Case Study Option 1: Python REST API Integration Script (Python-Focused)

Approx. time to complete: 1hr.

Scenario

You are provided with documentation on how to run the **AlphaSense Ingestion API** (see developer documentation at *developer.alpha-sense.com/api/ingestion/*). Every day our clients are using this API to establish a connection with their proprietary internal content to surface insights within our platform. With every implementation, the client follows this documentation and creates a pipeline (we generally don't have access to their environment and can't run their code), we ask you to do the same!

Problem: Create a simple ingestion pipeline to pull sample data from your file system to our platform. You are tasked with generating a script that can take content, properly tag it based on the metadata fields provided, and ingest it into AlphaSense.

Required Deliverables

- **Script:** A revised version of a script (preferred in Python or Javascript) that successfully formats the payload and includes attachments.
- **Explanation Document:** A short summary (e.g., in a README or separate write-up) describing:
  - What you implemented it and how someone would be able to run it.
  - Outline how you would improve the script and what pieces would be most important when going from a sample script to a full pipeline.
  - Any required configuration (e.g., where to set the API key or query parameters).

Expectations for a Good Solution

A strong solution will include:

- **Functional Correctness:** The script correctly handles ingestion, uploading the documents that were intended and includes the correct metadata and attachment file.

- **Code Quality:** The Python code should be clear and maintainable. Use of appropriate libraries (such as `requests` for HTTP calls, and Python's JSON handling) is expected. The logic should be modular (using functions where appropriate) and commented so another engineer can easily follow it. Avoid hard-coding values like API keys or query terms in the script; instead, use variables, configuration files, or environment variables for such inputs.

- **Documentation and Usage Info:** The provided explanation or README should make it easy to understand how to configure and run the script. It should mention any prerequisites (e.g. required Python libraries or Python version) and how to provide input.

Bonus Goals (Optional Stretch Tasks)

If you have extra time or want to demonstrate further skill, consider the following enhancements:

- **Command-Line Interface:** Refactor the script into a simple CLI tool (using a library like `click`) so that users can run it as a command, passing in parameters like query terms, output file path, etc., via command-line arguments.

- **Logging and Monitoring:** Implement logging within the script (using Python's `logging` module) to record progress (e.g., when each page of results is fetched) and any errors encountered. This would make it easier to troubleshoot if issues occur in the future. You could include different log levels (info, warning, error) and an option to turn on verbose logging.

- **Parallel Ingestion:** Modify the script to ingest documents in parallel to improve efficiency. (Be sure to document any such changes and ensure you're still respecting API rate limits.)

Optional/Bonus: Case Study Option 2: Containerized Microservice Troubleshooting (Cross-Functional)

Approx. time to complete: pending experience (30 min - 1+ hr)

Scenario

AlphaSense's platform is powered by dozens of microservices running in a cloud environment (AWS/GCP) using Docker containers orchestrated by Kubernetes. A new version of a microservice has been deployed to a Kubernetes cluster, but clients report that the service is unreachable. You check the pod status and see that the pod is stuck in a CrashLoopBackOff state. The application is expected to load a configuration file from a mounted ConfigMap and start a web server on port 5000. However, it never becomes ready, and requests to the service fail.

Deployment YAML Snippet:

```
Unset
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp-container
        image: myregistry/myapp:2.0.1
        ports:
          - containerPort: 5000
        env:
          - name: CONFIG_PATH
            value: "/etc/myapp/config.yaml"
        volumeMounts:
          - name: config-volume
            mountPath: "/etc/myapp/"   # Mount point for config files
      volumes:
        - name: config-volume
          configMap:
            name: myapp-config        # Reference to ConfigMap (expected to
have config.yaml)
```

Pod Status: Output of `kubectl get pods` after deployment:

```
Unset
NAME                                 READY   STATUS             RESTARTS   AGE
myapp-deployment-5f55c7d6f-abcde     0/1     CrashLoopBackOff   0          3m
```

Pod Description (Events): Except from `kubectl describe pod myapp-deployment-5f55c7d6f-abcde`:

```
Unset
Events:
  Type     Reason       Age    From               Message
  ----     ------       ----   ----               -------
  Normal   Scheduled    3m     default-scheduler  Successfully assigned
default/myapp-deployment-5f55c7d6f-abcde to node1
  Warning  FailedMount  3m     kubelet            MountVolume.SetUp failed for
volume "config-volume" : configmap "myapp-config" not found
  Warning  FailedMount  2m     kubelet            Unable to attach or mount
volumes: unmounted volumes=[config-volume], unattached volumes=[config-volume
etc...]: timed out waiting for the condition
  Warning  FailedMount  1m     kubelet            MountVolume.SetUp failed for
volume "config-volume" : configmap "myapp-config" not found
```

Problem: The Kubernetes deployment references a ConfigMap named `myapp-config`, but this ConfigMap was never created in the cluster. As a result, the pod cannot mount the required volume, fails on startup, and enters a crash loop. This is likely due to a missing dependency in the deployment pipeline or a typo in the ConfigMap reference.

You are tasked with identifying the root cause, correcting the deployment or Kubernetes resources, and verifying that the service becomes healthy and accessible.

Required Deliverables

- **Root Cause Analysis:** Examine the deployment configuration and event logs to determine why the pod is in CrashLoopBackOff. Identify the specific configuration issue or missing dependency that is causing the failure (e.g. a missing ConfigMap, incorrect image, misconfigured probe, etc.). In this case, focus on the evidence in the Events – what does "ConfigMap not found" imply?

- **Proposed Solution:** Explain why the missing ConfigMap prevents the pod from starting, how it leads to the CrashLoopBackOff state, and how your fixes resolve it. This might include creating the missing Kubernetes object (ConfigMap) with the expected content, or correcting a mistaken reference in the deployment YAML. Be precise: if creating a ConfigMap, specify what key data it should contain (even a simple example); if fixing a reference, show the corrected YAML snippet.

- **Verification Plan:** Describe how you would verify that the fix works (e.g., checking pod readiness, viewing logs, confirming service access).

Expectations for Good Deliverables

A high-quality response will show **systematic troubleshooting, knowledge of Kubernetes, and clear reasoning**:

- **Using Clues to Identify Cause:** The candidate recognizes the significance of the events in the pod description. The "configmap \"myapp-config\" not found" message is correctly interpreted as the ConfigMap being missing or named incorrectly. A strong answer will note that this is the root cause of the CrashLoopBackOff – the pod cannot start without the needed configuration, causing repeated failures. The candidate might explicitly mention that a CrashLoopBackOff indicates a pod is repeatedly crashing soon after starting , and that checking the events/logs pinpointed the config volume mount issue.

- **Correct Resolution Steps:** The proposed solution directly addresses the root cause. For example, the candidate would state that we need to **create the ConfigMap** in the cluster (with the name myapp-config) containing the config.yaml data required by the application. They might provide a sample kubectl create configmap myapp-config --from-file=config.yaml command or a YAML definition of the ConfigMap. If the issue were a naming typo, the solution could be to update the deployment to refer to the correct ConfigMap name. In either case, the fix should be precise and actionable.

- **Understanding of Kubernetes Configuration:** The explanation should demonstrate *why* the issue caused the pod failure. For instance, a good explanation might say: *"Because the ConfigMap was not present, the volume config-volume could not mount, and the container's startup script likely exited due to missing /etc/myapp/config.yaml. This caused Kubernetes to repeatedly restart the pod (CrashLoopBackOff). By creating the ConfigMap with the required configuration, the volume mount will succeed and the application can start normally."* The candidate might also mention that Kubernetes prevents the container from starting until the volume is mounted, hence the events showing the mount failure.

- **Thorough Verification:** A strong answer will include how the candidate verifies the fix. They might mention applying the ConfigMap and re-deploying (or updating the deployment) and then running kubectl get pods to see the pod become Running. Additionally, they could suggest checking kubectl logs for the container to ensure it reads the config without errors, and maybe performing a simple request to the service or checking the pod's readiness status. This shows an end-to-end approach to the problem.

- **Clarity and Structure:** The response should be well-organized and easy to follow. Each step (identify, fix, verify) should be clearly articulated, either in a short paragraph or as bullet points. The language should be professional and objective, focusing on facts of the scenario and solution. Any Kubernetes terminology (ConfigMap, volume mount, CrashLoopBackOff, etc.) should be used correctly.

Bonus Goals

If the candidate successfully addresses the primary issue quickly, you can explore additional **stretch topics** to gauge deeper knowledge:

- **Alternate Failure Scenarios:** Ask the candidate to briefly discuss how they would troubleshoot other common deployment failures. For example:

  - *Image pull errors:* If the pod Events showed an image pull error (e.g., ErrImagePull or ImagePullBackOff), what steps would they take? (Expected answer: check image name/tag correctness, repository accessibility, credentials, etc.)

  - *Readiness/Liveness probe failures:* If the pod was running but not routing traffic due to failing health checks, how would they identify and resolve that? (Expected: describe using kubectl describe to see probe failure events like “Readiness probe failed: connection refused”, then adjust the probe config or fix the app as needed).

  - *Resource limits:* If the container was OOMKilled (out of memory), how to detect that in events (kubectl describe showing “OOMKilled” reason) and what to do (increase memory limit or optimize the app).

    These discussions can show the candidate’s general expertise in Kubernetes troubleshooting beyond this specific case.

- **Preventative Measures:** Encourage the candidate to think about how such issues could be prevented in a real-world DevOps process. For example, they might suggest implementing configuration management practices: using Helm or Kubernetes admissions to ensure a ConfigMap exists before deployment, or CI/CD pipeline checks that all referenced resources (configmaps, secrets) are created. They could also mention monitoring and alerting (e.g., an alert on CrashLoopBackOff pods) to catch issues quickly. This demonstrates an SRE mindset of proactive prevention and not just reactive fixes.

- **Documentation and Communication:** As an extra challenge, the candidate could outline how they would document this incident for future reference. This might include updating runbooks or playbooks with the symptom (CrashLoopBackOff with ConfigMap not found) and resolution steps, or communicating to the team about ensuring configs are deployed. While not a purely technical task, it underscores the importance of **communication**, which is crucial in support roles.