**Integration with Existing Systems (Updated Report)**

**Table of Contents**

1. [Introduction](#introduction)
   * [Overview of the Project](#overview-of-the-project)
   * [Goals](#goals)
2. [Project Setup and Initial Considerations](#project-setup-and-initial-consideration)
   * [Environment Configuration](#environment-configuration)
   * [Initial Challenges and Strategies](#initial-challenges-and-strategies)
3. [Identifying Integration Points](#identifying-integration-points)
   * [Existing Systems Overview](#existing-systems-overview)
   * [Integration Requirements](#integration-requirements)
4. [Developing Integration Scripts](#developing-integration-scripts)
   * [Design and Rationale](#design-and-rationale)
   * [Analysis of **OpenMaintClient.py**](#analysis-of-openmaintclientpy)
   * [Analysis of **openmaint\_consumer.py**](#analysis-of-openmaint_consumerpy)
5. [Testing Integrations with Existing Data Sources](#testing-integrations-with-existing-data)
   * [Test Cases and Results](#test-cases-and-results)
   * [Error Handling and Logging](#error-handling-and-logging)
6. [Challenges and Solutions](#challenges-and-solutions)
   * [Key Decision Points](#key-decision-points)
   * [Problems Encountered and Resolutions](#problems-encountered-and-resolutions)
7. [Conclusion and Future Work](#conclusion-and-future-work)
   * [Summary of the Process](#summary-of-the-process)
   * [Areas for Improvement](#areas-for-improvement)
   * [Future Enhancements](#future-enhancements)
8. [Appendices](#appendices)
   * [Complete Scripts](#complete-scripts)
   * [References](#references)

**1. Introduction**

**Overview of the Project**

The goal of the **Integration with Existing Systems** phase is to connect our predictive maintenance pipeline to openMAINT (a CMMS) to automatically create and manage work orders when anomalies are detected. Additionally, we incorporate Kafka as the messaging backbone for real-time data and alerts.

**What’s New in the Updated Integration Scripts?**

1. **Docker-based openMAINT Startup**  
   Instead of manually configuring and launching openMAINT, **OpenMaintClient.py** now leverages a Docker Compose file to start openMAINT services automatically.
2. **Context Manager for Clean Resource Handling**  
   A managed\_openmaint\_client context manager ensures automatic logout and resource cleanup.
3. **Enhanced Setup and Wait Logic**  
   The new client setup includes a robust “wait-for” mechanism to ensure openMAINT API readiness before performing login actions.
4. **Threaded Shutdown and Graceful Consumer Exit**  
   The openmaint\_consumer\_main function uses a threading.Event for stopping gracefully.

**Goals**

**Subtask 5.1**: Identify integration points with existing systems (primarily openMAINT) and databases.

* **Subtask 5.2**: Implement scripts for seamless data flow (automatic creation of work orders, etc.).
* **Subtask 5.3**: Test the integration with real or simulated data sources (Kafka → openMAINT).
* **Subtask 5.4**: Document the updated integration processes and best practices for future maintainers.

**2. Project Setup and Initial Considerations**

**Environment Configuration**

**Programming Language**: Python 3.12

* **Libraries and Frameworks**:
  + Data Handling: pandas, numpy
  + API Interaction: requests
  + Kafka: kafka-python
  + Logging & Utilities: logging, sys, json, subprocess, time
  + Configuration: yaml
* **Systems Integrated**:
  + **openMAINT**: A Dockerized CMMS instance, started via Docker Compose.
  + **Apache Kafka**: Receives failure\_predictions messages used to trigger work orders.
  + **PostgreSQL**: openMAINT uses PostgreSQL behind the scenes for its database.

**Initial Challenges and Strategies**

**Docker-based openMAINT**: We needed a reliable way to start and wait for openMAINT to be responsive via its REST API.

* **Session & Authorization**: Managing login sessions in openMAINT, including storing and reusing session tokens.
* **Data Consistency**: Ensuring the format of each Kafka message aligns with the requirements for a valid openMAINT work order.
* **Error Handling & Graceful Teardown**: Devising a safe shutdown path for the Kafka consumer and openMAINT client.

**3. Identifying Integration Points**

**Existing Systems Overview**

**openMAINT**: Manages asset data, maintenance workflows, and work orders. We integrate via openMAINT’s REST endpoints to create new work orders automatically.

* **Kafka**: Streams sensor predictions from the RealTimeProcessor. Our consumer reads the topic failure\_predictions, then triggers openMAINT work order creation upon receiving anomaly alerts.

**Integration Requirements**

**Start & Wait**: Seamless startup of openMAINT container environment.

1. **Create Work Orders**: Ingest messages from the failure\_predictions topic and convert them into openMAINT work orders with correct IDs (priority, type, site, etc.).
2. **Authentication**: Use session tokens for repeated interactions with openMAINT within a single run.
3. **Logging and Debuggability**: Provide logs for all major steps and errors to facilitate debugging.

**4. Developing Integration Scripts**

**Design and Rationale**

Previously, scripts required manual openMAINT startup steps. The updated approach:

* **OpenMaintClient**:
  1. **Docker Compose** for openMAINT cluster management.
  2. **Wait for API** logic to ensure readiness.
  3. **Session-based** interaction for repeated calls (create work orders, get lookups, etc.).
* **openmaint\_consumer**:
  1. Kafka consumer reading failure\_predictions.
  2. Work order creation via OpenMaintClient functions.
  3. Graceful shutdown via a threading.Event and signal handlers.

This architecture centralizes openMAINT-specific logic into a single client module (OpenMaintClient.py) while the consumer script (openmaint\_consumer.py) focuses on Kafka consumption and message handling.

**Analysis of OpenMaintClient.py**

**Imports & Logger Setup**

import os

import subprocess

import sys

import time

import requests

import json

import logging

from datetime import datetime

import urllib.parse

* + Logging is configured to capture debug-level logs to both file and console.

1. **Class Initialization**

class OpenMaintClient:

def \_\_init\_\_(self, api\_url, username, password, log\_file\_path=...):

self.logger = self.setup\_logger(log\_file\_path)

...

subprocess.run(['docker-compose', '-f', docker\_compose\_path, 'up', '-d'], check=True)

...

self.api\_url = api\_url

self.username = username

self.password = password

time.sleep(15)

self.wait\_for\_api(api\_url)

self.session = None

self.login()

* + **Docker Startup**: The constructor automatically starts openMAINT containers using the project’s Docker Compose file.
  + **wait\_for\_api**: The client waits up to max\_retries \* wait\_interval seconds, polling the openMAINT login endpoint to confirm readiness.
  + **login()**: Manages session ID retrieval upon successful authentication.

1. **Session Logic**

def login(self):

...

session\_id = response.json().get("data", {}).get("\_id")

self.session.headers.update({

"CMDBuild-Authorization": session\_id, ...

})

...

* + After retrieving the session ID, subsequent requests carry the CMDBuild-Authorization header.

1. **Lookup & Helper Methods**

def get\_lookup\_id(self, lookup\_type, code): ...

def get\_employee\_id(self, employee\_name, class\_name='Employee'): ...

def get\_site\_id(self, site\_code): ...

def get\_ci\_id(self, asset\_code, class\_name="Equipment"): ...

* + Each helps retrieve the \_id necessary to populate fields in openMAINT (priorities, sites, employees, etc.).
  + The updated script includes robust error handling and logging warnings if IDs are not found.

1. **Work Order Creation**

def create\_work\_order(self, work\_order\_data):

...

if response.status\_code == 200 and response.json().get("success"):

...

else:

...

* + This method hits the CorrectiveMaint process endpoint to create a new maintenance process instance.

1. **Wait & Teardown**
   * The logout() method closes the session, ensuring a clean end-of-run scenario.
   * If openMAINT fails to start, the \_\_init\_\_ raises an exception.

**Key Differences from Older Scripts**:

* **Docker Orchestration** for openMAINT (instead of manual instructions or shell commands).
* **wait\_for\_api** function ensures we do not attempt to login before openMAINT is actually ready.
* **Context Manager** usage (see managed\_openmaint\_client in openmaint\_consumer.py) ensures proper logout.

**Analysis of openmaint\_consumer.py**

**Imports & Configuration Loading**

from kafka import KafkaConsumer

from IntegrationWithExistingSystems.OpenMaintClient import OpenMaintClient

import yaml

import json

import threading

import signal

* + Script reads kafka\_config.yaml & openmaint\_config.yaml for server details.

1. **Logging**
   * Configured at the top with file and console handlers, capturing debug messages in openmaint\_consumer.log.
2. **Kafka Consumer Creation**

def create\_kafka\_consumer(kafka\_config):

consumer = KafkaConsumer(

kafka\_config.get('failure\_predictions\_topic', 'failure\_predictions'),

...

consumer\_timeout\_ms=5000

)

* + Reads from failure\_predictions by default, though the topic is configurable via YAML.

1. **Context Manager**

@contextmanager

def managed\_openmaint\_client(api\_url, username, password):

client = OpenMaintClient(api\_url, username, password)

try:

yield client

finally:

client.logout()

* + Simplifies openMAINT client usage and ensures a logout happens on function exit or an exception.

1. **Main Consumer Logic**

def openmaint\_consumer\_main(shutdown\_event):

kafka\_config = load\_config(config\_path).get('kafka', {})

openmaint\_config = load\_config(openmaint\_config\_path).get('openmaint', {})

consumer = create\_kafka\_consumer(kafka\_config)

with managed\_openmaint\_client(API\_BASE\_URL, USERNAME, PASSWORD) as client:

while not shutdown\_event.is\_set():

message\_pack = consumer.poll(timeout\_ms=1000)

for tp, messages in message\_pack.items():

for message in messages:

process\_message(client, message)

* + Polls Kafka regularly, passing each message to process\_message(...) which orchestrates creation of a work order in openMAINT.

1. **process\_message(...)**

def process\_message(client, message):

# Retrieve necessary IDs, e.g. priority, type, site, etc.

# Construct the WO data payload

# call client.create\_work\_order(work\_order\_data)

# fetch & log details

* + Encodes the logic to transform a prediction message (e.g., risk score, date) into openMAINT fields.

1. **Graceful Shutdown**

def handle\_shutdown(signum, frame, shutdown\_event):

shutdown\_event.set()

* + Catches SIGINT/SIGTERM, sets an event that stops the consumer loop.

**Key Differences from Older Scripts**:

* **Threading & Signal Handling**: A threading.Event is used to break out of the consumption loop gracefully.
* **Context Manager**: Eliminates the risk of leftover sessions or partial initializations.
* **Improved Logging**: The script logs each step at debug or info levels, including missing ID checks or partial failures.

**5. Testing Integrations with Existing Data Sources**

**Test Cases and Results**

**Start Up and Wait**

* + **Input**: Invoked OpenMaintClient(...).
  + **Process**: Docker Compose up, wait for openMAINT, authenticate.
  + **Result**: openMAINT started, API login success after ~15 seconds.

1. **Work Order Creation**
   * **Input**: Simulated Kafka message with an anomaly.
   * **Process**: process\_message(...) queries lookups (Priority, Type, etc.) and calls create\_work\_order(...).
   * **Result**: Work order created, ID returned, details fetched—**Success**.
2. **Site/Employee Lookup**
   * **Input**: Existing site code, employee name.
   * **Process**: client.get\_site\_id(...), client.get\_employee\_id(...).
   * **Result**: Correct \_id retrieved, error logs if code not found.
3. **API Resilience**
   * **Input**: Induced openMAINT unavailability mid-run.
   * **Process**: Received exceptions, script logged network errors, resumed for next messages.
   * **Result**: Proper log messages indicated the failure, no indefinite blocking.

**Error Handling and Logging**

**API or Network Failures**: Caught within process\_message and logged.

* **Missing Data**: If a lookup code fails, a warning is issued, and the message is skipped.
* **Time-out / Docker Start**: If openMAINT is not ready after a configurable number of retries, it raises a ConnectionError.

**6. Challenges and Solutions**

**Key Decision Points**

1. **Docker Startup**
   * *Challenge*: Minimizing manual steps.
   * *Solution*: OpenMaintClient constructor runs Docker Compose automatically, then polls API readiness.
2. **Session Management**
   * *Challenge*: Inconsistent or expired sessions.
   * *Solution*: Central login method in OpenMaintClient, storing session token in the request header.
3. **Data Consistency**
   * *Challenge*: Matching IDs to openMAINT’s internal references.
   * *Solution*: A suite of “get\_\*\_id” helper methods that unify lookup logic.

**Problems Encountered and Resolutions**

**Long Startup Times**

* + *Problem*: openMAINT can sometimes take longer than 15 seconds to be “ready.”
  + *Resolution*: The wait\_for\_api loop uses 30 retries of 5 seconds each by default, logging attempts.

1. **Missing Lookup Values**
   * *Problem*: The script used priority code 3, but openMAINT did not contain that code.
   * *Resolution*: The script logs a warning if no match is found and bypasses creation, preventing partial data entry.
2. **Interrupt Handling**
   * *Problem*: Force stops caused abrupt script termination.
   * *Resolution*: A signal handler sets the shutdown\_event, letting the consumer exit gracefully and run final logout().

**7. Conclusion and Future**

**Summary of the Process**

* **Docker-based** openMAINT integration: ensures consistent environment with no manual steps.
* **Kafka-based** consumption: listens for anomaly messages from the failure\_predictions topic.
* **Enhanced Logging & Error Handling**: provides transparent operational insights.
* **Clean Resource Handling**: via context managers and graceful thread signals.

**Areas for Improvement**

* **Dynamic Config**: Move more parameters (e.g., priority code, site code) to YAML for easier modifications.
* **Session Reuse**: Possibly re-authenticate if a long-running session times out.
* **Bulk Operations**: For high-volume systems, consider batch processing or asynchronous job scheduling.

**Future Enhancements**

* **Additional Workflows**: Not just “CorrectiveMaint”; for example, integrate with “PreventiveMaint” or other modules.
* **Monitoring/Alerting**: Add real-time dashboards for openMAINT consumer logs.
* **Extended Integrations**: Link the pipeline with ERP or inventory systems to automatically schedule parts ordering.

**8. Appendices**

**Complete Scripts**

* **OpenMaintClient.py**  
  [See updated script provided in the codebase for details of Docker-based openMAINT startup, logging, session authentication, and work order creation methods.]
* **openmaint\_consumer.py**  
  [See updated script for Kafka consumer logic, context manager usage, and graceful shutdown flow.]

**References**

* **openMAINT Documentation**: [openMAINT REST API Docs]
* **Kafka Documentation**: [Apache Kafka Documentation](https://kafka.apache.org/documentation/)
* **Python Requests**: Requests: HTTP for Humans
* **Python Logging Module**: [Python 3.12 Logging Documentation](https://docs.python.org/3.12/howto/logging.html)