**1. Introduction**

The **run.py** script serves as the central orchestration point for the entire predictive maintenance system. It automates:

1. **Setup**
   * Configures the logging system.
   * Establishes paths and loads necessary YAML configuration files.
   * Extracts zip files (if needed) for Kafka and openMAINT.
2. **Service Startup**
   * Spins up essential Docker services (Kafka cluster, openMAINT) via Docker Compose.
   * Waits for services to stabilize.
3. **Script Execution**
   * Launches the real-time data simulation, the main data processing routines, and the integration consumer script for openMAINT.
4. **User-Friendly Access**
   * Automatically opens configured URLs (openMAINT and Grafana) in the user’s default browser after services are running.
5. **Graceful Shutdown**
   * Monitors the processes and handles interruptions (e.g., KeyboardInterrupt) to stop everything cleanly.

In essence, **run.py** is the “main runner” that integrates all components of the project—sensor data simulation, real-time processing, integration with openMAINT, and monitoring with Grafana.

**2. Overall Flow and Logical Chronology**

1. **Logging Setup**
   * Immediately configures and returns a logger to capture script events.
2. **Path and Configuration Handling**
   * Detects the base directory (where run.py lives) to maintain consistent relative paths.
   * Identifies YAML files for openMAINT and Grafana configurations.
3. **Zip Extraction**
   * Checks if the required folders already exist. If not, attempts to extract any zip files for Kafka or openMAINT.
   * This avoids manual extraction steps for the end-user.
4. **Loading YAML Configurations**
   * Fetches openMAINT and Grafana configs from YAML files.
5. **Docker Compose Services**
   * Sequentially starts Docker Compose services for Kafka and openMAINT.
   * Waits a specified period to ensure containers have time to initialize.
6. **Launching Scripts**
   * Spawns three subprocesses:
     1. **SensorDataSimulator**
     2. **RealTimeProcessor**
     3. **openmaint\_consumer**
   * These processes run in parallel to form the real-time pipeline.
7. **Browser Opening**
   * After a short delay, automatically opens the relevant openMAINT and Grafana URLs.
8. **Monitoring**
   * The main thread waits for the three processes to complete or get interrupted.
   * Handles graceful termination if a KeyboardInterrupt or error occurs.
9. **Final Cleanup**
   * Logs a message that all processes are done or terminated, concluding the script’s workflow.

**3. Detailed Line-by-Line Analysis**

Below is the **run.py** script annotated with explanations line by line. (Note: For brevity, each code excerpt is grouped by function or relevant block, then explained. The numbering references your original script lines as best as possible.)

python

Copy code

1 | import os

2 | import sys

3 | import time

4 | import logging

5 | import subprocess

6 | import webbrowser

7 | import yaml

8 | import zipfile

9 | import warnings

10 | from sklearn.exceptions import InconsistentVersionWarning

11 | warnings.filterwarnings("ignore", category=InconsistentVersionWarning)

12 | TF\_ENABLE\_ONEDNN\_OPTS = 0

* **Lines 1–8**: Import standard libraries needed for file system operations (os, sys), timing (time), logging, process launching (subprocess), opening web browsers, YAML parsing, and handling zip archives.
* **Lines 9–11**: Import warnings to suppress specific scikit-learn version warnings. This ensures logs aren’t cluttered with non-critical messages about version mismatches.
* **Line 12**: Disables certain TensorFlow onednn optimizations to potentially avoid warnings or numeric differences.

python

Copy code

14 | def setup\_logging(log\_level=logging.DEBUG):

15 | logger = logging.getLogger(\_\_name\_\_)

16 | logger.setLevel(log\_level)

17 |

18 | ch = logging.StreamHandler(sys.stdout)

19 | ch.setLevel(log\_level)

20 | formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

21 | ch.setFormatter(formatter)

22 |

23 | if not logger.handlers:

24 | logger.addHandler(ch)

25 |

26 | return logger

* **Lines 14–26: setup\_logging function**
  + Defines a function to initialize and return a logger instance.
  + Creates a StreamHandler to output logs to standard output and applies a formatter for consistent log messages.
  + Checks if the logger already has handlers to avoid adding duplicates.
  + Returns the configured logger, enabling debugging throughout the script.

python

Copy code

28 | def load\_yaml\_config(file\_path, logger):

29 | if not os.path.exists(file\_path):

30 | logger.error(f"Configuration file does not exist: {file\_path}")

31 | raise FileNotFoundError(f"Configuration file not found: {file\_path}")

32 |

33 | with open(file\_path, 'r') as f:

34 | try:

35 | config = yaml.safe\_load(f)

36 | return config

37 | except yaml.YAMLError as e:

38 | logger.error(f"Error loading YAML config from {file\_path}: {e}")

39 | raise

* **Lines 28–39: load\_yaml\_config function**
  + Accepts a file path and logger. Checks if the file exists.
  + Attempts to parse the YAML file.
  + If successful, returns the configuration dictionary. Otherwise, logs an error and raises an exception.

python

Copy code

41 | def extract\_zip\_file(zip\_path, extract\_to, logger):

42 | """

43 | Extracts the contents of the given zip file to the specified directory.

44 | If the directory does not exist, it will be created.

45 | """

46 | if not os.path.exists(zip\_path):

47 | logger.info(f"No ZIP file found at {zip\_path}. Skipping extraction.")

48 | return False

49 |

50 | try:

51 | logger.info(f"Extracting {zip\_path} to {extract\_to}...")

52 | if not os.path.exists(extract\_to):

53 | os.makedirs(extract\_to, exist\_ok=True)

54 | with zipfile.ZipFile(zip\_path, 'r') as zip\_ref:

55 | zip\_ref.extractall(extract\_to)

56 | logger.info(f"Extraction of {zip\_path} completed successfully.")

57 | return True

58 | except zipfile.BadZipFile as e:

59 | logger.error(f"Invalid ZIP file {zip\_path}: {e}")

60 | raise

61 | except Exception as e:

62 | logger.error(f"Failed to extract {zip\_path}: {e}")

63 | raise

* **Lines 41–63: extract\_zip\_file function**
  + Handles the logic for extracting a .zip archive if it exists.
  + If the .zip does not exist, logs a message and returns False.
  + Otherwise, it creates the destination folder (if necessary) and extracts the archive.
  + Errors during extraction are logged, and exceptions are re-raised to stop the script if critical.

python

Copy code

65 | def start\_services(docker\_compose\_path, service\_desc, logger, wait\_time=20):

66 | if not os.path.exists(docker\_compose\_path):

67 | logger.error(f"Docker Compose file not found at: {docker\_compose\_path}")

68 | raise FileNotFoundError(f"Docker Compose file not found at: {docker\_compose\_path}")

69 |

70 | logger.info(f"Starting {service\_desc} using Docker Compose...")

71 | try:

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

73 | logger.info(f"{service\_desc} started successfully. Waiting {wait\_time} seconds for services to be ready.")

74 | time.sleep(wait\_time)

75 | except subprocess.CalledProcessError as e:

76 | logger.error(f"Failed to start {service\_desc}: {e}")

77 | raise

78 | except KeyboardInterrupt as e:

79 | logger.error("An KeyboardInterrupt interrupt while running processes.", exc\_info=True)

80 | sys.exit(0)

81 |

82 | except Exception as e:

83 | logger.error(f"An Exception interrupted while running processes.{e}", exc\_info=True)

84 | raise

* **Lines 65–84: start\_services function**
  + Launches a Docker Compose service (like Kafka or openMAINT) by running docker-compose up -d.
  + Logs the service description, so the user knows which environment is starting.
  + Waits a specified wait\_time (default 20s) to give containers time to stabilize.
  + If any errors occur, logs them and raises or exits.

python

Copy code

86 | def main():

87 |

88 | # Use the directory where run.py is located as the base directory

89 | base\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_))

90 | logger.debug(f"Base directory: {base\_dir}")

91 |

92 | # Paths to docker-compose files

93 | kafka\_compose\_path = os.path.join(base\_dir, 'docker-compose.yml')

94 | openmaint\_compose\_path = os.path.join(base\_dir, 'IntegrationWithExistingSystems', 'openmaint-2.3-3.4.1-d',

95 | 'docker-compose.yml')

* **Line 86: def main():**
  + Defines the principal workflow inside main().
* **Lines 89–90**
  + Detects the absolute path of the current script’s directory (the “base\_dir”) and logs it for debugging.
* **Lines 92–95**
  + Assembles the paths to the Docker Compose files for Kafka (docker-compose.yml) and openMAINT.
  + The openMAINT docker-compose.yml is nested in a subdirectory (IntegrationWithExistingSystems/openmaint-...).

python

Copy code

97 | openmaint\_config\_path = os.path.join(base\_dir, 'config', 'openmaint\_config.yaml')

98 | grafana\_config\_path = os.path.join(base\_dir, 'config', 'grafana\_config.yaml')

* **Lines 97–98**
  + Construct full file paths for the openMAINT and Grafana YAML configuration files, kept under ./config/.

python

Copy code

101 | # Before loading configs, extract any required ZIP files.

102 | openmaint\_zip\_path = os.path.join(base\_dir, 'IntegrationWithExistingSystems', 'openmaint-2.3-3.4.1-d.zip')

103 | openmaint\_extract\_path = os.path.join(base\_dir, 'IntegrationWithExistingSystems')

104 |

105 | kafka\_zip\_path = os.path.join(base\_dir, 'RealTimeProcessing', 'kafka.zip')

106 | kafka\_extract\_path = os.path.join(base\_dir, 'RealTimeProcessing')

* **Lines 101–106**
  + Points to potential .zip archives for openMAINT and Kafka.
  + Also defines extraction directories—these mirror the expected folder structures for each component.

python

Copy code

108 | if not os.path.exists(openmaint\_compose\_path):

109 | # Extract openmaint zip if exists

110 | try:

111 | extract\_zip\_file(openmaint\_zip\_path, openmaint\_extract\_path, logger)

112 | except Exception as e:

113 | logger.error("Failed to extract openmaint ZIP file.", exc\_info=True)

114 | sys.exit(1)

* **Lines 108–114**
  + Checks if the openMAINT docker-compose.yml is missing. If so, it tries to extract the .zip archive.
  + If extraction fails, logs the error and exits. This ensures the script cannot continue if openMAINT is unavailable.

python

Copy code

116 | if not os.path.exists(os.path.join(base\_dir, 'RealTimeProcessing', 'kafka')):

117 | try:

118 | extract\_zip\_file(kafka\_zip\_path, kafka\_extract\_path, logger)

119 | except Exception as e:

120 | logger.error("Failed to extract kafka ZIP file.", exc\_info=True)

121 | sys.exit(1)

* **Lines 116–121**
  + Similar logic for Kafka: if the kafka folder doesn’t exist in RealTimeProcessing, attempts to extract from kafka.zip.
  + If any error surfaces, the script exits.

python

Copy code

124 | # Now load configurations

125 | try:

126 | openmaint\_config = load\_yaml\_config(openmaint\_config\_path, logger)

127 | grafana\_config = load\_yaml\_config(grafana\_config\_path, logger)

128 | except Exception as e:

129 | logger.error("Failed to load configuration files.", exc\_info=True)

130 | sys.exit(1)

* **Lines 124–130**
  + Attempts to load the openMAINT and Grafana configurations from YAML.
  + If anything fails (missing files, YAML parse error), logs it and terminates.

python

Copy code

133 | # Extract URLs

134 | try:

135 | openmaint\_url = openmaint\_config['openmaint']['work\_order']

136 | grafana\_url = grafana\_config['url']

137 | logger.info(f"OpenMaint work\_order URL: {openmaint\_url}")

138 | logger.info("OpenMaint's: Username: admin Password: admin")

139 | logger.info(f"Grafana URL: {grafana\_url}")

140 | except KeyError as e:

141 | logger.error(f"Missing expected keys in configuration: {e}", exc\_info=True)

142 | sys.exit(1)

* **Lines 133–142**
  + Reads the openmaint\_url and grafana\_url from their respective config dicts.
  + Logs these URLs for the user, along with default credentials for openMAINT.
  + If keys are missing, a KeyError is caught, and the script halts.

python

Copy code

145 | # Start Kafka cluster

146 | try:

147 | start\_services(kafka\_compose\_path, "Kafka cluster", logger, wait\_time=1)#15

148 | except Exception as e:

149 | logger.error("Could not start Kafka services.", exc\_info=True)

150 | sys.exit(1)

* **Lines 145–150**
  + Invokes start\_services for the Kafka cluster with a wait time of 1 second (commented as possibly 15 in the code).
  + If an exception arises, logs an error and exits.

python

Copy code

152 | # Start openMAINT cluster

153 | try:

154 | start\_services(openmaint\_compose\_path, "openMAINT cluster", logger, wait\_time=1)#45

155 | except Exception as e:

156 | logger.error("Could not start openMAINT services.", exc\_info=True)

157 | sys.exit(1)

* **Lines 152–157**
  + Similarly spins up openMAINT.
  + The wait time is set to 1 second here (with a comment indicating it might be 45 in a real scenario).

python

Copy code

160 | # Paths to scripts

161 | sensor\_script = os.path.join(base\_dir, 'RealTimeProcessing', 'SensorDataSimulator.py')

162 | processor\_script = os.path.join(base\_dir, 'RealTimeProcessing', 'RealTimeProcessor.py')

163 | openmaint\_script = os.path.join(base\_dir, 'IntegrationWithExistingSystems', 'openmaint\_consumer.py')

* **Lines 160–163**
  + Defines the three key Python scripts that together form the real-time data pipeline:
    1. **SensorDataSimulator.py** – Simulates sensor data and sends it to Kafka.
    2. **RealTimeProcessor.py** – Consumes sensor data, runs ML models, and produces failure predictions back to Kafka.
    3. **openmaint\_consumer.py** – Listens for failure predictions, then interfaces with openMAINT to generate work orders.

python

Copy code

165 | for script\_path in [sensor\_script, processor\_script, openmaint\_script]:

166 | if not os.path.exists(script\_path):

167 | logger.error(f"Script not found: {script\_path}")

168 | sys.exit(1)

* **Lines 165–168**
  + Verifies all three scripts exist before proceeding. Logs an error and exits if any are missing.

python

Copy code

170 | processes = []

171 | try:

172 | logger.info("Starting SensorDataSimulator...")

173 | processes.append(subprocess.Popen([sys.executable, sensor\_script]))

174 | logger.info("Starting RealTimeProcessor...")

175 | processes.append(subprocess.Popen([sys.executable, processor\_script]))

176 |

177 | logger.info("Starting openmaint\_consumer...")

178 | processes.append(subprocess.Popen([sys.executable, openmaint\_script]))

179 |

180 | # Wait a bit before opening browsers

181 | time.sleep(10)

182 |

183 | # Open the URLs in the default browser

184 | logger.info("Opening configured URLs in the default browser...")

185 | try:

186 | webbrowser.open(openmaint\_url)

187 | webbrowser.open(grafana\_url)

188 | except Exception as e:

189 | logger.error(f"Failed to open URLs in the browser: {e}", exc\_info=True)

190 |

191 | for p in processes:

192 | p.wait()

* **Lines 170–192**
  + **Line 170** initializes an empty list to keep track of the processes.
  + **Lines 172–178** start each script using subprocess.Popen(), appending process objects to processes.
  + **Lines 181–182** waits 10 seconds to let scripts warm up.
  + **Lines 184–189** attempts to open openmaint\_url and grafana\_url in the default browser, logging any failures.
  + **Lines 191–192**: The script loops over all spawned processes and waits for them to finish. This effectively blocks here until user ends the process or the scripts exit.

python

Copy code

194 | except KeyboardInterrupt:

195 | logger.info("KeyboardInterrupt received. Terminating all processes.")

196 | for p in processes:

197 | if p.poll() is None:

198 | p.terminate()

199 | except Exception as e:

200 | logger.error("An unexpected error occurred while running processes.", exc\_info=True)

201 | for p in processes:

202 | if p.poll() is None:

203 | p.terminate()

204 | sys.exit(1)

205 | finally:

206 | logger.info("All processes have completed or been terminated.")

207 |

208 | logger.info("Script completed successfully.")

* **Lines 194–208**
  + **except KeyboardInterrupt block**: If the user presses Ctrl+C, logs a message and terminates the child processes.
  + **Generic except Exception**: Any other error also triggers process termination.
  + **finally**: Logs that all processes concluded or got terminated.
  + **Line 208**: Final log statement indicating success.

python

Copy code

210 | if \_\_name\_\_ == "\_\_main\_\_":

211 | logger = setup\_logging()

212 | try:

213 | main()

214 | except Exception as e:

215 | logger.error(f"An Exception interrupted while running processes.{e}", exc\_info=True)

* **Lines 210–215**
  + Standard Python “entry point” block.
  + **Line 211** obtains a logger from setup\_logging().
  + **Lines 212–213** calls the main() function.
  + **Lines 214–215** logs any unhandled exceptions from main().

**4. How run.py Ties the Entire Project Together**

* **Configuration-Driven**  
  By loading YAML files for openMAINT and Grafana, the script ensures the system can adapt to environment or credential changes without code modifications.
* **Automated Docker Compose**  
  This approach eliminates the need to manually start Kafka or openMAINT, streamlining the setup for both developers and end-users.
* **Chain of Scripts**  
  After ensuring both Kafka and openMAINT are running, **run.py** executes three parallel scripts that simulate data, process it, and relay critical maintenance events to openMAINT. Without this centralized runner, users would have to launch each step individually.
* **Convenience**  
  The script then automatically opens the relevant dashboards (Grafana) and openMAINT URLs in the browser, guiding the user directly to the system’s interfaces.
* **Graceful Shutdown**  
  Using exception handling and process monitoring, **run.py** ensures an orderly shutdown of all child processes if the user stops the script or an error occurs.

**5. Key Benefits and Final Observations**

1. **Modularity**: Each crucial task (logging, YAML loading, Docker Compose orchestration, etc.) is encapsulated in its own function, keeping main() readable.
2. **Robustness**: Handles common failure points—missing files, extraction errors, or service startup problems.
3. **Scalability**: Extending this system (e.g., adding a new script for advanced analytics) only requires adding another subprocess call, as long as Docker Compose and config steps are updated accordingly.
4. **Maintainability**: The code is straightforward to modify if paths or wait times need adjustments.

**Conclusion**: The **run.py** script is the operational “heart” of the entire predictive maintenance project, orchestrating Docker-based dependencies, launching real-time pipeline components, and offering user-friendly links to dashboards. Its design ensures a single command can bring the entire system online, making it an essential central hub for production or local testing environments.