LUMENDATA

Data Sheet

The Guide to Building a Custom Databricks Notification System

5201 GREAT AMERICAN PARKWAY, SUITE 320
SANTA CLARA, CA 95054
Tel: (855) 695-8636
E-mail: info@lumendata.com
Website: www.lumendata.com

This data sheet talks about solutions that can create a more informative notification system for Databricks workloads. It explains how to build a custom Databricks notification system using Python and the Databricks API. The techniques discussed in the sheet can help transform Databricks alerting for effective job monitoring and improved workflow efficiency.

Notification Alerting in Databricks:

- **Utilize Databricks REST API:** Databricks offers a REST API that allows you to integrate with various notification channels. We can develop custom scripts that leverage the API to send alerts to Slack, PagerDuty, or any other supported channel upon job failures.
- **Databricks notebooks for alerting:** Create Databricks notebooks specifically for monitoring purposes. These notebooks can query the Databricks job history API and other relevant endpoints to identify issues and trigger custom notifications using the REST API calls.
- **Custom alert conditions:** While built-in options are limited, explore using Databricks python-based scripts within notebooks to define custom alert conditions. This allows for more granular alerts based on specific job metrics or failures.
- **Utilize Databricks Widgets:** Databricks widgets can be used to create custom dashboards that display job health and other relevant metrics. These dashboards can be configured to trigger alerts based on certain conditions.
- **Third-party integrations:** Consider third-party tools that integrate with Databricks and offer more advanced notification functionalities. These tools can provide features like customizable notifications, priority levels, and automated workflows based on alerts.

Notification Alerting in Databricks:

At Lumendata, we built a custom solution utilizing the platform's robust REST API. This API grants programmatic access, allowing the construction of a more granular alerting system.

- **Modular Design with Alert-Specific Notebooks:** We developed a modular design, employing separate notebooks specifically tailored for alert generation. These notebooks interact with the Databricks job history API and potentially other relevant endpoints to extract critical data.
- **Custom Metrics and Dynamic HTML Alerts:** Beyond basic success/failure notifications, our solution empowers the definition of custom alert conditions using Databricks python-based script within the notebooks. This enables us to generate alerts based on specific job metrics or tailored failure scenarios. This approach goes beyond the limitations of basic email notifications by generating comprehensive, HTML-formatted alert emails. These informative reports provide a clear view of the health of your recent Databricks jobs, like a comprehensive operational report.

Job Run Summary Table: The email body features a table containing details about the 5 most recent Databricks job runs. Each table row captures the following attributes:

- **Run ID:** Unique identifier for the job run.
- **Job ID:** Unique identifier for the Databricks job.
- **Run State:** Indicates whether the job run was successful (highlighted in green) or failed (highlighted in red).
- **Run Name:** User-defined name assigned to the Databricks job.
- **User Name:** Username of the individual who triggered the job run.
- **Run Duration:** Total time taken by the job run to execute.
- **Start Time:** Timestamp representing when the job run commenced.
- **End Time:** Timestamp representing when the job run concluded.

Hi Team,
This email provides a summary of the latest 5 job runs in our Databricks workspace.

Run ID	Job ID	Run State	Run Name	User Name	Run Duration	Start Time	End Time
586876226895181	1121560712770720	SUCCESS	test_job_ritesh_multiple_tasks	ritesh.chidrewar@lumendata.com	0.472	2024-02-12 17:11:10.740000	2024-02-12 17:11:39.042000
965759163430842	1121560712770720	SUCCESS	test_job_ritesh_multiple_tasks	ritesh.chidrewar@lumendata.com	0.414	2024-02-12 17:10:38.645000	2024-02-12 17:11:03.514000
117647760130479	1121560712770720	FAILED	test_job_ritesh_multiple_tasks	ritesh.chidrewar@lumendata.com	0.335	2024-02-12 16:40:31.854000	2024-02-12 16:40:51.954000
963439973253933	487220538123195	SUCCESS	test_job_ritesh	ritesh.chidrewar@lumendata.com	0.186	2024-02-12 16:01:42.129000	2024-02-12 16:01:53.282000
49751404785932	487220538123195	SUCCESS	test_job_ritesh	ritesh.chidrewar@lumendata.com	0.466	2024-02-12 16:01:10.186000	2024-02-12 16:01:38.162000
1003318758876806	487220538123195	FAILED	test_job_ritesh	ritesh.chidrewar@lumendata.com	0.201	2024-02-09 05:09:16.283000	2024-02-09 05:09:28.338000

Note:
* Run state is highlighted in red for failed runs and green for successful runs.

* Below Table lists the failure run_id's and error message along with run page URL.

Run ID	Job ID	Error Message	Run URL
117647760130479	1121560712770720	Task display_notebook_2 failed with message: Workload failed, see run output for details. This caused all downstream tasks to get skipped.	117647760130479 URL
1003318758876806	487220538123195	Workload failed, see run output for details	1003318758876806 URL

- **Failure Details:** The email also includes a section dedicated to failed jobs. It lists the Run ID, Job ID, error message, and a link to the corresponding run page URL for each failed job run.
- This email notification serves as a concise report providing insights into the execution status of recent Databricks jobs. It facilitates monitoring job health and pinpointing potential failures that necessitate further investigation.

NOTE: The JOB-RUNS are configurable through code. For testing and example purposes we have hardcoded the value to last 5 Job runs.

- **Code Walkthrough and Workflow:** Import necessary python libraries required in databricks NB. Select Notebook language as python.

```
import requests
import json
import datetime
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
import sys
```

Overall Workflow:

1. The script initialization.
2. It retrieves Databricks access token and workspace URL.
3. Calls helper functions to validate credentials, configure the API request, and define the number of jobs and job runs to fetch.
4. Fetches details about the most recent job runs from the Databricks API.
5. Finally, it sends an email notification with a summary of the retrieved job run details.

```
if __name__ == "__main__":
    print("program to fetch last 5 job runs and send an email alert...")

    try:
        #setup databricks api configuration
        #databricks access token is personal access token used for authentication -> we can get this from Databricks workspace->User Settings -> Developer -> Access tokens -> Manage ->g
        token
        databricks_access_token = "[redacted]"

        #databricks isntance is workspace url ex::  https://<instance-name>.cloud.databricks.com
        databricks_instance = "[redacted]cloud.databricks.com"
        api_endpoint, api_token,api_endpoint_error_message = setup_databricks_api_token_and_endpoint(databricks_access_token,databricks_instance)
        #print(api_endpoint,api_token)

        #setup setup_databricks_job_api_config , please pass the num_of jobs to be fetched
        global num_of_jobs,num_of_job_runs
        #set below variable value to fetch no.of jobs
        num_of_jobs = 5
        #set below variable to fetch number of job_runs for specific job/job_id
        num_of_job_runs = 5
        url,headers,payload = setup_databricks_job_api_config(api_endpoint,api_token)
        print(url,headers,payload)

        #fetch the job run details
        list_of_extracted_job_runs = fetch_databricks_job_runs(url,headers,payload)

        #send email notification
```

Databricks API Configuration:

- These lines set up the credentials needed to interact with the Databricks REST API.
- **databricks_access_token**: This variable stores personal access token, which grants programmatic access to Databricks workspace. We can obtain this token from your Databricks settings.
- **databricks_instance**: This variable holds the URL of our Databricks workspace.

Function Calls:

- **setup_databricks_api_token_and_endpoint**: This function validates the access token and workspace URL and returns the API endpoint and any error messages encountered.
- **setup_databricks_job_api_config**: This function constructs the API URL, headers, and payload (data) required to fetch job run details. It also defines two global variables:
- **num_of_jobs**: This variable sets the maximum number of jobs to retrieve (currently set to 5).
- **num_of_job_runs**: This variable specifies the maximum number of runs to fetch per retrieved job (currently set to 5).
- **fetch_databricks_job_runs**: This function sends an API request to Databricks and retrieves information about the most recent job runs based on the configured parameters.
- **send_email_notification**: This function takes the list of retrieved job runs and a list of recipient email addresses and sends an email notification summarizing the job run details.

Error Handling:

- The try-except block attempts to execute the main code. If any exceptions (errors) occur, the except block catches the error message, prints it, and then raises the exception for further handling.

Function Name: setup_databricks_api_token_and_endpoint

```python
def setup_databricks_api_token_and_endpoint(databricks_access_token,databricks_instance):
    print("setting up api token and endpoint for databricks jobs api...")
    try:
        #Set your Databricks API endpoint
        print("setting up api_endpoint using databricks_instance...")
        api_endpoint = "https://{databricks_instance}/api/2.1".format(databricks_instance=databricks_instance)
        print("api_endpoint::"+api_endpoint)

        #below api endpoint fetches the error message for the failed job runs
        print("setting up the api_endpoint to fetch failed job runs error messages...")
        #using /api/2.0/  instead of 2.1
        api_endpoint_for_error_message = "https://{databricks_instance}/api/2.0".format(databricks_instance=databricks_instance)
        print("api_endpoint_for_error_message::"+api_endpoint_for_error_message)

        # Set your Databricks token
        print("setting up api_token using databricks_access_token...")
        api_token = "{databricks_access_token}".format(databricks_access_token=databricks_access_token)
        print("api_token::"+api_token)

        print("setup_databricks_api_token_and_endpoint success!!")

        return api_endpoint, api_token,api_endpoint_for_error_message
    except Exception as e:
        print("Error setting up databricks_api_token_and_endpoint ...please check the configurations/arguments passed!!\n"+str(e))
        raise Exception("Error setting up databricks_api_token_and_endpoint ...please check the configurations/arguments passed!!\n"+str(e))
```

Purpose:

- Validates and prepares the Databricks API endpoint and authentication token for subsequent API calls.
- Handles potential errors during setup.

Key Steps:

1. Constructs the base API endpoint using the provided Databricks workspace URL. This forms the starting point for constructing specific API calls.
2. Creates a separate API endpoint specifically for retrieving error messages associated with failed job runs. This endpoint uses a different API version for compatibility with the error message retrieval functionality.
3. Sets up the Databricks authentication token using the provided personal access token. This token is included in API requests to ensure authorized access.
4. Returns three values:
    - **api_endpoint**: The base API endpoint for Databricks API calls.
    - **api_token**: The Databricks authentication token for API access.
    - **api_endpoint_for_error_message**: The dedicated endpoint for retrieving error messages of failed job runs.

Function: setup_databricks_job_api_config

```
def setup_databricks_job_api_config(api_endpoint,api_token):

    try:
        #Set the number of job runs you want to fetch
        num_runs = num_of_jobs
        print("fetching latest::"+str(num_runs)+" job runs..")

        #Build the API request URL
        print("builing rest api url...")
        url = "{api_endpoint}/jobs".format(api_endpoint=api_endpoint)
        print("job run api url::"+url)

        #Set the request headers
        headers = {
            "Authorization": "Bearer {api_token}".format(api_token=api_token),
            "Content-Type": "application/json"
        }

        print("headers::")
        print(headers)

        #set request payload
        payload=({
            "limit":num_runs
        })

        print("payload::")
        print(payload)

        return url,headers,payload
    except Exception as e:
        print("Error setting up databricks_job_api_config ...please check the configurations/arguments passed!!\n"+str(e))
        raise Exception("Error setting up databricks_job_api_config ...please check the configurations/arguments passed!!\n"+str(e))
```

Purpose:

- Orchestrates the setup of necessary components for interacting with the Databricks Jobs API.
- Constructs the API request URL, headers, and payload for fetching job run details.

Steps:

1. Fetches the number of job runs to retrieve:
    - Retrieves the num_of_jobs global variable (set in the main code) to determine the intended number of job runs to fetch.
2. Builds the API request URL:
    - Formats the API endpoint provided as input to construct a complete URL for the Jobs API (https://<workspace-url>/api/2.1/jobs).
    - Prints the generated URL for debugging purposes.
3. Sets the request headers:
    - Constructs a dictionary containing authentication and content type headers:
        - Authorization: Embeds the Databricks access token using the Bearer schema for authentication.

- Content-Type: Specifies that the request payload is formatted as JSON.

4. Sets the request payload:
   - Creates a JSON-formatted payload with a single key-value pair:
     - limit: Stores the number of job runs to retrieve as specified in num_runs.
   - Prints the payload for debugging purposes.
5. Returns the API components:
   - Returns a tuple containing the formatted URL, headers, and payload to be used for subsequent API calls.

Function: fetch_databricks_job_runs

```
def fetch_databricks_job_runs(url,headers,payload):
    print("fetching latest jobs and their run details...")

    try:
        #send the api_request
        #below api fetches only latest jobs
        url_jobs = url+"/list"
        response = requests.get(url_jobs, headers=headers, json=payload)
        print(response.status_code)

        #convert reesponse to json format
        response_json = response.json()

        #extract jobs and their id's from the response json
        jobs = response_json["jobs"]
        print(jobs)

        extracted_job_run_details_dict = {}
        list_of_extracted_job_runs = []

        for job in jobs:
            print("extracting run details for job::"+str(job["job_id"]))
            #below api fetches job runs for specific job
            url_job_runs = url+"/runs/list?job_id={job_id}".format(job_id=job["job_id"])
            print("url_job_runs::"+url_job_runs)

            #set request payload -> set num of runs to be fetched for specific job/job_id
            job_run_payload={{"limit":num_of_job_runs}}
            response_job_runs = requests.get(url_job_runs, headers=headers, json=job_run_payload)
            print("response_job_runs status_code::"+str(response_job_runs.status_code))

            if response_job_runs.status_code==200:
                response_job_runs_json = response_job_runs.json()
                job_runs = response_job_runs_json["runs"]
```

```
                #extracting the details from job runs
                for run in job_runs:
                    print(f"Run ID: {run['run_id']}")
                    extracted_job_run_details_dict["run_id"] = run['run_id']

                    print(f"Job ID: {run['job_id']}")
                    extracted_job_run_details_dict["job_id"] = run['job_id']

                    print(f"Run State: {run['state']['result_state']}")
                    extracted_job_run_details_dict["run_state"] = run['state']['result_state']

                    #fetch the error message and run page url for the dailed job run id
                    if run['state']['result_state'] == "FAILED":
                        error_message,run_page_url = fetch_databricks_failed_job_runs_error_log(run['job_id'],run['run_id'])
                        extracted_job_run_details_dict["error_message"] = error_message
                        extracted_job_run_details_dict["run_page_url"] = run_page_url

                    extracted_job_run_details_dict["run_name"] = run["run_name"]
                    print(f"Job Name: {run['run_name']}")

                    extracted_job_run_details_dict["creator_user_name"] = run['creator_user_name']
                    print(f"User Namee: {run['creator_user_name']}")

                    extracted_job_run_details_dict["run_duration"] = run['run_duration']
                    print(f"Duration: {run['run_duration']}")

                    #TODO->convert this to from ms to timestamp
                    print(f"Run Start Time: {run['start_time']}")
                    extracted_job_run_details_dict["run_start_time"] = run['start_time']

                    print(f"Run End Time: {run['end_time']}")
                    extracted_job_run_details_dict["run_end_time"] = run['end_time']

                    #appending extracted details to list
                    list_of_extracted_job_runs.append(extracted_job_run_details_dict.copy())
                    print(list_of_extracted_job_runs)
                    print("----------")
```

```
                print(len(list_of_extracted_job_runs))

            else:
                print("unable to fetch job runs for the job_id::"+str(job["job_id"])+" response status code::"+str(response_job_runs.status_code))
                print("Error fetching job run..!!Please recheck the job id and configs!!")

        return list_of_extracted_job_runs

    except Exception as e:
        print("Error fetching latest job runs through api...Please recheck the configs!!\n"+str(e))
        raise Exception("Error fetching latest job runs through api...Please recheck the configs!!\n"+str(e))
```

Purpose:

- This function retrieves detailed information about the most recent Databricks job runs. It leverages the Databricks Jobs REST API to fetch data and returns a list containing extracted details for each retrieved job run.

Steps:

1. API Configuration:
   - The function takes three arguments: url, headers, and payload. These represent the pre-configured API endpoint URL, authentication headers, and any additional data required for the request.
2. Fetching Job List:
   - It constructs a URL to query for a list of jobs (url+"/list").
   - It sends a GET request using the requests library and retrieves the response.
   - It checks the response status code (ideally 200 for success).
   - If successful, it parses the JSON response to extract details about available jobs, including their IDs.
3. Extracting Job Run Details (if successful):
   - If the job run details retrieval is successful (status code 200), it parses the JSON response to extract details for each run, including:
     - Run ID
     - Job ID
     - Run State (Success/Failure)
     - Run Name (if available)
     - User Name who triggered the run (if available)
     - Run Duration
     - Run Start Time (in milliseconds)
     - Run End Time (in milliseconds)
   - For failed runs, it calls another helper function to retrieve the error message and run page URL.
4. Building and Returning Results:
   - The function creates a dictionary (extracted_job_run_details_dict) to temporarily store the extracted details for each run within the current job.
   - It iterates through each retrieved run and populates the temporary dictionary with details.
   - It appends a copy of the populated dictionary to a list (list_of_extracted_job_runs) to store details for all runs across all jobs.

After processing all jobs and their runs, the function returns the final list containing extracted details for each retrieved job run.

Function: fetch_databricks_failed_job_runs_error_log

```python
def fetch_databricks_failed_job_runs_error_log(job_id,run_id):
    print("fetching error log for the failed job_run::"+str(run_id))
    try:
        #get endpoint for error message, databricks instance and token
        api_endpoint, api_token,api_endpoint_for_error_message = setup_databricks_api_token_and_endpoint(databricks_access_token,databricks_instance)

        #Build the API request URL
        print("builing rest api url...")
        url_for_error_message = "{api_endpoint_for_error_message}/jobs/runs/get?run_id={job_run_id}".format(api_endpoint_for_error_message=api_endpoint_for_error_message,job_run_id=run_id)
        print("api_endpoint_for_error_message::"+api_endpoint_for_error_message)

        #Set the request headers
        headers = {
            "Authorization": "Bearer {api_token}".format(api_token=api_token)
        }

        print("headers::")
        print(headers)

        response = requests.get(url_for_error_message, headers=headers)
        response_json = response.json()

        if response.status_code == 200:
            # Check the run_status field to see if the job run failed
            if response_json['state']['result_state'] == 'FAILED':
                # Get the error message
                error_message = response_json["state"]["state_message"]
                print(f"Error message: {error_message}")
                run_page_url = response_json["run_page_url"]
                print(f"Run Page URL: {run_page_url}")

                return error_message,run_page_url
            else:
                print("The job run is not failed.")
                return 'Error message not found','NA'
        else:
            print(f"Failed to retrieve the job run output. Status code: {response.status_code}")
```

Purpose:

This function retrieves specific information about failed Databricks job runs, specifically the error message and run page URL. It's designed to be called within the fetch_databricks_job_runs function to gather additional details for failed runs.

Steps:

1. API Configuration:
   - The function takes two arguments: job_id and run_id. These represent the unique identifiers for the failed job and its run.
   - It calls another function (setup_databricks_api_token_and_endpoint), to retrieve and set up necessary API configuration details like the endpoint URL and authentication token.
2. Building API Request:
   - It constructs a specific API URL to query details for the failed job run ({api_endpoint_for_error_message}/jobs/runs/get?run_id={job_run_id}).
   - It sets up authentication headers with the Databricks access token.
3. Sending API Request:
   - It sends a GET request using the requests library to retrieve the API response.
4. Processing Response:
   - It checks the response status code for success (200).

- If successful, it parses the JSON response.
  - It confirms that the run state is indeed "FAILED".
  - If confirmed, it extracts the error message from the state_message field.
  - It also extracts the run page URL from the run_page_url field.
- If the run is not marked as failed, it returns a placeholder message indicating that no error message was found.
- If the API request itself fails (status code not 200), it prints an error message with the status code.

5. Returning Values:
   - It returns two values:
     - The error message (or a placeholder message if not found)
     - The run page URL (or "NA" if not available)

Function: send_email_notification

```
#-------------------------------------------------------------------------------
def send_email_notification(list_of_extracted_job_runs,recipient_emails):

    print("email notification setup...")
    try:

        #email body table header
        html_table = """
<table style="border-collapse: collapse; width: 100%;">
  <thead>
    <tr>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px; background-color: #dddddd;">Run ID</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">Job ID</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">Run State</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">Run Name</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">User Name</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">Run Duration</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">Start Time</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">End Time</th>
    </tr>
  </thead>
  <tbody>
"""
        html_table_error_message = """
        <table style="border-collapse: collapse; width: 100%;">
  <thead>
    <tr>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px; background-color: #dddddd;">Run ID</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">Job ID</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">Error Message</th>
      <th style="border: 1px solid #dddddd; text-align: left; padding: 8px;">Run URL</th>
    </tr>
  </thead>
  <tbody>
        """
```

```
html_table += """  </tbody> </table>"""
#print(html_table)
html_table_error_message += """  </tbody> </table>"""

body = """
<html>
<head>
</head>
<body>
Hi Team,<br>
This email provides a summary of the latest 5 job runs in our Databricks workspace.<br>
{html_table}<br>
<br>
Note:<br>
* Run state is highlighted in red for failed runs and green for successful runs.<br>
<br>* Below Table lists the failure run_id's and error message along with run page URL.<br>
{html_table_error_message}
</body>
</html>
""".format(html_table=html_table,html_table_error_message=html_table_error_message)
print(body)

#sender email, smtp server details
sender_email = "dev-monitoring@databricks.com" # Replace with your sender email
smtp_server = "[redacted]" # Replace with your SMTP server
smtp_port = [redacted]  #replace the port
print("sender_email::"+sender_email+" smtp_server::"+smtp_server+" smtp_port::"+str(smtp_port))

# Create email message with multipart structure
message = MIMEText(body,"html")
message["Subject"] = "Latest Job Run Details"
message["From"] = sender_email
message["To"] = ", ".join(recipient_emails)
```

```
    # Create email message with multipart structure
    message = MIMEText(body,"html")
    message["Subject"] = "Latest Job Run Details"
    message["From"] = sender_email
    message["To"] = ", ".join(recipient_emails)

    # Send the email
    with smtplib.SMTP(smtp_server, smtp_port) as server:
        server.starttls()  # Enable TLS encryption
        server.login("[redacted]", "[redacted]")  # Replace with your email password
        server.sendmail(sender_email, recipient_emails, message.as_string())
        print("email sent successfully...")



except Exception as e:
    print("Error sending the email notification...Please recheck the configs!!\n"+str(e))
    raise Exception("Error sending the email notification...Please recheck the configs!!\n"+str(e))
```

Purpose:

This function constructs and sends an email notification summarizing the details of recent Databricks job runs. It leverages the provided list of extracted job run details and a list of recipient email addresses.

Steps:

1. Setting Up Email Structure:
    - The function initializes HTML code for the email body, including tables to display the job run information.
    - It defines separate HTML table structures for successful runs and failed runs (to highlight failures).
2. Sending the Email:
    - The function defines sender email address, SMTP server details (server address and port), and recipient email addresses.
    - It constructs a MIME message object with the HTML email body and sets the subject, sender email, and recipient list.
    - It establishes a secure SMTP connection using TLS encryption.
    - It sends the email notification through the SMTP server.
    - In case of success, it prints a confirmation message.

Benefits:

This approach offers several advantages:

- Clear and Concise Reports: Users receive informative HTML emails summarizing recent job runs, facilitating easier identification of potential issues.
- Actionable Insights: Detailed information empowers users to take prompt corrective actions.
- Customization Potential: The solution leverages Python's flexibility, allowing for further customization of alerts based on specific needs.

Authors

Ritesh Chidrewar

Senior Consultant - Data Engineering

About LumenData

LumenData is a leading provider of **Enterprise Data Management, Cloud & Analytics** solutions. We help businesses navigate their data visualization and analytics anxieties and enable them to accelerate their innovation journeys.

Founded in 2008, with locations in multiple countries, LumenData is privileged to serve over 100 leading companies. LumenData is **SOC2 certified** and has instituted extensive controls to protect client data, including adherence to GDPR and CCPA regulations.