales Application Encryption along with the AWS Comprehend API to find sensitive data in a file.

```
import java.io.*;
import java.util.Properties;

import software.amazon.awssdk.regions.Region;

/**
 * This class serves as a batch processor for protecting and revealing data using Thales Protect
Reveal CADP.
 * It reads configuration from a properties file, processes files in a specified directory,
 * and performs either protection or revelation based on the provided mode.
 */
public class ThalesAWSProtectRevealCADPBatchProcessor {

	// Defines the AWS region to be used. This can be changed to your desired region.
	private static final Region REGION = Region.US_EAST_2;

	// Properties object to hold configurations loaded from the properties file.
	private static Properties properties;

	/**
	 * Static initializer block to load properties from "thales-config.properties" file.
	 * This block is executed once when the class is loaded.
	 * It throws a RuntimeException if the file is not found or an error occurs during
loading.
	 */
	static {
		try (InputStream input =
ThalesAWSProtectRevealCADPBatchProcessor.class.getClassLoader()
				.getResourceAsStream("thales-config.properties")) {
			properties = new Properties();
			if (input == null) {
				// Throws an exception if the properties file is not found.
				throw new RuntimeException("Unable to find udfConfig.properties");
			}
			// Loads properties from the input stream.
			properties.load(input);

		} catch (Exception ex) {
			// Catches any exception during properties loading and re-throws as a
RuntimeException.
			throw new RuntimeException("Error loading properties file", ex);
		}
	}

	/**
	 * The main method, serving as the entry point of the application.
	 * It processes command-line arguments, initializes necessary objects,
	 * and orchestrates the protection or revelation of files.
	 * @param args Command-line arguments: [mode] [inputDirectory] [outputDirectory]
[fileExtension]
	 * @throws IOException If an I/O error occurs during file processing.
	 */
	public static void main(String[] args) throws IOException {
		// Records the start time of the application for performance measurement.
		long startTime = System.nanoTime();

		// Flag to indicate whether to skip header rows in files.
		boolean skiphdr = false;
		// Cloud Service Provider, currently set to AWS.
```

```
                String csp = "AWS";
                // Counter for the number of records processed in the current file.
                int nbrofrecords = 0;
                // Counter for the total number of records processed across all files.
                int totalnbrofrecords = 0;

                // Retrieves various configuration properties from the loaded properties object.
                String metadata = properties.getProperty("METADATA");
                String crdpip = properties.getProperty("CRDPIP");

                String keymanagerhost = properties.getProperty("KEYMGRHOST"); // Unused in
provided code.
                String crdptkn = properties.getProperty("CRDPTKN"); // Unused in provided code.
                String policyType = properties.getProperty("POLICYTYPE");
                String showmetadata = properties.getProperty("SHOWMETADATA");
                String revealuser = properties.getProperty("REVEALUSER");
                System.out.println(" user  " + revealuser);

                String defaultpolicy = properties.getProperty("DEFAULTPOLICY");
                // Converts the "SHOWMETADATA" property to a boolean.
                boolean showmeta = showmetadata.equalsIgnoreCase("true");

                // Initializes ThalesProtectRevealHelper, responsible for interacting with the
Thales Protect Reveal service.
                ThalesProtectRevealHelper tprh = new ThalesRestProtectRevealHelper(crdpip,
metadata, policyType, showmeta);

                // Sets properties for the ThalesProtectRevealHelper instance.
                tprh.revealUser = revealuser;
                tprh.policyType = policyType;
                tprh.policyName = properties.getProperty("POLICYNAME");
                tprh.defaultPolicy = defaultpolicy;

                // Initializes an AWSContentProcessor if the CSP is AWS.
                AWSContentProcessor cp = null;
                if (csp.equalsIgnoreCase("aws"))
                        cp = new AWSContentProcessor(properties);

                // Sets the AWS region for the content processor.
                cp.awsregion = REGION;

                // Declare File objects for input and output directories
                File inputDir = null;
                File outputDir = null;

                // Parse command-line arguments
                String mode = args[0]; // "protect" or "reveal" mode
                inputDir = new File(args[1]); // Path to the input directory
                outputDir = new File(args[2]); // Path to the output directory
                String fileextension = args[3]; // File extension to process (e.g., ".pdf",
".txt")
                // Counter for the number of files processed.
                int nbroffiles = 0;

                // Get a list of files in the input directory that match the specified file
extension.
                File[] inputFiles = inputDir.listFiles((dir, name) ->
name.toLowerCase().endsWith(fileextension));

                File outputFile = null;
                // Checks if any input files were found.
                if (inputFiles != null) {

                        // Iterates through each input file.
                        for (File inputFile : inputFiles) {

                                // Determines the output file name based on the mode (protect or
reveal).
                                if (mode.equalsIgnoreCase("protect"))
                                        outputFile = new File(outputDir, "protected" +
inputFile.getName());
```

```
                                else
                                        outputFile = new File(outputDir, "revealed" +
inputFile.getName());

                                // Processes the file using the appropriate content processor (AWS
in this case).
                                // The processFile method performs either protection or revelation.
                                if (csp.equalsIgnoreCase("aws"))
                                        nbrofrecords = cp.processFile(inputFile, outputFile, null,
tprh, mode, skiphdr);

                                // Increments the count of processed files.
                                nbroffiles++;
                        }
                        // Updates the total number of records processed.
                        totalnbrofrecords = totalnbrofrecords + nbrofrecords;
                } else {
                        // Prints a message if no files with the specified extension are found.
                        System.out.println("No " + fileextension + " files found in the directory:
" + inputDir.getAbsolutePath());
                }

                // Records the end time in nanoseconds.
                long endTime = System.nanoTime();

                // Calculates the elapsed time in nanoseconds.
                long elapsedTimeNano = endTime - startTime;

                // Converts nanoseconds to seconds for better readability.
                double elapsedTimeSeconds = (double) elapsedTimeNano / 1_000_000_000.0;

                // Prints the application's execution time.
                System.out.printf("Application execution time: %.6f seconds%n",
elapsedTimeSeconds);

                // Prints summary statistics.
                System.out.println("Number of files = " + nbroffiles);
                System.out.println("Total nbr of records = " + totalnbrofrecords);
                System.out.println("Total skipped entities = " + cp.total_skipped_entities);
                System.out.println("Total entities found = " + cp.total_entities_found);

        }

}
```

To view the entire solution see the following github repository: ???

Prompt Protection

This java example uses a couple of helper classes, ThalesProtectRevealHelper to control settings for the CiphterTrust Manager and associated profile information and ContentProcessor to control settings for the content to be protected. To view the entire solution see the following github repository: ???

```
import java.io.InputStream;
import java.util.Properties;

// AWS SDK imports for Bedrock Runtime, Comprehend, credentials, and regions
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.bedrockruntime.BedrockRuntimeClient;
import software.amazon.awssdk.services.bedrockruntime.model.ContentBlock;
import software.amazon.awssdk.services.bedrockruntime.model.ConversationRole;
```

```
import software.amazon.awssdk.services.bedrockruntime.model.ConverseResponse;
import software.amazon.awssdk.services.bedrockruntime.model.Message;
import software.amazon.awssdk.services.comprehend.ComprehendClient;

/**
 * ThalesAWSBedrockPromptExample demonstrates how to integrate AWS Bedrock
 * with a Thales data protection solution to process prompts containing PII.
 * It detects PII using AWS Comprehend, protects it using Thales's encryption,
 * and then simulates sending it to a Bedrock model (commented out in this example)
 * before revealing (decrypting) the data.
 */
public class ThalesAWSBedrockPromptExample {

	// Define the AWS region for Bedrock and Comprehend clients
	private static final Region REGION = Region.US_EAST_2; // Change to your desired region

	// Properties object to load configuration from a file
	private static Properties properties;

	// Static initializer block to load properties from 'thales-config.properties'
	static {
		try (InputStream input = ThalesAWSBedrockPromptExample.class.getClassLoader()
				.getResourceAsStream("thales-config.properties")) {
			properties = new Properties();
			if (input == null) {
				// Throw a runtime exception if the properties file is not found
				throw new RuntimeException("Unable to find udfConfig.properties");
			}
			// Load properties from the input stream
			properties.load(input);

		} catch (Exception ex) {
			// Catch any exceptions during properties loading and re-throw as a
runtime exception
			throw new RuntimeException("Error loading properties file", ex);
		}
	}

	/**
	 * Main method to run the example.
	 *
	 * @param args Command line arguments (not used in this example).
	 */
	public static void main(String[] args) {

		// Build a BedrockRuntimeClient using default credential provider and the
specified region
		BedrockRuntimeClient client = BedrockRuntimeClient.builder()

	.credentialsProvider(DefaultCredentialsProvider.create()).region(REGION).build();

		// Define the Bedrock model ID to be used (though Bedrock call is commented out)
		String modelId = "amazon.titan-embed-text-v2:0";
		// Retrieve various configuration properties from the loaded properties file
		String metadata = properties.getProperty("METADATA");
		String crdpip = properties.getProperty("CRDPIP");
		String keymanagerhost = properties.getProperty("KEYMGRHOST"); // Note:
keymanagerhost seems is if using ThalesRestProtectRevealHelper
		String crdptkn = properties.getProperty("CRDPTKN"); // Note: crdptkn is unused if
using ThalesRestProtectRevealHelper
		String policyType = properties.getProperty("POLICYTYPE");
		String showmetadata = properties.getProperty("SHOWMETADATA");
		String revealuser = properties.getProperty("REVEALUSER");
		System.out.println(" user  " + revealuser); // Print the reveal user

		// Convert showmetadata string to a boolean
		boolean showmeta = showmetadata.equalsIgnoreCase("true");

		// Initialize a ThalesProtectRevealHelper. This example uses
ThalesRestProtectRevealHelper,
		// implying a REST-based integration with the Thales product.
```

```
                ThalesProtectRevealHelper tprh = new ThalesRestProtectRevealHelper(crdpip,
metadata, policyType,
                                showmeta);

                // Set reveal user, policy type, and policy name on the Thales helper object
                tprh.revealUser = revealuser;
                tprh.policyType = policyType;
                tprh.policyName = properties.getProperty("POLICYNAME");

                // Create an instance of AWSContentProcessor, passing in the loaded properties
                AWSContentProcessor cp = new AWSContentProcessor(properties);

                // Set the AWS region for the Content Processor (used by Comprehend within it)
                cp.awsregion = REGION;

                // Build a ComprehendClient using the specified region
                ComprehendClient comprehendClient =
ComprehendClient.builder().region(REGION).build();

                // Define an example input text containing PII
                String inputText = "Explain 'rubber duck debugging' in one line to Kathi Wolf
        Burney Circle  Mertzmouth     OH      24114   328.639.0623x3743
        becker.drury@hotmail.com      7/7/1992        5.32755E+15    15      203-04-0501\r\n";
                System.out.println("Original prompt = " + inputText);

                // Process the input text to detect and protect PII using AWSContentProcessor
                inputText = cp.processText(inputText, tprh);
                System.out.println("Protected prompt = " + inputText);

                // The following Bedrock API call is commented out, indicating it's not
                // actively executed in this example, but shows the intended flow.
                /*
                 * Message message =
                 * Message.builder().content(ContentBlock.fromText(inputText)).role(
                 * ConversationRole.USER) .build(); ConverseResponse response =
                 * client.converse(request -> request.modelId(modelId).messages(message));
                 * String responseText = response.output().message().content().get(0).text();
                 * System.out.println(responseText); System.out.println("new string = " +
                 * inputText);
                 */

                // Close the Comprehend client to release resources
                comprehendClient.close();

                // Test decryption example: decrypt the previously protected data
                String decrypteddata = cp.decryptLine(inputText, tprh);
                System.out.println("decrypteddata = " + decrypteddata);
        }
}
```

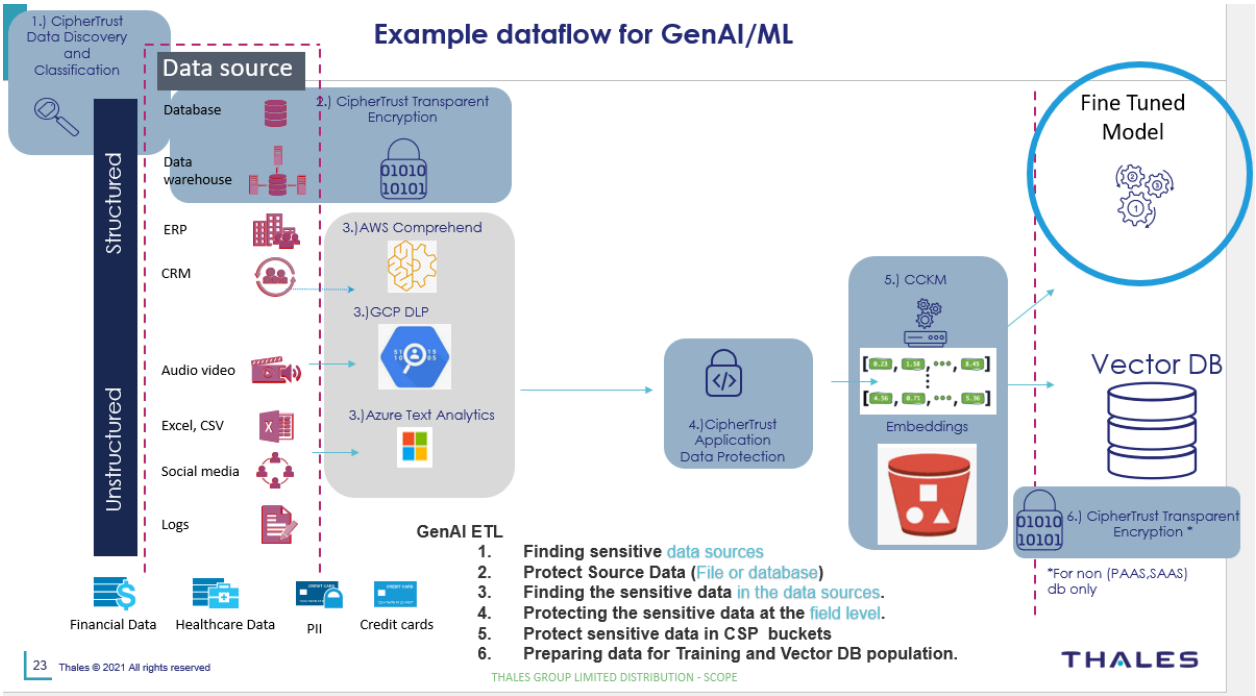
Output:
This is the output in testing mode without the Bedrock calls.

```
new string = Explain 'rubber duck debugging' in one line to RU5DY2hhcg==-17:1001001DBrtA xEc4
        RU5DY2hhcg==-40:1001001aQGtZI q1Ml6a  TzRpKn13XE     VB      vw2em   RU5DbmJy-
24:1001001817.646.7874x7333   RU5DY2hhcg==-31:1001001mCRkY7.vfa7J@BGKCOjv.HKN      RU5DY2hhcg==-
15:1001001B/u/8sLh     RU5DY2hhcg==-33:1001001M.bK4FKG+8S    4S      uGO-En-kSal


decrypteddata = Explain 'rubber duck debugging' in one line to Kathi Wolf  Burney Circle
        Mertzmouth     OH      24114   328.639.0623x3743      becker.drury@hotmail.com
        7/7/1992       5.32755E+15    15      203-04-0501
```

Overall workflow

The diagram below shows the entire process




```
Here are some samples that can be used to implement the above.

1.) Run Thales DDC or other datasoruce discovery tool to find datasources that have sensitive
data.
2.) Use Thales CTE to protect those data sources.
3.) Run Thales DDC Report to export a list of datasources with sensitive data.
4.) If any of the sources are of file types excel,word,pdf run sample code
ExtractFilesFromDDCReport.java to generate a list of the datsources that need to be converted to
text.
5.) Run sample code ConvertToTextBatchProcessor.java to convert files found in step for to text
version.
6.) Depending on customer preference for CSP (AWS,GCP,Azure) run appropriate sample code
ThalesCSPProtectRevealBatchProcessor.java to find sensitive fields in datasources and create a
new output with sensitive data protected.
7.) Now load protected data into appropriate AWS,GCP, Azure training data upload.
```

Thales Protect/Reveal Options

The solutions provide in this document for field level protection are based on two application development offerings from Thales CRDP and CADP for Java.

1.)CRDP

CipherTrust RESTful Data Protection (CRDP) is a RESTful webservice that protects sensitive data using encryption, tokenization, or data generalization. CRDP is designed from the ground up to seamlessly fit with existing cloud and on-premise applications. CRDP is deployed as a container and performs a wide

range of data protection functions on the sensitive data including data masking, pseudonymization, and redaction. CRDP is easy to deploy, configure, manage, and migrate.
https://thalesdocs.com/ctp/con/crdp/latest/index.html

2.) CipherTrust Application Data Protection for Java (CADP for Java)

Is a Java Cryptography Extension (JCE) provider that enables users to integrate the Key Manager's capabilities into their Java applications. CADP for Java is available in two variants:

- **Centrally Managed APIs**
- **Traditional APIs**

The examples provided in this document are based on Centrally Managed APIs.

This is our new set of APIs available from CADP for Java 8.18.1 release. CADP for Java offers simplified APIs (Protect/Reveal) to perform cryptography operations. CADP for Java integrates seamlessly with **Application Data Protection** on CipherTrust Manager, enabling organizations to centrally configure, manage, and enforce data-centric cryptographic policies in a reusable, human-readable format, providing flexibility and ease of management.
Protection policy (ciphers, keys, IV, tweak, and so on) defines how to protect the sensitive data. Whereas, Access policy determines who can view sensitive data and how (plaintext, ciphertext, custom masking format or redacted).
Centrally Managed APIs offer several advantages over traditional APIs, including:

- Developers do not need to understand cryptographic parameters (cipher, key, IV, Nonce, Tweak, and so on) to protect data as the Data Security Admins are responsible to handle configurations and policies.
- Each deployed application with CADP for Java is visible on CipherTrust Manager (providing a Single Pane of Glass view).
- Data Security Admins gain Crypto Agility, enabling real time changes to cipher, keys, and parameters.

https://thalesdocs.com/ctp/con/cadp/cadp-java/latest/admin/index.html#centrally-managed-apis

The code provided in this document provides examples using both the CRDP and CADP for Java Centralized API's. As you can see from the code there are two helper classes that make it easy to switch between either api.

```
// Initialize ThalesProtectRevealHelper. Two options are commented out, showing different
// implementations (CADP vs. REST). The REST implementation is currently active.
// ThalesProtectRevealHelper tprh = new ThalesCADPProtectRevealHelper(keymanagerhost, crdptkn,
null, policyType,showmeta);

// Using ThalesRestProtectRevealHelper for content protection and revelation.
ThalesProtectRevealHelper tprh = new ThalesRestProtectRevealHelper(crdpip, metadata,
policyType,showmeta);
```

Note: For each option there will be different properties required please refer to the code samples for detailed explanations.