SOFTWARE TEST DESCRIPTION

Sahar: Online Mental Health Support

Team Members:

Hila Elbaz

Omer Dor

Nir Yaron

Ron Franco

סהר
סיוע והקשבה ברשת

Model testing:

GSR Production Model Testing:

The GSR (General Sucidal Risk) model is a binary classifier that processes user-generated text (single messages, or entire conversations) and outputs a suicidality risk score. During training, we employed a label synthesis approach to construct synthetic labels for the messages (as they were unlabeled, only conversation labeling were available). additionally we employed an ensemble approach which handles long conversations. The testing process have followed these steps:

Statistical Evaluation (on Test Set, 80:20 Split)

the following metrics were computed:

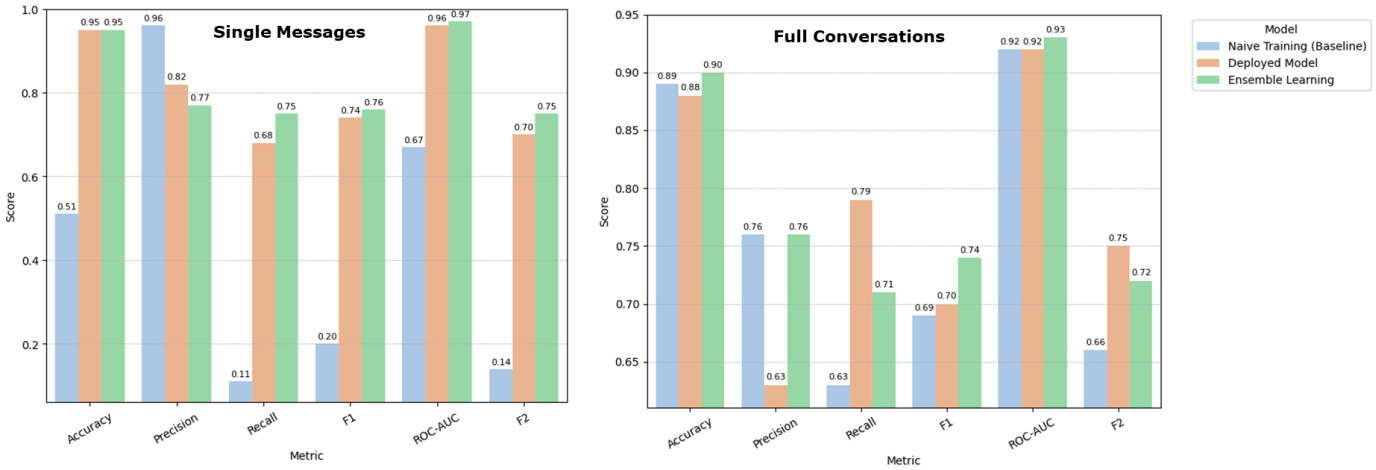
- **Accuracy**: Proportion of total predictions that are correct.
  - Expected Result:
    - on conversation prediction task: **above 0.80.**
    - on single-message prediction task: **above 0.80.**
  - Actual results with deployed Model:
    - on conversation prediction task: **0.898.**
    - on single-message prediction task: **0.933.**
  - Actual results with Ensemble Model:
    - on conversation prediction task: **0.909.**
    - on single-message prediction task: **0.951.**

- **Recall (Sensitivity)**: Ability to correctly identify true positives (i.e., high-risk conversations\messages).
  - Expected Result:
    - on conversation prediction task: **above 0.8.**
    - on single-message prediction task: **above 0.85.**
  - Actual results with deployed Model:
    - on conversation prediction task: **0.797.**
    - on single-message prediction task: **0.682.**
  - Actual results with Ensemble Model:
    - on conversation prediction task: **0.718.**
    - on single-message prediction task: **0.753.**
- **Precision**: Ability to avoid false positives (i.e., flagging non-risky conversations as risky).
  - Expected Result:
    - on conversation prediction task: **above 0.7.**
    - on single-message prediction task: **above 0.75.**
  - Actual results with deployed Model:
    - on conversation prediction task: **0.638.**

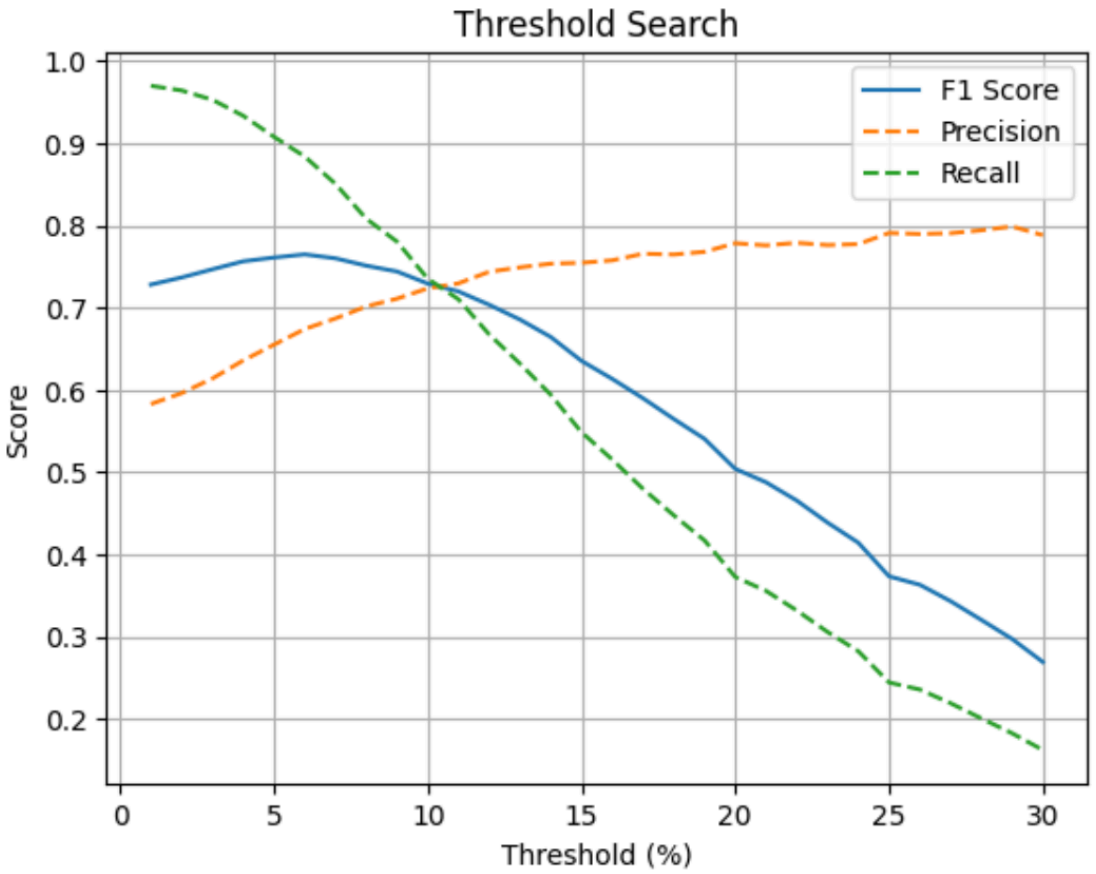
        - on single-message prediction task: **0.822.**
    - Actual results with Ensemble Model:
        - on conversation prediction task: **0.764.**
        - on single-message prediction task: **0.776.**
- **F1 Score**: Harmonic mean of precision and recall, providing a single score for balance.
    - Expected Result:
        - on conversation prediction task: **above 0.75.**
        - on single-message prediction task: **above 0.80.**
    - Actual results with deployed Model:
        - on conversation prediction task: **0.708.**
        - on single-message prediction task: **0.746.**
    - Actual results with Ensemble Model:
        - on conversation prediction task: **0.718.**
        - on single-message prediction task: **0.741.**
- **AUC (Area Under ROC Curve)**: Reflects the model's ability to distinguish between classes across thresholds.
    - Expected Result:
        - on conversation prediction task: **above 0.9.**
        - on single-message prediction task: **above 0.95.**
    - Actual results with deployed Model:
        - on conversation prediction task: **0.920.**
        - on single-message prediction task: **0.969.**
    - Actual results with Ensemble Model:
        - on conversation prediction task: **0.934.**
        - on single-message prediction task: **0.972.**

Message vs. Conversation-Level Generalization
Note: Conversation-level data is non-synthesized, but human labeled, making it more reliable for final evaluation.

for a more convenient look on things the following graph represent the different scors with different models (baseline represent a very basic model which just train on the full conversation task):



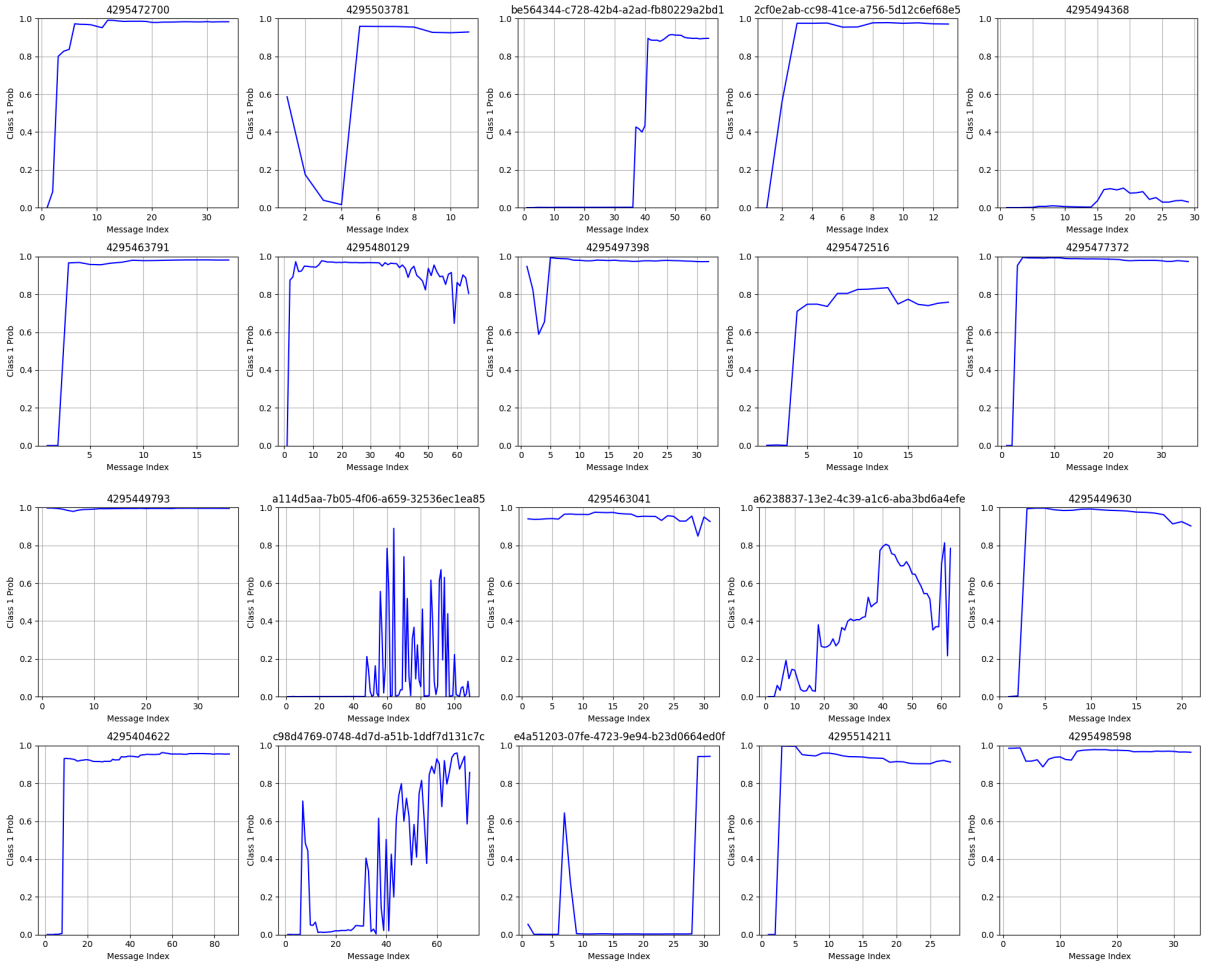
In addition, we introduce another model which works as a linear binary classifier. It takes a conversation and tries to assess the risk label based on the percentage of messages which are labeled as risky (this idea makes sense because we achieved very high results when trying to build a single message classifier which was trained only on the single message task). the following is a graph showing for each threshold (percentage of risky messages) what is the F1, recall, and precession results:



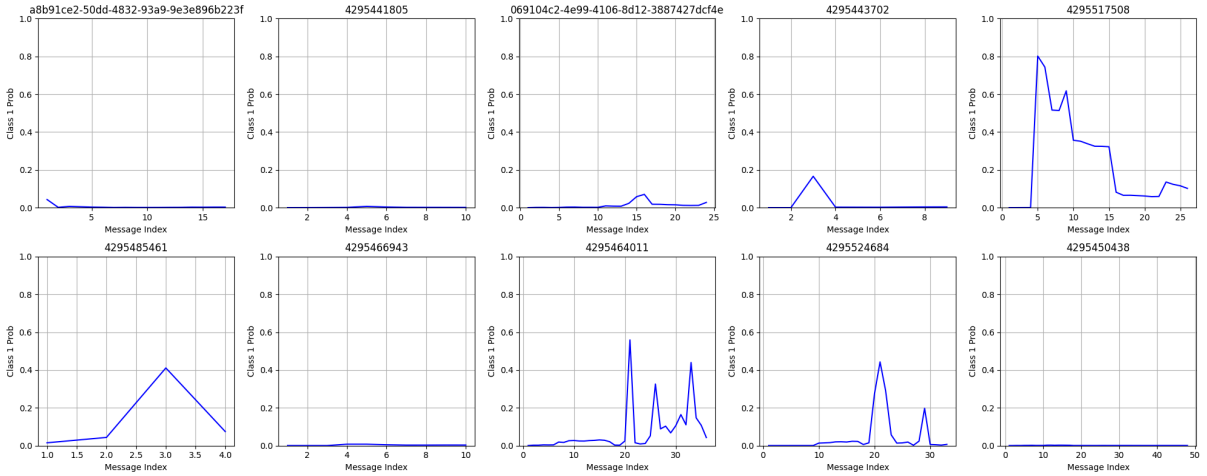
Finally we made the following analysis and sent the results to the Sahar association for professional-domain-specific assessment (yet to receive feedback):

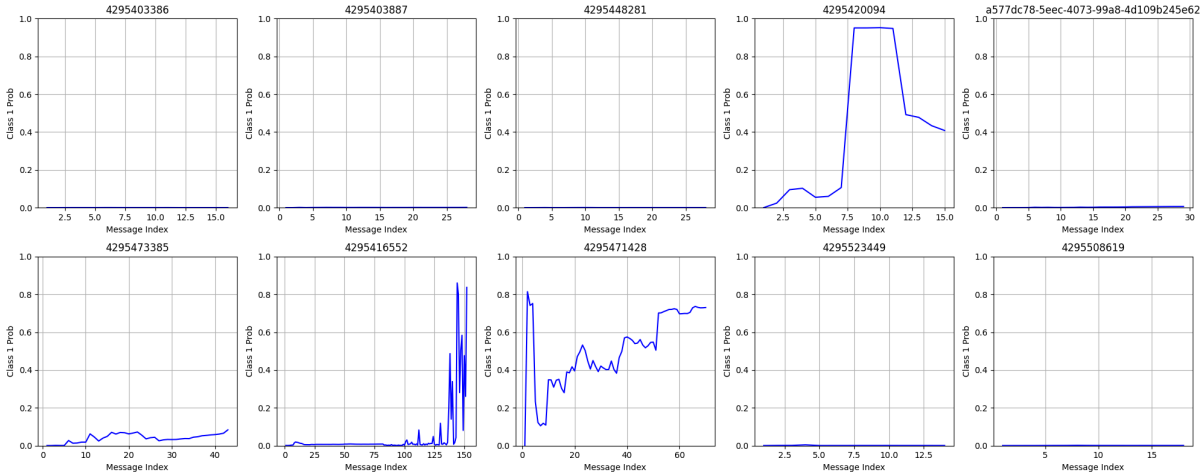
we took the deployed model and tested the how the risk level vary throughout the conversation for a small sample of conversations, that is the x-axis represents the number of messages processed so far (point n on the x-axis represent a prediction made on the first n messages concatenated together with the deployed) and y-axis is the risk level.

GSR:



NoSR:





Summarization model testing:

The summarization model generates abstractive summaries of full conversations, focusing on the core concern expressed in the chat.

Evaluation Methods:

- **Human Feedback:** Mental health professionals from the Share association will review the summaries and give feedback on the following matrices:
  - Faithfulness to original text
  - Relevance and coverage of critical issues (maintenance of the suicidal intent if one exists)
  - Readability and tone

  Expected Result: Professionals should generally rate summaries as faithful and concise, retaining key emotional and psychological cues while simplifying content for reviewers or clinicians.

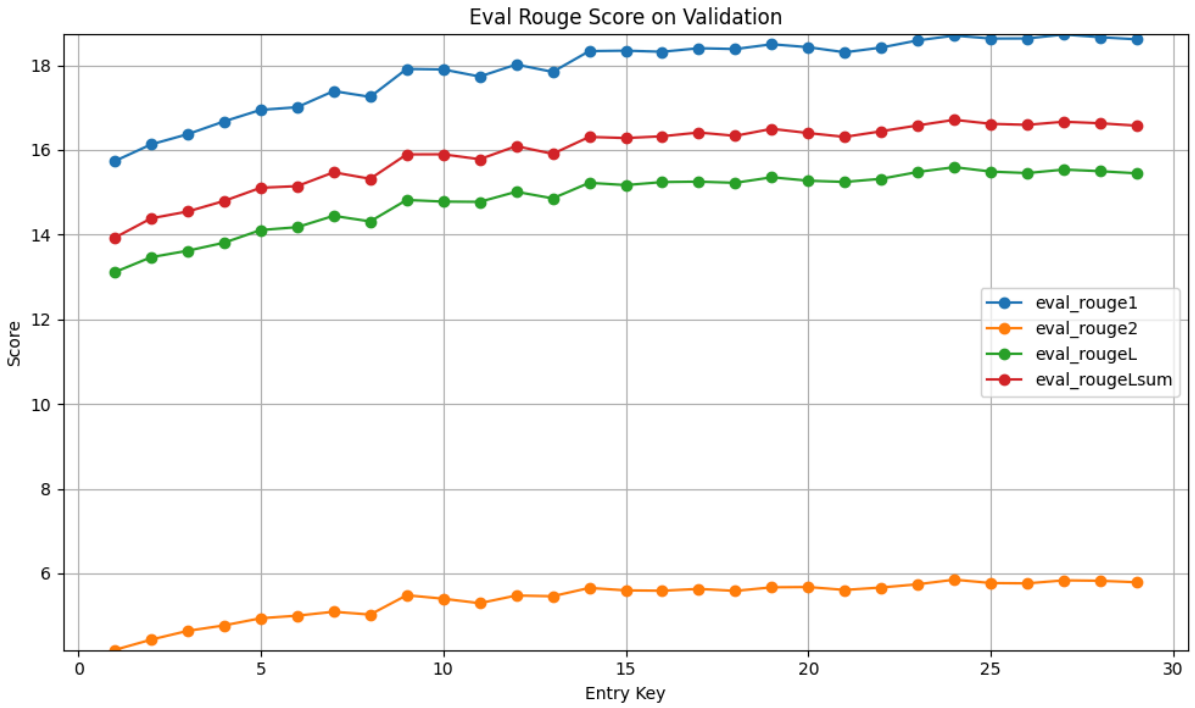
- **ROUGE Metrics:** we will also apply the ROUGE matric to assess the quality of the summaries produced by the model. we will test the following variation of the ROUGE score:
  - ROUGE 1
  - ROUGE 2
  - ROUGEL
  - ROUGELSum

  Expected Result: High ROUGE scores (around 0.4 on the ROUGE1 and 0.15 on ROUGE2)

Actual Results on the Test-Set:

- **ROUGE 1**: 26.013
- **ROUGE 2:** 7.7692
- **ROUGEL:** 20.0656
- **ROUGELSum:** 22.7229

Additionally the reader might find the following graph useful, the graph depicts the different ROUGE score on the validation test throughout the training process with respect to the amount of epochs preformed, seems to reach a plateau after 14-15 epochs:



Time performance testing:

performance testing on the deployed models to check inference time on a CPU architecture (the environment in which we deploy the model is CPU only).

Results:

- prediction model: ~1sec to make one prediction of a conversation.
- summarization model: ~3sec to make one summarization.

Additional Requirements for testing:

Most tests of statistical performance on the models will be performed in the BGU SLURM cluster and require no special hardware requirements (even the most basic allocation will suffice).

Server crash testing:

Test Case 1: Verify server recovers after a single crash

Objective: Confirm that the system service restarts successfully and becomes available after a crash.

Steps:

1. Start the Sahar service.
2. Wait 5 seconds to allow the service to fully start.
3. Confirm the service is active. (check it via terminal commands)
4. Get the Gunicorn process ID.
5. Kill the Gunicorn process.
6. Wait 7 seconds to allow the watchdog/systemd to restart the service.
7. Confirm again that the service is active.
8. Make a POST request to /login with valid credentials.
9. Confirm the response is OK (HTTP 200).

Expected Result:

- The system service automatically restarts.
- The server responds correctly to the /login API.
- Console shows: Server recovery test passed!

Test Case 2: Verify server handles multiple consecutive crashes

- **Objective**: Ensure the system recovers consistently after multiple crash events.
- **Steps**:
    1. Start the Sahar service.
    2. Wait 5 seconds.
    3. For 3 consecutive times:
        - Kill the Gunicorn process.
        - Wait 7 seconds.
        - Verify Sahar service is active.
        - Send a login POST request to /login as in Test Case 1.
        - Confirm HTTP 200 OK.
    4. Record results of each crash-recovery cycle.
- **Expected Result**:
    - All 3 crash events are automatically recovered.
    - Server becomes available at the /login endpoint after each recovery.
    - Console shows: Multiple crash recovery test passed!

Integration testing:

Test Case 1: Initialization of LpUtils with Valid Config and Successful Login

1. Purpose

To verify that the LpUtils class initializes correctly .

---

2. Requirements Addressed

- REQ-LP-001: The system shall retrieve a bearer token upon initialization of LpUtils.
- REQ-LP-002: The system shall extract configuration attributes from the configuration utility during initialization.
- REQ-LP-003: The system shall perform a login request using the extracted credentials.

---

3. Test Items

- LpUtils class in server_project.lp_api_manager.lp_utils
- Dependencies:
    - lp_login method.
    - ConfigUtil.get_config_attribute.
    - LpExecutor.execute.

---

4. Input Specifications

Mocked Config Attributes:

Attribute	Value
userName	test_user
password	test_pass

Attribute	Value
accountId	test_account_id
serviceLogInName	test_service_name
messageHistName	test_message_hist_name
numberOfRetriesAPICalls	0

Mocked Login Response:

- HTTP 200 OK
- JSON body: { "bearer": "test_bearer_token" }

Mocked execute Response:

- HTTP 200 OK

---

5. Output Specifications

Output	Expected Result
lp_utils.bearer_token	"test_bearer_token"
lp_login call status	Called with correct credentials

---

6. Environmental Needs

- Python unittest environment.
- unittest.mock.patch for dependency mocking.
- Access to LpUtils and its dependencies in codebase.

---

7. Procedures

1. Patch the following:
    - lp_login method to return a mocked successful login.
    - ConfigUtil.get_config_attribute to return test credentials.
    - LpExecutor.execute to simulate backend API call success.

2. Initialize the LpUtils class.
3. Assert that:
    - bearer_token is not None.
    - bearer_token equals "test_bearer_token".
4. Optionally, verify:
    - lp_login is called with "test_user", "test_pass", and "test_account_id".

---

8. Special Procedural Requirements

- Ensure this test is run in isolation or with controlled mocks to avoid side effects.

---

9. Intercase Dependencies

- Can be a prerequisite for token-dependent logic tests such as:
    - TC-INIT-002: Token Expiry Handling.
    - TC-AUTH-001: API Request with Token.

Test Case 2: Initialization of LpUtils with Valid Config and Successful Login

1. Purpose

To verify the successful and failure scenarios for the lp_login() method in the LpUtils class, ensuring correct behavior when a login is successful or when authentication fails.

---

2. Requirements Addressed

- **REQ-LPLOGIN-001:** The system shall return the correct login response when the credentials are valid.
- **REQ-LPLOGIN-002:** The system shall handle login failure correctly by returning the appropriate error message and status code.

---

3. Test Items

- Method under test: LpUtils.lp_login(username, password, account_id).
- Dependencies:
    - server_project.lp_api_manager.lp_utils.LpExecutor.execute.
    - server_project.lp_api_manager.lp_utils.LpApiConstants.login_call_uri and base_call_uri.
    - server_project.lp_api_manager.lp_utils.ConfigUtil.
    - server_project.lp_api_manager.lp_utils.LpUtils.system_authenticate.
    - server_project.lp_api_manager.lp_utils.LpUtils._extract_domain.
    - MagicMock (for mocking response and behaviors).

---

4. Input Specifications

Test Case 1 - Successful Login:

- **Username:** "test_user".
- **Password:** "test_pass".
- **Account ID:** "test_account_id".
- Mock the system_authenticate() to return a token "test_bearer_token".
- Mock the login URI and response to return a status code 200 with a valid token {"access_token": "test_token"}.

Test Case 2 - Failed Login:

- **Username:** "test_user".
- **Password:** "test_pass".
- **Account ID:** "test_account_id".
- Mock the login URI and response to return a status code 401 with an error message {"error": "Unauthorized"}.

---

5. Output Specifications

Test Case 1 - Successful Login:

- **Response Status Code:** 200
- **Response JSON:** {"access_token": "test_token"}
- **Verify:**
    - mock_execute is called once.
    - mock_execute returns the expected login response.

Test Case 2 - Failed Login:

- **Response Status Code:** 401
- **Response JSON:** {"error": "Unauthorized"}
- **Verify:**
    - mock_extract_domain is called.
    - mock_execute is called once.
    - The error message "Unauthorized" is present in the response.

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- MagicMock for mocking external dependencies.
- LpUtils class and related modules for integration.

---

7. Procedures

TC-LPLOGIN-001: LP Login Success Test

1. Mock the necessary dependencies:
    - system_authenticate() returns "test_bearer_token".
    - login_call_uri returns "https://api.liveperson.net/login".
    - execute() returns a mocked response with status 200 and JSON {"access_token": "test_token"}.
2. Create an instance of LpUtils.
3. Call the lp_login() method with valid credentials:
    - username = "test_user", password = "test_pass", account_id = "test_account_id".
4. Verify:
    - The response has status code 200.
    - The response JSON contains "access_token": "test_token".
    - mock_execute is called once.

TC-LPLOGIN-002: LP Login Failure Test

1. Mock the necessary dependencies:
    - system_authenticate() returns a valid token.
    - base_call_uri is mocked to return "https://api.test.com".
    - execute() returns a mocked response with status 401 and JSON {"error": "Unauthorized"}.
    - Mock _extract_domain() to simulate domain extraction.

2. Create an instance of LpUtils.
3. Call the lp_login() method with invalid credentials:
    - username = "test_user", password = "test_pass", account_id = "test_account_id".
4. Verify:
    - The response has status code 401.
    - The error message in the response contains "Unauthorized".
    - mock_extract_domain is called.
    - mock_execute is called once.

---

8. Special Procedural Requirements

- Mock the responses from external API calls, ensuring correct return values for both success and failure scenarios.
- Verify the correct handling of credentials and the response from the login API.

---

9. Pass/Fail Criteria

- Each test case passes if the expected login behavior occurs (correct status code and response JSON).
- Test cases fail if the behavior deviates from the expected login success or failure handling.

Test Case 3: Tests for get_conversations_by_conv_id() in LpUtils.

1. Purpose

To validate the functionality of the get_conversations_by_conv_id() method in the LpUtils class, ensuring it correctly handles both successful and failed scenarios when retrieving conversation details by their IDs.

---

2. Requirements Addressed

- **REQ-LPGETCONV-001:** The system shall successfully return conversation details when the request is valid.
- **REQ-LPGETCONV-002:** The system shall handle failure scenarios correctly, logging the appropriate errors when the response indicates a failure.

---

3. Test Items

- Method under test: LpUtils.get_conversations_by_conv_id(conversation_ids).
- Dependencies:
    - server_project.lp_api_manager.lp_utils.LpExecutor.execute.
    - server_project.lp_api_manager.lp_utils.ConfigUtil.
    - server_project.lp_api_manager.lp_utils.LpUtils.check_response
    - server_project.lp_api_manager.lp_utils.LpUtils.lp_login.
    - Logging mechanism (for error logging).
    - MagicMock (for mocking responses and behaviors).

---

4. Input Specifications

Test Case 1 - Successful Retrieval of Conversations:

- **Conversation IDs:** ["1234"]

- Mock the necessary configurations and responses:
    - lp_login() returns a bearer token: {"bearer": "test_bearer_token"}.
    - execute() returns a mocked response with status code 200 and conversation history.
    - check_response() returns the mocked response data with valid conversation records.

Test Case 2 - Failure Scenario (Internal Server Error):

- **Conversation IDs:** A list of 101 conversation IDs (to simulate the failure due to exceeding the limit).
- Mock the necessary configurations and responses:
    - lp_login() returns a bearer token: {"bearer": "test_bearer_token"}.
    - execute() returns a mocked response with status code 500 and error details: {"error": "Internal Server Error"}.
    - check_response() returns error data.
    - Log an error message due to exceeding the maximum number of allowed conversation IDs.

---

5. Output Specifications

Test Case 1 - Successful Retrieval of Conversations:

- **Response:**
    - Contains "conversationHistoryRecords".
    - The first conversation record has "conversationId": "1234".
    - The conversation record status is "CLOSE".
- **Verify:**
    - mock_execute is called once.
    - The response contains conversation history.
    - mock_check_response correctly processes the response data.

Test Case 2 - Failure Scenario (Internal Server Error):

- **Response:**
    - Status code: 500.

    - Error message: {"error": "Internal Server Error"}.

- **Verify:**
    - mock_logging_error is called with the message: "conversationsIds size > 100, data lost".
    - mock_execute is called once.
    - The system correctly logs an error for the failed request.

---

6. Environmental Needs

- Python 3.x
- unittest and unittest.mock.
- MagicMock for mocking external dependencies.
- LpUtils class and related modules for integration.
- Logging mechanism for error messages.

---

7. Procedures

TC-LPGETCONV-001: Get Conversations by Conv ID Success

1. Mock the necessary dependencies:
    - lp_login() returns the token {"bearer": "test_bearer_token"}.
    - execute() returns a mocked response with status code 200 and valid conversation data.
    - check_response() returns valid data.
2. Create an instance of LpUtils.
3. Call get_conversations_by_conv_id() with conversation_ids = ["1234"].
4. Verify:
    - The response contains "conversationHistoryRecords".
    - The first conversation record has "conversationId": "1234".
    - The status of the conversation is "CLOSE".

- mock_execute is called once.
- mock_check_response returns valid data.

TC-LPGETCONV-002: Get Conversations by Conv ID Failure

1. Mock the necessary dependencies:
   - lp_login() returns the token {"bearer": "test_bearer_token"}.
   - execute() returns a mocked response with status 500 and error data.
   - check_response() returns error data.
2. Create an instance of LpUtils.
3. Call get_conversations_by_conv_id() with conversation_ids = [str(i) for i in range(1, 102)] (exceeds 100 IDs).
4. Verify:
   - The response has status 500.
   - The error message in the response is "Internal Server Error".
   - The error log message "conversationsIds size > 100, data lost" is logged.
   - mock_execute is called once.

---

8. Special Procedural Requirements

- Mock the responses from external API calls, ensuring correct return values for both success and failure scenarios.
- Verify that error logs are properly recorded when the number of conversation IDs exceeds the limit.

---

9. Pass/Fail Criteria

- Each test case passes if the expected behavior occurs (correct handling of both success and failure cases, and correct logging).
- Test cases fail if the response deviates from the expected behavior or logging does not occur when required.

Test Case 4: Tests for system_authenticate() in LpUtils

1. Purpose

To validate the functionality of the system_authenticate() method in the LpUtils class, ensuring it correctly handles both successful and failed authentication scenarios.

---

2. Requirements Addressed

- **REQ-LPSYSAUTH-001:** The system shall return the bearer token if authentication is successful.
- **REQ-LPSYSAUTH-002:** The system shall handle failed authentication scenarios by logging an appropriate error message.

---

3. Test Items

- Method under test: LpUtils.system_authenticate().
- Dependencies:
    - server_project.lp_api_manager.lp_utils.LpExecutor.execute.
    - server_project.lp_api_manager.lp_utils.ConfigUtil.
    - server_project.lp_api_manager.lp_utils.LpUtils.lp_login.
    - server_project.lp_api_manager.lp_utils.LpUtils.is_response_ok.
    - Logging mechanism (for error logging).
    - MagicMock (for mocking responses and behaviors).

---

4. Input Specifications

Test Case 1 - Successful Authentication:

- **Username:** test_user.
- **Password:** test_pass.
- **Account ID:** test_account_id.

- Mock the following behaviors:
    - lp_login() returns a mocked response with status code 200 and a JSON body containing the bearer token: {"bearer": "test_bearer_token"}.
    - execute() returns a mocked response with status code 200.

Test Case 2 - Failed Authentication (Unauthorized):

- **Username:** test_user.
- **Password:** test_pass.
- **Account ID:** test_account_id.
- Mock the following behaviors:
    - lp_login() returns a mocked response with status code 401 and error data: {"error": "Unauthorized"}.
    - is_response_ok() returns False (indicating an unsuccessful login attempt).
    - The error message "system_authenticate: login failed" is logged.

---

5. Output Specifications

Test Case 1 - Successful Authentication:

- **Response:** The method should return the bearer token "test_bearer_token".
- **Verify:**
    - mock_execute is called once.
    - mock_lp_login is called with the correct parameters.
    - The result is the bearer token "test_bearer_token".

Test Case 2 - Failed Authentication (Unauthorized):

- **Response:** The method should return None due to authentication failure.
- **Verify:**
    - mock_lp_login is called with the correct parameters.
    - mock_is_response_ok is called and returns False.

- The error log message "system_authenticate: login failed" is logged.
- The result is None.

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- MagicMock for mocking external dependencies.
- LpUtils class and related modules for integration.
- Logging mechanism for error messages.

---

7. Procedures

TC-LPSYSAUTH-001: System Authenticate Success

1. Mock the necessary dependencies:
    - lp_login() returns a successful response with bearer token: {"bearer": "test_bearer_token"}.
    - execute() returns a mocked response with status code 200.
2. Create an instance of LpUtils.
3. Call system_authenticate().
4. Verify:
    - The method returns the bearer token "test_bearer_token".
    - mock_execute is called once.
    - mock_lp_login is called with the correct username, password, and account ID.

TC-LPSYSAUTH-002: System Authenticate Failure (Unauthorized)

1. Mock the necessary dependencies:
    - lp_login() returns a mocked response with status code 401 and error data: {"error": "Unauthorized"}.
    - is_response_ok() returns False.

    - execute() returns a mocked response with status code 200.
    - Logging error is triggered with the message "system_authenticate: login failed".
2. Create an instance of LpUtils.
3. Call system_authenticate().
4. Verify:
    - The method returns None due to the failed authentication.
    - mock_lp_login is called with the correct username, password, and account ID.
    - mock_is_response_ok is called and returns False.
    - The error message "system_authenticate: login failed" is logged.

---

8. Special Procedural Requirements

- Mock the responses from external API calls to simulate both successful and failed authentication scenarios.
- Verify that the system correctly handles both success and failure scenarios, and logs errors when necessary.

---

9. Pass/Fail Criteria

- **Pass:** Each test case passes if the expected behavior occurs:
    - For success, the method should return the bearer token.
    - For failure, the method should return None and log the appropriate error message.
- **Fail:** A test case fails if the response deviates from the expected behavior or if the error logging does not occur when required.

Test Case 5: Successful Extraction of Conversation Data

1. Purpose

To verify that the extract_conversations() method in LpUtils correctly parses a successful JSON response from the conversation history API and returns the expected data in a structured format.

---

2. Requirements Addressed

- **REQ-CONV-001:** The system shall extract conversation data from a valid JSON response.
- **REQ-CONV-002:** The system shall preserve mapping of conversation IDs to conversation objects.
- **REQ-CONV-003:** The system shall extract relevant consumer and agent information from each record.

---

3. Test Items

- LpUtils.extract_conversations()
- Dependencies:
    - ConfigUtil.get_config_attribute
    - LpUtils.lp_login
    - LpExecutor.execute

---

4. Input Specifications

Mocked Configuration Values:

Attribute	Value
userName	test_user
password	test_pass

Attribute	Value
accountId	test_account_id
serviceLogInName	test_service_name
messageHistName	test_message_hist_name
numberOfRetriesAPICalls	"0"

Mocked API Response JSON:


```
{
  "conversationHistoryRecords": [
    {
      "info": {
        "startTime": "2024-12-14T12:00:00Z",
        "startTimeL": 1702555200000,
        "conversationId": "1234",
        "latestAgentId": "agent_5678",
        "fullDialogStatus": "active",
        "latestAgentFullName": "Agent Smith",
        "status": "OPEN"
      },
      "consumerParticipants": [
        {
          "participantId": "consumer_001",
          "firstName": "John",
          "lastName": "Doe",
          "token": "sample_token",
          "email": "johndoe@example.com",
          "phone": "+123456789",
          "avatarURL": "https://example.com/avatar.jpg",
```

```
        "time": "2024-12-14T12:01:00Z",
        "timeL": 1702555260000,
        "consumerName": "John Doe",
        "dialogId": "dialog_5678"
      }
    ],
    "monitoring": { "status": "active" }
  }
 ]
}
```


---

5. Output Specifications

Output	Expected Value
Return type	dict[str, ConversationObject]
Number of conversations extracted	1
Key in result	"1234"
conversationId in output object	"1234"
consumerParticipants.firstName value	"John"

---

6. Environmental Needs

- Python 3.x.
- unittest module.
- unittest.mock for patching and mocking dependencies.
- Working implementation of LpUtils and ConversationObject (or equivalent).

---

7. Procedures

1. Patch:
    - ConfigUtil to return predefined attributes.

    - lp_login to simulate a successful login.
    - LpExecutor.execute to return a mock response with conversation history.
2. Simulate a response with one conversation record and one participant.
3. Call LpUtils.extract_conversations(response_json).
4. Assert:
    - The return type is a dictionary with one element.
    - The conversation ID 1234 is a key in the dictionary.
    - The returned object contains correct details for conversationId and participant name.

---

8. Special Procedural Requirements

- Assumes successful login and valid response JSON structure.
- The ConversationObject structure or returned type must support attribute access (e.g., .conversationId, .consumerParticipants.firstName).

---

9. Intercase Dependencies

- None required; however, logically follows a successful retrieval of conversation history (e.g., test_get_conversations_by_conv_id_success).

Test Case 6: Extraction of Base URI from API Responses

1. Purpose

To verify that the _extract_base_uri method in LpUtils correctly extracts the base URI for a given service name, handling both success and failure scenarios.

---

2. Requirements Addressed

- **REQ-LP-001**: The system shall extract the correct base URI when a valid service name is provided.
- **REQ-LP-002**: The system shall return None when an invalid service name is provided.
- **REQ-LP-003**: The system shall handle both single and multiple base URI responses.
- **REQ-LP-004**: The system shall return the correct URI even when only one service is available.

---

3. Test Items

- **LpUtils._extract_base_uri()**
- **Dependencies**:
  - ConfigUtil.get_config_attribute
  - LpUtils.lp_login
  - LpExecutor.execute

---

4. Input Specifications

Mocked Configuration Values:

Attribute	Value
userName	test_user.
password	test_pass.

Attribute	Value
accountId	test_account_id.
serviceLogInName	test_service_name.
messageHistName	test_message_hist_name.
numberOfRetriesAPICalls	"0"

Mocked API Response JSON (for all tests):

```
{
  "baseURIs": [
    {"service": "service1", "baseURI": "https://api.service1.com"},
    {"service": "service2", "baseURI": "https://api.service2.com"}
  ]
}
```

Mocked API Response JSON for single service scenario (for test_extract_base_uri_single_uri_success):

```
json
CopyEdit
{
  "service": "service1",
  "baseURI": "https://api.service1.com"
}
```

---

5. Output Specifications

Output	Expected Value
Return type	Str.
Base URI for service1	"https://api.service1.com".
Base URI for service2	"https://api.service2.com".

Output	Expected Value
Base URI for invalid service	None

---

6. Environmental Needs

- Python 3.x.
- unittest module.
- unittest.mock for patching and mocking dependencies.
- Working implementation of LpUtils and MagicMock for mock API responses.

---

7. Procedures

Test Case 1: test_extract_base_uri_success

1. **Patch**:
    - ConfigUtil to return predefined attributes for configuration.
    - lp_login to simulate a successful login and return a mock bearer token.
    - LpExecutor.execute to return a mock response with base URI information.
2. **Simulate Response**:
    - Provide a response with multiple services (service1, service2).
3. **Call**:
    - Call LpUtils._extract_base_uri() with a valid service name ("service1").
4. **Assert**:
    - Ensure the returned base URI is "https://api.service1.com".

Test Case 2: test_extract_base_uri_failure

1. **Patch**:
    - Mock the same dependencies as in the success case.

2. **Simulate Response**:
    - Provide a response with multiple services (service1, service2).

3. **Call**:
    - Call LpUtils._extract_base_uri() with an invalid service name ("service3").
4. **Assert**:
    - Ensure the returned value is None.

Test Case 3: test_extract_base_uri_single_uri_success

1. **Patch**:
    - Mock the same dependencies as in the success case.
2. **Simulate Response**:
    - Provide a response with a single service (service1).
3. **Call**:
    - Call LpUtils._extract_base_uri() with the service name ("service1").
4. **Assert**:
    - Ensure the returned base URI is "https://api.service1.com".

Test Case 4: test_extract_base_uri_single_uri_failure

1. **Patch**:
    - Mock the same dependencies as in the success case.
2. **Simulate Response**:
    - Provide a response with a single service (service2).
3. **Call**:
    - Call LpUtils._extract_base_uri() with an invalid service name ("service1").
4. **Assert**:
    - Ensure the returned value is None.

---

8. Special Procedural Requirements

- Assumes a valid mock response is returned with correct service URIs.
- The mock configuration must match the expected setup for API calls to succeed.
- The method _extract_base_uri must return the correct value or None depending on the provided service name.

---

9. Intercase Dependencies

- None required, but logically follows the behavior of methods interacting with external APIs (e.g., test_lp_login_success).

Test Case 7: Extraction of Domain from Service API

1. Purpose

To verify that the _extract_domain() method in LpUtils correctly extracts the domain URL for a given service, handling both success and failure scenarios.

---

2. Requirements Addressed

- **REQ-LP-005**: The system shall extract the correct domain URL for a valid service.
- **REQ-LP-006**: The system shall log a warning for errors in API response and raise appropriate exceptions.
- **REQ-LP-007**: The system shall handle server errors appropriately and raise exceptions.

---

3. Test Items

- **LpUtils._extract_domain().**
- **Dependencies**:
    - ConfigUtil.get_config_attribute.
    - LpUtils.lp_login.
    - LpExecutor.execute.
    - LpUtils.check_response.
    - logging.warning.

---

4. Input Specifications

Mocked Configuration Values:

Attribute	Value
userName	test_user.
password	test_pass.
accountId	test_account_id.
serviceLogInName	test_service_name.
messageHistName	test_message_hist_name.
numberOfRetriesAPICalls	"0".

Mocked API Response JSON (for all tests):

json

CopyEdit

```
{
  "baseURIs": [{"service": "test_service_name", "baseURI": "https://api.test.com/v1"}]
}
```

---

5. Output Specifications

Output	Expected Value
Return type	Str.
Extracted domain for service	"https://api.test.com/v1".
Exception message (in failure)	"Internal Server Error".

---

6. Environmental Needs

- Python 3.x.
- unittest module.
- unittest.mock for patching and mocking dependencies.

- Working implementation of LpUtils, MagicMock for mock API responses, and logging.warning for handling warnings.

---

7. Procedures

Test Case 1: test_extract_domain_success

1. **Patch**:
    - ConfigUtil to return predefined attributes for configuration.
    - lp_login to simulate a successful login and return a mock bearer token.
    - LpExecutor.execute to return a mock response with the base URI information.
    - LpUtils.check_response to ensure the response is processed.
    - LpApiConstants.base_call_uri to set the expected base URI format.
2. **Simulate Response**:
    - Provide a mock response with the service name (test_service_name) and base URI ("https://api.test.com/v1").
3. **Call**:
    - Call LpUtils._extract_domain() with the valid service name ("test_service_name").
4. **Assert**:
    - Ensure the extracted domain is "https://api.test.com/v1".
    - Ensure mock_execute and mock_check_response were called correctly.

Test Case 2: test_extract_domain_failure

1. **Patch**:

- Mock the dependencies for configuration and login as in the success case.
- Mock LpExecutor.execute to simulate a server error (status code 500) with a generic error response.
- Mock LpUtils.check_response to raise an exception when the response is unsuccessful.

2. **Simulate Response**:
   - Provide a mock response with status code 500 and an error message ({"error": "Internal Server Error"}).
3. **Call**:
   - Call LpUtils._extract_domain() with the valid service name ("test_service_name").
4. **Assert**:
   - Ensure that the exception is raised with the message "Internal Server Error".
   - Ensure that mock_logging_warning was not called (indicating proper error handling).
5. **Exception Handling**:
   - Ensure that the exception is raised and the test fails if it does not.

---

8. Special Procedural Requirements

- Assumes that mock responses from the API are correctly formatted.
- Assumes that logging.warning is used to log warnings for certain API failures.
- Assumes that the error handling mechanism raises exceptions correctly for server errors (e.g., 500 status code).

---

9. Intercase Dependencies

- None required, though logically follows the behavior of handling successful and failed API responses (e.g., test_lp_login_success, test_execute_mock_response).

Test Case 8: Extract Messages from Conversation History Records

1. Purpose

To verify that the extract_messages() method in the LpUtils class correctly extracts and formats message records from a given conversation history response.

---

2. Requirements Addressed

- **REQ-MESSAGE-001:** The system shall correctly parse message records from a conversation history response.
- **REQ-MESSAGE-002:** The system shall return a dictionary where the key is the conversation ID and the value is a list of MessageRecord objects corresponding to each message in the conversation.

---

3. Test Items

- Method under test: LpUtils.extract_messages(response_data).
- Dependencies:
  - LpUtils.lp_login.
  - ConfigUtil.get_config_attribute.
  - LpExecutor.execute.

---

4. Input Specifications

Mocked Config Values:

Attribute	Value
userName	test_user
password	test_pass
accountId	test_account_id
serviceLogInName	test_service_name
messageHistName	test_message_hist_name
numberOfRetriesAPICalls	"0".

Mocked Response Data:


```
{
  "conversationHistoryRecords": [
    {
      "info": {
        "conversationId": "12345",
        "status": "OPEN",
        "startTime": "2024-12-01T12:00:00Z"
      },
      "messageRecords": [
        {
          "messageData": {"content": "Hello"},
          "seq": 1,
          "time": "2024-12-01T12:01:00Z",
          "timeL": 1700122860000,
          "messageId": "msg_001",
          "sentBy": "agent"
        },
        {
```

```
        "messageData": {"content": "How can I help you?"},
        "seq": 2,
        "time": "2024-12-01T12:02:00Z",
        "timeL": 1700122920000,
        "messageId": "msg_002",
        "sentBy": "agent"
      }
    ]
  }
 ]
}
```

---

5. Output Specifications

Expected Output:

```
{
 "12345": {
   MessageRecord(
     messageData={"content": "Hello"},
     seq=1,
     time="2024-12-01T12:01:00Z",
     timeL=1700122860000,
     messageId="msg_001",
     sentBy="agent"
   ),
   MessageRecord(
     messageData={"content": "How can I help you?"},
     seq=2,
     time="2024-12-01T12:02:00Z",
```

```
      timeL=1700122920000,
      messageId="msg_002",
      sentBy="agent"
    )
  }
}
```

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- Functional LpUtils class and MessageRecord object model.

---

7. Procedures

1. Mock the necessary dependencies like config values and login response.
2. Provide a mocked response with conversation history records and message records.
3. Call LpUtils.extract_messages() with the mocked response data.
4. Verify that the returned result matches the expected structure, with conversation IDs and corresponding message records.
5. Ensure that the MessageRecord objects are constructed correctly for each message in the conversation.

---

8. Special Procedural Requirements

- Assumes that the MessageRecord class is implemented and can correctly handle data passed to it.
- Verifies that the function returns a dictionary of conversation IDs to MessageRecord objects in the correct format.

---

9. Intercase Dependencies

- This test complements other tests that check the login process and message parsing in other contexts, but it stands alone for verifying message extraction.

Test Case 9: Fetch Open Conversations from LivePerson API

1. Purpose

To verify that the get_open_conversations() method in LpUtils correctly retrieves open conversation data from the LivePerson API, handling both success and failure scenarios.

---

2. Requirements Addressed

- **REQ-LP-008**: The system shall retrieve open conversation records for the given time range and other parameters.
- **REQ-LP-009**: The system shall handle API failures, including unauthorized errors (status code 401), appropriately.
- **REQ-LP-010**: The system shall construct the correct API request with headers, body, and parameters.

---

3. Test Items

- **LpUtils.get_open_conversations()**
- **Dependencies**:
    - ConfigUtil.get_config_attribute.
    - LpUtils.lp_login.
    - LpExecutor.execute.
    - LpUtils.check_response.

- LpApiBuilder.
- LpApiConstants.message_hist_uri.

---

4. Input Specifications

Mocked Configuration Values:

Attribute	**Value**
userName	test_user.
password	test_pass.
accountId	test_account_id.
serviceLogInName	test_service_name.
messageHistName	test_message_hist_name.
numberOfRetriesAPICalls	"0".

Mocked API Response JSON (for success):


```
{
  "conversationHistoryRecords": [
    {
      "info": {
        "conversationId": "12345",
        "status": "OPEN",
        "startTime": "2024-12-01T12:00:00Z"
      },
      "consumerParticipants": [
        {
          "participantId": "consumer_001",
          "consumerName": "John Doe",
```

```
        "dialogId": "dialog_5678"
      }
    ],
    "monitoring": {"status": "active"}
  }
 ]
}
```


Mocked API Response JSON (for failure):


```
{
  "error": "Unauthorized"
}
```


---

5. Output Specifications

Output	Expected Value
Return type	dict (response JSON).
Open conversations retrieved	conversation history records.
Exception message (failure)	"API call failed with status 401".

---

6. Environmental Needs

- Python 3.x.
- unittest module.
- unittest.mock for patching and mocking dependencies.
- Working implementation of LpUtils, MagicMock for mock API responses.

---

7. Procedures

Test Case 1: test_get_open_conversations_success

1. **Patch**:
    - Mock the ConfigUtil to return the necessary configuration values.
    - Mock lp_login to simulate a successful login and return a mock bearer token.
    - Mock LpExecutor.execute to simulate a successful API response with open conversation data.
    - Mock LpApiBuilder to verify the API call construction (headers, body, and parameters).
    - Mock LpApiConstants.message_hist_uri to return the correct message history URI.
2. **Simulate Response**:
    - Provide a mock response with a successful status code (200) and a conversation record containing the relevant fields.
3. **Call**:
    - Call LpUtils.get_open_conversations() with predefined start_time, end_time, and offset values.
4. **Assert**:
    - Ensure that the response matches the expected open conversation data.
    - Verify that message_hist_uri, add_headers, add_body, and add_params are called with the correct parameters.
    - Verify that the execute method is called with the correct API call object.
    - Verify that the check_response method is called to validate the response.

Test Case 2: test_get_open_conversations_failure

1. **Patch**:
    - Mock the ConfigUtil to return the necessary configuration values.
    - Mock lp_login to simulate a successful login and return a mock bearer token.

    - Mock LpExecutor.execute to simulate a failed API response with status code 401 and an error message (Unauthorized).
    - Mock LpApiBuilder and LpApiConstants.message_hist_uri to ensure the correct API call is constructed.
2. **Simulate Response**:
    - Provide a mock response with a failed status code (401) and an error message (Unauthorized).
3. **Call**:
    - Call LpUtils.get_open_conversations() with predefined start_time, end_time, and offset values.
4. **Assert**:
    - Ensure that an exception is raised with the message "API call failed with status 401".
    - Verify that the message_hist_uri, add_headers, add_body, and add_params methods are called with the correct parameters.
    - Verify that execute is called with the correct API call object.
    - Verify that check_response is called to validate the response.

---

8. Special Procedural Requirements

- Assumes that mock responses from the API are correctly formatted.
- Assumes that logging.warning or other logging methods are used to handle warnings or errors in the API response.

---

9. Intercase Dependencies

- None required, although it depends on correct configuration setup, API response handling, and method interactions.

Test Case 10: Check Response Failure Handling for Unauthorized, Non-JSON, and Missing Headers

1. Purpose

To verify that the check_response() method in the LpUtils class handles various failure scenarios appropriately, including unauthorized responses, non-JSON responses, and responses with missing headers.

---

2. Requirements Addressed

- **REQ-RESPONSE-001:** The system shall handle Unauthorized API responses by logging a warning and returning the response's JSON content.
- **REQ-RESPONSE-002:** The system shall handle non-JSON responses by returning the raw content.
- **REQ-RESPONSE-003:** The system shall handle responses with missing headers by returning the raw content.

---

3. Test Items

- Method under test: LpUtils.check_response(response, is_need_authentication=False).
- Dependencies:
    - LpUtils.lp_login.
    - LpExecutor.execute.

- logging.warning.

---

4. Input Specifications

Mocked Config Values:

Attribute	Value
userName	test_user
password	test_pass
accountId	test_account_id
serviceLogInName	test_service_name
messageHistName	test_message_hist_name
numberOfRetriesAPICalls	"0"

Mocked API Response Data (for different scenarios):

1. **Unauthorized Response (401 Error)**:

```
{
  "error": "Unauthorized"
}
```

2. **Non-JSON Response (text/plain)**:
   - Content: "Unexpected Content".
3. **Missing Headers Response (No headers)**:
   - Content: "Server Error".
4. **Valid JSON Response (200 OK)**:

```
{
  "success": true
}
```

---

5. Output Specifications

Expected Outputs:

- For Unauthorized Response: Logs a warning and returns the error message: {"error": "Unauthorized"}.
- For Non-JSON Response: Returns the raw content: "Unexpected Content".
- For Missing Headers Response: Returns the raw content: "Server Error".
- For Valid JSON Response: Returns the parsed JSON content: {"success": true}.

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- Functional LpUtils class, logging module.

---

7. Procedures

1. Mock the necessary dependencies, including the configuration values and login response.
2. Mock various API responses to simulate different failure and success scenarios:
    - Unauthorized response with a 401 error.
    - Non-JSON response with text/plain content.
    - Missing headers response.
    - Valid JSON response with status 200.
3. Call the check_response() method for each mocked response and assert that the system handles each case appropriately:
    - Logs a warning for the unauthorized response.
    - Returns raw content for non-JSON responses and missing headers.
    - Returns parsed JSON for valid JSON responses.

---

8. Special Procedural Requirements

- Assumes that the check_response() method is designed to handle different response types, including error handling, JSON parsing, and logging.
- Verifies the logging of warnings and correct parsing/handling of API responses.

---

9. Intercase Dependencies

- This test complements other tests that verify the API response handling and authentication flow but specifically tests the failure handling in the check_response() method.

Test Case 11: Log Message Tests for LpExecutor

1. Purpose

To verify the correct logging behavior of the _log() method in the LpExecutor class for various conditions, such as success, failure, last attempts, and cases with or without a prepared request.

---

2. Requirements Addressed

- **REQ-LOGGER-001:** The system shall log the correct warning when the attempt is not the last attempt and there is a failure.
- **REQ-LOGGER-002:** The system shall log an error when the attempt is the last attempt and the request fails.
- **REQ-LOGGER-003:** The system shall handle both cases where the prepared_request is provided and where it is not.

---

3. Test Items

- Method under test: LpExecutor._log(current_try, number_of_retries, message, prepared_request).
- Dependencies:
    - logging.error.

    - logging.warning.
    - MagicMock (for mocking request and response).
    - LpExecutor class.

---

4. Input Specifications

Each test case uses a mocked prepared_request object or None as the request and simulates specific logging scenarios for both success and failure.

---

5. Output Specifications

- **Test Case 1 (TC-LOGGER-001)**: The system should log a warning with the message "Number of retries: 1, Message: Request failed".
- **Test Case 2 (TC-LOGGER-002)**: The system should log an error containing "Error message: Request failed" and should log this error twice (since it's the last attempt).
- **Test Case 3 (TC-LOGGER-003)**: The system should log a warning with the message "Number of retries: 1, Message: Request failed".
- **Test Case 4 (TC-LOGGER-004)**: The system should log an error containing "Error message: Request failed" when prepared_request is None and it's the last attempt.
- **Test Case 5 (TC-LOGGER-005)**: The system should log a warning with the message "Number of retries: 1, Message: Request failed" when prepared_request is None and it's not the last attempt.

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- logging module for capturing logs.

---

7. Procedures

TC-LOGGER-001: Log Success Test for Non-Last Attempt Failure with Prepared Request

1. Create a mock prepared_request with the following attributes:
    - headers={'Content-Type': 'application/json'}, body='{"key": "value"}', url='https://test.api.com', method='GET'.
2. Create an instance of LpExecutor.
3. Call _log() method with current_try=1, number_of_retries=3, message='Request failed', and the prepared_request.
4. Verify that logging.warning was called once with the message "Number of retries: 1, Message: Request failed".
5. Verify that logging.error was not called.

TC-LOGGER-002: Log Last Attempt Failure Test with Prepared Request

1. Create a mock prepared_request with the following attributes:
    - headers={'Content-Type': 'application/json'}, body='{"key": "value"}', url='https://test.api.com', method='GET'.
2. Create an instance of LpExecutor.
3. Call _log() method with current_try=2, number_of_retries=3, message='Request failed', and the prepared_request.
4. Verify that logging.error was called twice, with the message "Error message: Request failed".
5. Verify that logging.warning was not called.

TC-LOGGER-003: Log Non-Last Attempt Failure Test with Prepared Request

1. Create a mock prepared_request with the following attributes:
    - headers={'Content-Type': 'application/json'}, body='{"key": "value"}', url='https://test.api.com', method='GET'.
2. Create an instance of LpExecutor.
3. Call _log() method with current_try=1, number_of_retries=3, message='Request failed', and the prepared_request.
4. Verify that logging.warning was called once with the message "Number of retries: 1, Message: Request failed".

5. Verify that logging.error was not called.

TC-LOGGER-004: Log Last Attempt Failure Test with No Prepared Request

1. Create an instance of LpExecutor.
2. Call _log() method with current_try=2, number_of_retries=3, message='Request failed', and prepared_request=None.
3. Verify that logging.error was called once with the message "Error message: Request failed".
4. Verify that logging.warning was not called.

TC-LOGGER-005: Log Non-Last Attempt Failure Test with No Prepared Request

1. Create an instance of LpExecutor.
2. Call _log() method with current_try=1, number_of_retries=3, message='Request failed', and prepared_request=None.
3. Verify that logging.warning was called once with the message "Number of retries: 1, Message: Request failed".
4. Verify that logging.error was not called.

---

8. Special Procedural Requirements

- Mock logging methods (logging.error, logging.warning) to validate the output messages.
- Verify logging behavior based on current_try and number_of_retries with and without a prepared_request.

---

9. Pass/Fail Criteria

- Each test case passes if the expected logging behavior (either warning or error with the appropriate message) is observed.
- Test cases fail if the actual logging behavior deviates from the expected behavior.

Test Case 12: Execute API Call Tests for LpExecutor

1. Purpose

To verify that the execute() method in the LpExecutor class behaves correctly under various conditions, including success and different error scenarios (e.g., 404, 401, 429, 500, etc.). The method will be tested for correct logging, retry handling, and proper responses in different scenarios.

---

2. Requirements Addressed

- **REQ-EXECUTE-001:** The system shall return a valid response when the API call is successful (status code 200).
- **REQ-EXECUTE-002:** The system shall log an error when the response status code is 404, 401, 403, or other errors.
- **REQ-EXECUTE-003:** The system shall handle retries correctly when the status code is 429 or 500.

---

3. Test Items

- Method under test: LpExecutor.execute(api_call, number_of_retries).
- Dependencies:
    - requests.Session.send.
    - LpApiCall.

    - Logging.
    - time.sleep (for retry handling).

---

4. Input Specifications

Each test case uses a mocked API call with varying status codes and retry configurations.

---

5. Output Specifications

- **Test Case 1 (TC-EXECUTOR-001)**: The system should return the mock response when the status code is 200.
- **Test Case 2 (TC-EXECUTOR-002)**: The system should log an error containing "404 Not Found".
- **Test Case 3 (TC-EXECUTOR-003)**: The system should log an error containing "Unauthorized — Bad Authentication".
- **Test Case 4 (TC-EXECUTOR-004)**: The system should log an error containing "Unauthorized — Forbidden request".
- **Test Case 5 (TC-EXECUTOR-005)**: The system should log a warning with a message "Retry after X seconds" for a 429 status code and retry after the specified delay.
- **Test Case 6 (TC-EXECUTOR-006)**: The system should log an error with a message "Internal server error" for a 500 status code and retry with a 15-second delay.
- **Test Case 7 (TC-EXECUTOR-007)**: The system should log an error containing "Unhandled status code: 418" for a 418 I'm a teapot status code.
- **Test Case 8 (TC-EXECUTOR-008)**: The system should log an error with "Error executing lp_api_manager call" for a request exception scenario.

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- requests library.
- logging module for capturing logs.

- time.sleep for controlling retry timing.

---

7. Procedures

TC-EXECUTOR-001: Execute API Call Success Test in LpExecutor

1. Mock requests.Session.send to return a response with a 200 status code.
2. Create an instance of LpApiCall with the following parameters:
    - type="POST", url="https://test.api.com", headers={"Authorization": "Bearer test_token"}, json={"key": "value"}
3. Create an instance of LpExecutor.
4. Call execute() method.
5. Verify the returned response matches the mock response.

TC-EXECUTOR-002: Execute API Call Failure Test for 404 Status Code

1. Mock requests.Session.send to return a response with a 404 status code.
2. Create an instance of LpApiCall with the following parameters:
    - type="GET", url="https://test.api.com/missing"
3. Create an instance of LpExecutor.
4. Call execute() method.
5. Verify that the log contains "404 Not Found".

TC-EXECUTOR-003: Execute API Call Failure Test for 401 Status Code

1. Mock requests.Session.send to return a response with a 401 status code.
2. Create an instance of LpApiCall with the following parameters:
    - type="GET", url="https://test.api.com/unauthorized"
3. Create an instance of LpExecutor.
4. Call execute() method.
5. Verify that the log contains "Unauthorized — Bad Authentication".

TC-EXECUTOR-004: Execute API Call Failure Test for 403 Status Code

1. Mock requests.Session.send to return a response with a 403 status code.
2. Create an instance of LpApiCall with the following parameters:

    - type="GET", url="https://test.api.com/forbidden"
3. Create an instance of LpExecutor.
4. Call execute() method.
5. Verify that the log contains "Unauthorized — Forbidden request".

TC-EXECUTOR-005: Execute API Call Retry Test for 429 Status Code

1. Mock requests.Session.send to return a response with a 429 status code and a Retry-After header.
2. Create an instance of LpApiCall with the following parameters:
    - type="GET", url="https://test.api.com/too-many-requests"
3. Create an instance of LpExecutor.
4. Call execute() method.
5. Verify that the log contains "Retry after X seconds".
6. Verify that time.sleep is called with the retry delay.

TC-EXECUTOR-006: Execute API Call Retry Test for 500 Status Code

1. Mock requests.Session.send to return a response with a 500 status code.
2. Create an instance of LpApiCall with the following parameters:
    - type="GET", url="https://test.api.com/server-error"
3. Create an instance of LpExecutor.
4. Call execute() method.
5. Verify that the log contains "Internal server error".
6. Verify that time.sleep is called with a 15-second delay.

TC-EXECUTOR-007: Execute API Call Test for Unhandled Status Code (418)

1. Mock requests.Session.send to return a response with a 418 status code.
2. Create an instance of LpApiCall with the following parameters:
    - type="GET", url="https://test.api.com/teapot"
3. Create an instance of LpExecutor.
4. Call execute() method.
5. Verify that the log contains "Unhandled status code: 418".

TC-EXECUTOR-008: Execute API Call Test for Request Exception

1. Mock requests.Session.send to raise a RequestException after the first response.
2. Create an instance of LpApiCall with the following parameters:
    - type="GET", url="https://test.api.com/exception"
3. Create an instance of LpExecutor.
4. Call execute() method.
5. Verify that the log contains "Error executing lp_api_manager call".

---

8. Special Procedural Requirements

- Proper mock setup for requests.Session.send to simulate various status codes and conditions.
- Ensure correct retry logic and sleep timings are applied.
- Log assertions must match the expected log messages.

---

9. Pass/Fail Criteria

- Each test case passes if the expected behavior (response or log message) is observed.
- Test cases fail if the actual behavior deviates from the expected response or logs.

System testing:

Test Case 1: Analyze Conversations with Threaded Execution in Analyzer Class

1. Purpose

To verify that the analyze_for_testing method in the Analyzer class correctly analyzes conversations using multiple threads, ensuring that the necessary algorithm adapters are invoked and the conversations are processed as expected.

2. Requirements Addressed

- **REQ-ANA-001**: The system shall process conversations using multiple threads to improve performance.
- **REQ-ANA-002**: The system shall correctly call the algorithm adapters (BasicAlgorithmAdapter, GSRAlgorithmAdapter) for each conversation.
- **REQ-ANA-003**: The system shall handle concurrent analysis of multiple conversations without errors or data loss.

3. Test Items

- Analyzer.analyze_for_testing().

Dependencies:

- ConfigUtil.get_config_attribute.
- ConversationCache.get_conversations_to_enrich.
- BasicAlgorithmAdapter.reference_to_analyze.
- GSRAlgorithmAdapter.reference_to_analyze.

4. Input Specifications

- **Mocked Configuration Values**:
    - "analyzer_interval": 15.
- **Mocked API Response**:
  Mocked data for get_conversations_to_enrich returns two conversations:

```
{
  "conv_id_1": [{"messageId": "ms::dialog:1", "text": "hello"}],
  "conv_id_2": [{"messageId": "ms::dialog:2", "text": "hi"}]
}
```

5. Output Specifications

- **Output**:
    - Dictionary containing analyzed data for each conversation.
    - Calls to reference_to_analyze methods for both BasicAlgorithmAdapter and GSRAlgorithmAdapter.
- **Expected Behavior**:
    - BasicAlgorithmAdapter.reference_to_analyze() should be called for each conversation.
    - GSRAlgorithmAdapter.reference_to_analyze() should be called for each conversation.
    - The final result should reflect the processed data from both adapters.

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- threading for simulating concurrent execution.
- MagicMock to mock external dependencies.

7. Procedures

Test Case 1: test_analyze_with_threads

- **Patch**:
    - Mock the ConfigUtil to return the necessary configuration values.

    - Mock ConversationCache.get_conversations_to_enrich to return conversation data.
    - Mock BasicAlgorithmAdapter.reference_to_analyze and GSRAlgorithmAdapter.reference_to_analyze to simulate algorithm results.
    - Mock time.sleep to avoid real delays during testing.
- **Simulate Response**:
  Provide mock responses for both adapters, returning simulated analyzed data for each conversation.
- **Call**:
  Call Analyzer.analyze_for_testing() with appropriate arguments.

- **Assert**:
    - Verify that get_conversations_to_enrich was called.
    - Verify that reference_to_analyze for both adapters was called.
    - Ensure that the analysis method was executed multiple times (5 in this case, based on the threading).

---

8. Special Procedural Requirements

- Assumes that mock responses from the algorithms are correctly formatted and return expected results.
- Assumes that threading does not cause race conditions or errors in the analysis process.

9. Intercase Dependencies

- The test depends on mock configurations and external class dependencies being correctly set up.
- The Analyzer class must be properly implemented to handle multiple threads and external method calls as described.

Unit testing:

Test Case 1: ConversationCache Class and Verify Default States

1. Purpose

To verify that the ConversationCache class is initialized correctly with default values, ensuring that the required components (e.g., RWLock and Thread) are instantiated and that the cache's internal dictionaries are set to their expected initial states.

2. Requirements Addressed

- **REQ-CC-001**: The system shall initialize a ConversationCache instance with empty dictionaries for closed_conversations, messages, and open_conversations.
- **REQ-CC-002**: The system shall use a RWLockFair instance for managing concurrent read and write operations on the cache.
- **REQ-CC-003**: The system shall spawn threads for cleanup_old_data and log_data methods during initialization.

3. Test Items

- ConversationCache.__init__().

Dependencies:

- rwlock.RWLockFair.
- threading.Thread.
- ConversationCache.cleanup_old_data.
- ConversationCache.log_data.

4. Input Specifications

- **Mocked Components**:
    - RWLockFair (to mock the locking mechanism).
    - threading.Thread (to mock the thread creation for background tasks).

5. Output Specifications

- **Output**:
    - A ConversationCache instance with default internal states:
        - closed_conversations: Empty dictionary ({}).
        - messages: Empty dictionary ({}).
        - open_conversations: Empty dictionary ({}).
- **Expected Behavior**:
    - The RWLockFair constructor should be called once to initialize the lock.
    - The threading.Thread should spawn two threads for cleanup_old_data and log_data methods.

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- threading for simulating thread creation.
- MagicMock to mock external components (RWLockFair, Thread).

7. Procedures

Test Case 1: test_conversation_cache_init

- **Patch**:
    - Mock RWLockFair to ensure the lock is correctly initialized.
    - Mock threading.Thread to simulate thread creation and verify that the expected methods (cleanup_old_data and log_data) are targeted.
- **Simulate Response**:
    - No need for actual responses from the threads or lock; we only need to verify that the mock methods were called.
- **Call**:

    - Initialize ConversationCache() to invoke the constructor and setup the cache.
- **Assert**:
    - Verify that RWLockFair was called once to initialize the lock.
    - Ensure that threading.Thread was called with target=cache.cleanup_old_data and target=cache.log_data.
    - Confirm that the cache's internal states are initialized to empty dictionaries.

---

8. Special Procedural Requirements

- Assumes that ConversationCache's cleanup_old_data and log_data methods are valid and correctly implemented.
- Assumes that the threading functionality is correctly mocked and does not execute any actual background tasks during the test.

9. Intercase Dependencies

- No intercase dependencies, but the test assumes that ConversationCache and its associated methods (cleanup_old_data, log_data) are implemented as expected.

Test Case 2: Start Cleanup Thread for ConversationCache

1. Purpose

To verify that the _start_cleanup_thread() method in the ConversationCache class correctly initializes and starts a new thread to run the cleanup_old_data method.

---

2. Requirements Addressed

- **REQ-CC-004**: The system shall support launching a background thread for periodic cleanup of outdated conversation data.
- **REQ-CC-005**: The background thread shall target the cleanup_old_data method of the ConversationCache instance.
- **REQ-CC-006**: The cleanup thread shall be started immediately upon initialization.

---

3. Test Items

- ConversationCache._start_cleanup_thread().

Dependencies:

- threading.Thread.
- ConversationCache.cleanup_old_data.

---

4. Input Specifications

- **Mocked Components**:
    - threading.Thread: to intercept thread instantiation and verify the correct target method is used.

---

5. Output Specifications

- **Output**:
    - The threading.Thread is instantiated with target=ConversationCache.cleanup_old_data.
    - The start() method on the thread is called exactly once.

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- Access to ConversationCache class definition.
- Mocking capabilities for threading (via patch.)

---

7. Procedures

Test Case 1: test_start_cleanup_thread

- **Patch**:
    - Patch threading.Thread to intercept thread creation.
- **Simulate Response**:
    - mock_thread.return_value is treated as the thread instance, with a mock start() method.
- **Call**:
    - Create a ConversationCache instance.
    - Call the _start_cleanup_thread() method directly.
- **Assert**:
    - Verify threading.Thread is called with cache.cleanup_old_data.
    - Confirm that start() is called once on the created thread instance.

---

8. Special Procedural Requirements

- Assumes that cleanup_old_data is a valid method of ConversationCache.
- Assumes that thread execution is not required—only validation of instantiation and starting is necessary.

---

9. Intercase Dependencies

- None explicitly. This test is independent of other ConversationCache behaviors and only validates the thread launch mechanism.

Test Case 3: Start Log Thread for ConversationCache

1. Purpose

To verify that the _start_log_thread() method in the ConversationCache class correctly initializes and starts a new thread to execute the log_data method.

---

2. Requirements Addressed

- **REQ-CC-007**: The system shall support launching a background thread for logging conversation cache data periodically.
- **REQ-CC-008**: The background thread shall target the log_data method of the ConversationCache instance.
- **REQ-CC-009**: The logging thread shall be started upon initialization when _start_log_thread() is invoked.

---

3. Test Items

- ConversationCache._start_log_thread()

Dependencies:

- threading.Thread
- ConversationCache.log_data

---

4. Input Specifications

- **Mocked Component**:
    - threading.Thread: to intercept thread creation and monitor usage of the log_data method as the thread’s target.

---

5. Output Specifications

Output	Expected Value
Thread instantiation target	ConversationCache.log_data .
Thread start method	Called exactly once.

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- ConversationCache implementation.
- Ability to patch threading behavior.

---

7. Procedures

Test Case: test_start_log_thread

- **Patch**:
    - Patch threading.Thread to intercept thread initialization.
- **Simulate Response**:
    - Use mock_thread.return_value.start to mock the thread's start method.
- **Call**:
    - Instantiate ConversationCache.
    - Call cache._start_log_thread() directly.
- **Assert**:
    - Verify threading.Thread was called with target=cache.log_data.
    - Confirm that the mock thread's start() method was called exactly once.

---

8. Special Procedural Requirements

- Assumes log_data exists as a method within ConversationCache.
- This test does not require actual execution of the thread, only confirmation that it was correctly initialized and started.

---

9. Intercase Dependencies

- This test is independent but complements test_start_cleanup_thread, both verifying internal threading logic for ConversationCache.

Test Case 4: Update Open Conversations in ConversationCache

1. Purpose

To validate that the update_conversations() method in the ConversationCache class correctly updates internal dictionaries for open conversations and messages using the provided inputs.

---

2. Requirements Addressed

- **REQ-CC-010**: The cache shall support updating open conversations with a given timestamp and message set.
- **REQ-CC-011**: The update shall be thread-safe using internal locking mechanisms.
- **REQ-CC-012**: The cache shall maintain a consistent mapping of conversation data and corresponding messages.

---

3. Test Items

- ConversationCache.update_conversations().

Dependencies:

- ConversationCache._lock.
- ConversationCache.open_conversations.
- ConversationCache.messages.

- ConfigUtil.
- ConversationCache._start_log_thread.
- ConversationCache._start_cleanup_thread.

---

4. Input Specifications

Input	Value
mock_current_time	123456789.
mock_conversation	{'conv1': MagicMock()}.
mock_messages	{'conv1': {MagicMock()}}.

Mocked Components:

- ConfigUtil.
- _start_log_thread.
- _start_cleanup_thread.
- _lock (to simulate thread-safe block).

---

5. Output Specifications

Output Property	Expected Value
cache.open_conversations	Matches mock_conversation.
cache.messages	Matches mock_messages.

---

6. Environmental Needs

- Python 3.x.
- unittest and unittest.mock.
- ConversationCache class definition.
- Ability to patch internal and external class methods.

---

7. Procedures

Test Case: test_update_open_conversations

- **Patch**:
    - Patch ConfigUtil to suppress configuration loading.
    - Patch _start_log_thread and _start_cleanup_thread to prevent thread execution.
    - Patch cache._lock with a MagicMock to simulate thread-safe block without real locking.
- **Setup**:
    - Create a mock current time, conversations, and messages.
- **Call**:
    - Instantiate ConversationCache and invoke update_conversations(...).
- **Assert**:
    - Ensure cache.open_conversations matches mock_conversation.
    - Ensure cache.messages matches mock_messages.

---

8. Special Procedural Requirements

- Thread-related methods are mocked to avoid spawning actual threads during testing.
- Locking is mocked to test the behavior without requiring concurrency.

---

9. Intercase Dependencies

- May follow initialization tests such as test_conversation_cache_init or complement them to test post-initialization logic.
- Works independently from cleanup and logging functionalities.

Test Case 5: Move Conversations to Closed in ConversationCache

1. Purpose

To verify that _move_conv_to_close() in the ConversationCache correctly transitions specified conversations from open to closed, updates message data, and applies appropriate field updates.

---

2. Requirements Addressed

- **REQ-CC-020**: The system shall move specified conversation IDs from open to closed state in the cache.
- **REQ-CC-021**: The system shall fetch full conversation and message data for the given IDs.
- **REQ-CC-022**: The system shall update closed conversation fields such as need_analyze and lastUpdateTime.
- **REQ-CC-023**: The open conversation entries shall be removed upon successful move.

---

3. Test Items

- ConversationCache._move_conv_to_close().

Dependencies:

- LpUtils.get_conversations_by_conv_id().
- LpUtils.extract_conversations().
- LpUtils.extract_messages().
- closed_conversations.update().
- closed_conversations.update_field().
- open_conversations.
- Messages.

---

4. Input Specifications

Input Name	Value
mock_now	123456789.
mock_conversations_ids	['conv1', 'conv2'].
open_conversations	{'conv1': MagicMock(), 'conv2': MagicMock()}.
closed_conversations	Initially empty.
messages	Initially empty.
mock_conversations_records	{'conv1': MagicMock(), 'conv2': MagicMock()}.
mock_messages_records	{'conv1': [MagicMock()], 'conv2': [MagicMock()]}.

Mocks:

- LpUtils.get_conversations_by_conv_id() → mock_response.
- LpUtils.extract_conversations() → mock_conversations_records.
- LpUtils.extract_messages() → mock_messages_records.

---

5. Output Specifications

Output	Expected Result
closed_conversations	Contains updated entries for conv1, conv2.
open_conversations	conv1 and conv2 removed.
messages	Updated with messages for conv1 and conv2.
update_field("need_analyze", True)	Called on each closed conversation.
update_field("lastUpdateTime", mock_now)	Called on each closed conversation.

---

6. Environmental Needs

- Python 3.x.
- unittest.mock.
- ConversationCache and LpUtils definitions.
- Proper internal structure of closed_conversations and messages dictionaries.

---

7. Procedures

Test Case: test_move_conv_to_close

- **Patch**:
    - Patch LpUtils to mock its internal methods:
        - get_conversations_by_conv_id().
        - extract_conversations().
        - extract_messages().
- **Setup**:
    - Create ConversationCache object.
    - Initialize open_conversations with mocked conversations.
    - Create mocked values for conversations, messages, and current timestamp.

- **Call**:
    - Call cache._move_conv_to_close(mock_now, mock_conversations_ids).
- **Assert**:
    - Verify closed_conversations contains both conv1 and conv2.
    - Verify open_conversations no longer contains the moved IDs.
    - Check that messages were correctly updated for both conversations.
    - Ensure update_field("need_analyze", True) and update_field("lastUpdateTime", mock_now) were called for each closed conversation.

---

8. Special Procedural Requirements

- Assumes that conversation and message objects support update() and update_field() methods.
- Depends on correct mocking of LpUtils API.

---

9. Intercase Dependencies

- May follow update or fetch conversation tests.
- Complements open-to-closed lifecycle tests and message tracking.

Test Case 6: Retrieve All Conversations (Open + Closed) from ConversationCache

1. Purpose

To ensure that the get_all_conversations() method in ConversationCache returns a complete and accurate set of all currently tracked conversations, combining both open and closed conversations under thread-safe access.

---

2. Requirements Addressed

- **REQ-CC-030**: The system shall provide access to both open and closed conversation records.
- **REQ-CC-031**: The retrieval of conversation records shall use a read-lock to ensure thread-safe access.

---

3. Test Items

- ConversationCache.get_all_conversations().

Dependencies:

- ConversationCache.open_conversations.
- ConversationCache.closed_conversations.
- RWLockFair.gen_rlock() for read-lock synchronization.

---

4. Input Specifications

Input Name	Value
open_conversations	{'conv1': MagicMock(), 'conv2': MagicMock()}.
closed_conversations	{'conv3': MagicMock(), 'conv4': MagicMock()}.
RWLockFair	Mocked to simulate read-lock behavior.

Mocks:

- mock_rwlock: Patches the RWLockFair lock object used within the cache.
- gen_rlock: Simulates thread-safe lock entry and exit context management.

---

5. Output Specifications

Output	Expected Result
all_conversations	A set containing all values from open_conversations and closed_conversations.
Lock behavior	gen_rlock().__enter__() and __exit__() are called properly.

---

6. Environmental Needs

- Python 3.x.
- unittest.mock.
- Functional implementation of ConversationCache with internal dictionaries.
- Read-lock (RWLockFair) implementation and patching.

---

7. Procedures

Test Case: test_get_all_conversations

- **Patch**:
    - Patch RWLockFair to simulate locking behavior during access to shared data.
- **Setup**:
    - Instantiate ConversationCache.
    - Inject mock open_conversations and closed_conversations into the cache.
    - Set up read-lock mock to correctly simulate context manager behavior (__enter__, __exit__).
- **Call**:
    - Invoke cache.get_all_conversations().
- **Assert**:
    - Verify that the return value is a set containing all mocked conversation objects.
    - Ensure proper read-lock usage through calls to gen_rlock().

---

8. Special Procedural Requirements

- Assumes conversation entries can be compared and stored in a set.
- Assumes correct internal setup of thread locking via RWLockFair.

---

9. Intercase Dependencies

- Independent test but complements open/closed access tests and thread-safe method tests.

Test Case 7: Retrieve Closed Conversations from ConversationCache

1. Purpose

To verify that the get_closed_conversations() method in ConversationCache correctly returns the list of closed conversations, ensuring that the read-lock mechanism is properly employed for thread-safe access.

---

2. Requirements Addressed

- **REQ-CC-032**: The system shall provide access to closed conversation records.
- **REQ-CC-033**: The retrieval of closed conversations shall utilize a read-lock for thread safety.

---

3. Test Items

- ConversationCache.get_closed_conversations().

Dependencies:

- ConversationCache.closed_conversations.
- RWLockFair.gen_rlock() for read-lock synchronization.

4.Input Specifications

Input Name	Value
closed_conversations	{‘conv1’: MagicMock(), 'conv2': MagicMock()}.
RWLockFair	Mocked to simulate read-lock behavior.

Mocks:

- mock_rwlock: Patches the RWLockFair lock object used within the cache.
- gen_rlock: Simulates thread-safe lock entry and exit context management.

---

5. Output Specifications

Output	Expected Result
result	The mock_closed_conversations dictionary, as returned by get_closed_conversations() method.
Lock behavior	gen_rlock().__enter__() and __exit__() should be invoked for thread-safe access.

---

6. Environmental Needs

- Python 3.x.
- unittest.mock.
- Functional implementation of ConversationCache with a closed_conversations dictionary.
- Read-lock (RWLockFair) implementation and patching.

---

7. Procedures

Test Case: test_get_closed_conversations

- **Patch**:
    - Patch RWLockFair to simulate read-lock behavior during access to the closed conversations data.
- **Setup**:
    - Instantiate ConversationCache.
    - Inject mock closed_conversations into the cache.
    - Set up the mock_rwlock and gen_rlock() mock to simulate proper context management.
- **Call**:
    - Invoke cache.get_closed_conversations().
- **Assert**:
    - Verify that the result matches the expected mock closed_conversations.
    - Ensure that the read-lock context (gen_rlock) is used correctly.

---

8. Special Procedural Requirements

- Assumes that conversation records can be mocked with MagicMock.
- Assumes that the closed_conversations dictionary contains valid conversation records that can be returned directly.

---

9. Intercase Dependencies

- Independent test, focuses solely on thread-safe access and retrieval of closed conversations.

Test Case 8: Retrieve Conversations to Enrich from ConversationCache

1. Purpose

To verify that the get_conversations_to_enrich() method in ConversationCache correctly identifies and returns conversations that are open or marked as needing enrichment (need_analyze=True), while excluding those that are closed or don't require enrichment.

---

2. Requirements Addressed

- **REQ-CC-034**: The system shall identify and return conversations that need enrichment (those that are open or have the need_analyze flag set to True).
- **REQ-CC-035**: The system shall not return conversations that do not require enrichment (i.e., need_analyze=False).

---

3. Test Items

- ConversationCache.get_conversations_to_enrich().

Dependencies:

- ConversationCache.open_conversations.
- ConversationCache.closed_conversations
- ConversationCache.messages.

---

4. Input Specifications

Input Name	Value
open_conversations	{'conv1': MagicMock(), 'conv2': MagicMock()}.
closed_conversations	{'conv3': MagicMock(need_analyze=True), 'conv4': MagicMock(need_analyze=False)}.
messages	{'conv1': [MagicMock()], 'conv2': [MagicMock()], 'conv3': [MagicMock()], 'conv4': [MagicMock()]}.

Mocks:

- mock_gen_rlock: Simulates the context manager for the read-lock (gen_rlock) to ensure thread-safety.
- MagicMock(): Mocks the conversation objects.

---

5. Output Specifications

Output	Expected Result
result	A dictionary containing conversations to enrich (open or marked with need_analyze=True) and their associated messages.
Excluded Conversations	Conversations marked with need_analyze=False should not be included in the result.

---

6. Environmental Needs

- Python 3.x.
- unittest.mock.
- Functional implementation of ConversationCache with open_conversations, closed_conversations, and messages attributes.
- Read-lock (RWLockFair) implementation and patching.

---

7. Procedures

Test Case: test_get_conversations_to_enrich

- **Patch**:
  - Patch ConfigUtil to simulate external configuration (if needed).
  - Mock gen_rlock to simulate thread-safe lock access.
- **Setup**:
  - Instantiate ConversationCache.
  - Set up mock data for open_conversations, closed_conversations, and messages.
  - Mock need_analyze flag for closed conversations.
- **Call**:
  - Invoke cache.get_conversations_to_enrich().
- **Assert**:
  - Verify that the result includes the correct conversations (conv1, conv2, conv3).
  - Ensure that conv4 (which does not need enrichment) is excluded from the result.
  - Ensure that the messages associated with each returned conversation are correct.

---

8. Special Procedural Requirements

- Assumes that mock objects properly simulate the required behaviors for open and closed conversations, including the need_analyze flag.
- Assumes proper handling of thread-safe locks when accessing conversation data.

---

9. Intercase Dependencies

- Independent test, ensures correct filtering of conversations for enrichment.

Test Case 9: Retrieve Open Conversations from ConversationCache

1. Purpose

To verify that the get_open_conversations() method in ConversationCache correctly returns all currently open conversations without modification and in expected format.

---

2. Requirements Addressed

- **REQ-CC-032**: The system shall provide a mechanism to access all open conversations currently stored in memory.
- **REQ-CC-033**: Returned conversations shall maintain object integrity and match the internal cache content.

---

3. Test Items

- ConversationCache.get_open_conversations().

Dependencies:

- ConversationCache.open_conversations.

---

5. **Input Specifications**

Input Name	Value
open_conversations	{‘conv1’: MagicMock(), 'conv2': MagicMock()}.

Mocks:

- MagicMock(): Used to simulate open conversation objects.
- ConfigUtil: Patched as required by the constructor (even if unused directly in the test).

---

5. Output Specifications

Output	Expected Result
result	Should equal the dictionary of open_conversations.
Result Keys	'conv1' and 'conv2' must be present in the result.
Result Values	Should be instances of MagicMock to simulate conversation objects.

---

6. Environmental Needs

- Python 3.x.
- unittest.mock.
- Functional implementation of ConversationCache class.
- Patched ConfigUtil to satisfy class dependencies.

---

7. Procedures

Test Case: test_get_open_conversations

- **Patch**:
    - Patch ConfigUtil for constructor compatibility.
- **Setup**:
    - Instantiate ConversationCache.
    - Assign mocked open conversations to cache.open_conversations.

- **Call**:
    - Invoke cache.get_open_conversations().
- **Assert**:
    - Check that the result matches the mock dictionary assigned.
    - Verify that both keys (conv1, conv2) exist in the result.
    - Confirm that each value in the result is an instance of MagicMock.

---

8. Special Procedural Requirements

- Assumes no additional processing or transformation occurs when fetching open conversations.
- Focuses on data integrity and correct return behavior.

---

9. Intercase Dependencies

- No dependency on other tests. Can be executed independently.

Test Case 10: : Enriching Conversations in ConversationCache

1. Purpose

To verify the correct behavior of ConversationCache.enrich_conversation() when enriching both open and closed conversations, handling nonexistent conversations, and managing invalid JSON input gracefully.

---

2. Requirements Addressed

- **REQ-CC-041**: The system shall enrich open or closed conversations with data provided in JSON format.
- **REQ-CC-042**: Enrichment shall update conversation fields including last_effected_message_id.
- **REQ-CC-043**: Closed conversations shall set need_analyze to False after enrichment.
- **REQ-CC-044**: Invalid JSON inputs shall be handled without crashing and must be logged as errors.

---

3. Test Items

- ConversationCache.enrich_conversation(conv_id, json_str, last_message_id).

Dependencies:

- _lock.gen_wlock() for thread-safe updates.
- update_field() on conversation records.
- Logging of JSON errors.

---

4. Input Specifications

Input Name	Value
conv_id_open	'open_conv' (in open_conversations).
conv_id_closed	'closed_conv' (in closed_conversations).
conv_id_nonexistent	'nonexistent_conv' (not present in any dict).
json_str	'{"key1": "value1", "key2": "value2"}'.
last_message_id	'msg123'.
invalid_json_str	'{"key1": "value1", "key2":' (truncated JSON).

Mocks:

- MagicMock() for open and closed conversations.
- logging.error for capturing JSON parsing issues.

---

5. Output Specifications

Scenario	Expected Behavior
Enriching open_conv	update_field() is called for all JSON keys + last_effected_message_id.
Enriching closed_conv	Same as above + need_analyze set to False.
Enriching nonexistent_conv	No effect, function silently skips updating.
Invalid JSON	Error is logged using logging.error() with specific parsing error message.

---

6. Environmental Needs

- Python 3.x.
- unittest.mock.
- Logging module.
- ConversationCache class with enrich_conversation() method.
- ConfigUtil patched for constructor compatibility.

---

7. Procedures

Test Case: test_enrich_conversation().

- **Setup**:
  - Patch ConfigUtil for instantiating ConversationCache.
  - Patch _lock.gen_wlock() to simulate thread-safe writing.
  - Set up mock open_conversations and closed_conversations.
- **Test Steps**:
  - Enrich an open conversation and assert that fields are updated.
  - Enrich a closed conversation and assert that fields are updated + need_analyze is set to False.
  - Try to enrich a nonexistent conversation and confirm no errors or side effects.
  - Pass invalid JSON and confirm that an appropriate logging error is triggered.

---

8. Special Procedural Requirements

- Must run in an environment with logging enabled.
- Ensure patching does not interfere with concurrent thread logic.

---

9. Intercase Dependencies

- Can be run independently. Does not require other test cases to run first.

Test Case 11: Periodic Logging of Cache Data in ConversationCache

1. Purpose

To validate that the ConversationCache.log_data() method periodically logs cache. summary and conversation info as expected, using configuration-defined intervals.

---

2. Requirements Addressed

- **REQ-CC-051**: The system shall periodically log cache data including summary and individual conversation details.
- **REQ-CC-052**: The logging frequency shall be determined by the logCacheInterval configuration parameter.
- **REQ-CC-053**: Logging shall be thread-safe and handle interruption gracefully.

---

3. Test Items

- ConversationCache.log_data().

Dependencies:

- ConfigUtil.get_config_attribute().
- RWLockFair.gen_rlock().
- Methods: log_summary_line(), log_conversation_info().

- time.sleep() for interval control.

---

4. Input Specifications

Input Name	Value
logCacheInterval	1 (second).
sleep.side_effect	KeyboardInterrupt after 2 iterations.

Mocks:

- ConfigUtil returns config with logCacheInterval.
- RWLockFair and .gen_rlock() to simulate thread-safe locking.
- log_summary_line and log_conversation_info to simulate logging functions.
- logging.debug and logging.error patched to suppress actual logs.

---

5. Output Specifications

Scenario	Expected Behavior
Logging starts and runs 2 cycles	Both log_summary_line and log_conversation_info called twice each.
Logging path format	Uses logs/cache_log_<MM_DD_YY>.xlsx for log output.
Logging ends via interrupt	KeyboardInterrupt is caught and test completes gracefully.

---

6. Environmental Needs

- Python 3.x.
- unittest.mock.
- ConversationCache class.
- Logging infrastructure (mocked).
- File system path for logging (virtual).

---

7. Procedures

Test Case: test_log_data().

- **Setup**:
    - Patch ConfigUtil to return logCacheInterval of 1 second.
    - Patch RWLockFair to simulate read locks.
    - Patch sleep() to simulate timing + inject KeyboardInterrupt after 2 loops.
    - Patch logging methods to avoid console clutter.
    - Patch log_summary_line() and log_conversation_info() methods.
- **Test Steps**:
    - Initialize ConversationCache.
    - Start log_data() method within a try-except block.
    - Verify that both logging functions are called exactly twice.
    - Confirm correct file path used for both logging calls.

---

8. Special Procedural Requirements

- Must ensure KeyboardInterrupt simulates loop interruption cleanly.
- Must avoid writing any actual files to disk.

---

9. Intercase Dependencies

- Independent test. No dependencies on other test cases.

Test Case 12: : Backup Handling for DR Cache File (DR_cache_file)

1. Purpose

To verify the behavior of the ConversationCache.DR_cache_file() method which is responsible for managing backup operations when accessing or replacing an Excel file used in disaster recovery (DR) logging.

---

2. Requirements Addressed

- **REQ-CC-070**: The system shall create a new Excel workbook when DR logging is initiated.
- **REQ-CC-071**: If an existing cache file is present, it shall be renamed with a timestamp to preserve the old data.
- **REQ-CC-072**: Errors during file rename shall be gracefully handled without preventing the creation of a new workbook.

---

3. Test Items

- ConversationCache.DR_cache_file(file_path: str) -> Workbook.

---

4. Input Specifications

Input	Description
file_path	A string path to the Excel file (test_file.xlsx).
datetime.now()	Used to generate a unique timestamp for the backup

Mocks:

- os.rename: Mocked to simulate rename success, file-not-found, and generic exception.
- openpyxl.workbook.Workbook: Mocked to return a fake Workbook.
- ConfigUtil: Patched but unused (kept for consistency with surrounding codebase).

---

5. Output Specifications

Scenario	Expected Behavior
File exists and is renamed	os.rename() is called, and a new Workbook is returned.
File not found during rename	FileNotFoundError is caught, new Workbook still returned.
Generic rename error (e.g., permission denied)	Error is caught, new Workbook returned, system continues.

---

6. Environmental Needs

- Python 3.x.
- unittest.mock.
- openpyxl.workbook.Workbook.
- Filesystem interactions (mocked).

---

7. Procedures

Test Case 1: test_DR_cache_file

- Simulate successful rename of existing cache file.

- Assert os.rename is called exactly once.
- Assert returned object is a Workbook instance.

Test Case 2: test_DR_cache_file_file_not_found

- Simulate FileNotFoundError from os.rename.
- Assert graceful fallback and new workbook creation.

Test Case 3: test_DR_cache_file_rename_error

- Simulate generic Exception during os.rename.
- Assert error is caught and fallback proceeds with new workbook.

---

8. Special Procedural Requirements

- Time-sensitive test logic (based on datetime.now) should be mocked if deterministic timestamps are necessary for file path assertions.

---

9. Intercase Dependencies

- All test cases are isolated and can be run independently.

Test Case 13: Update Field Logic in ConversationHistoryRecord

1. Purpose

To verify the correctness and constraints of the update_field method in the ConversationHistoryRecord class, which is used to dynamically update internal fields while preserving data integrity (e.g., ignoring undefined/null values, handling GSR/max_GSR logic).

---

2. Requirements Addressed

- **REQ-HIST-001**: Must allow valid field updates and handle special logic for related fields (GSR and max_GSR).
- **REQ-HIST-002**: Must ignore updates where the value is "undefined" or None for non-nullable fields.

---

3. Test Items

- ConversationHistoryRecord.update_field(field_name: str, value: Any).

---

4. Input Specifications

Each test initializes a ConversationHistoryRecord with standard values:

Field	Example Value
start_time	"2024-04-29 16:10:18.818+0000".
start_time_L	1714407018818.
conversation_id	"47b21186-4501-4363-8752-27bc8bd212b7" .
latest_agent_id	"3998511138".
full_dialog_status	"CLOSE".
latest_agent_full_name	"טל הלן פרנץ".
status	"CLOSE".

---

5. Output Specifications

Test Case Name	Description	Expected Outcome
test_update_valid_field_success	Tests valid update of GSR and max_GSR	Field values updated properly; max_GSR reflects the latest assigned value.
test_ignore_undefined_field_value	Tests that "undefined" string value is ignored for status	status remains "CLOSE".
test_undefined_field_value	Tests that None is ignored for latestAgentId.	latestAgentId remains as initially set.

---

6. Environmental Needs

- Python 3.x
- Unit test framework (e.g., unittest).
- A defined ConversationHistoryRecord class with update_field method.

---

7. Procedures

Test Case 1: test_update_valid_field_success

- Create record with known state.
- Update field GSR to 5 → verify both GSR and max_GSR reflect the change.
- Update field max_GSR directly to 10 → verify only max_GSR is updated.

Test Case 2: test_ignore_undefined_field_value

- Create record.
- Attempt to update status to "undefined".
- Assert that status value does not change.

Test Case 3: test_undefined_field_value

- Create record.
- Attempt to update latestAgentId to None.
- Assert that latestAgentId remains unchanged.

---

8. Special Procedural Requirements

None.

---

9. Intercase Dependencies

None – each test is independent and can be executed in isolation.

Test Case 14: Update Maximum GSR in ConversationHistoryRecord

1. Purpose

To validate the behavior of the update_maxGSR() method, ensuring it correctly updates the max_GSR field only when the provided value is higher than the existing one, or when it is uninitialized.

---

2. Requirements Addressed

- **REQ-HIST-003**: The max_GSR field should be updated only if the new GSR value is greater than the existing value.
- **REQ-HIST-004**: The field should remain unchanged when the new value is lower or equal.

---

3. Test Items

- ConversationHistoryRecord.update_maxGSR(new_value: int).

---

4. Input Specifications

Field	Example Value
start_time	"2024-04-29 16:10:18.818+0000"
start_time_L	1714407018818
conversation_id	"47b21186-4501-4363-8752-27bc8bd212b7"
latest_agent_id	"3998511138"
full_dialog_status	"CLOSE"
latest_agent_full_name	"טל הלן פרנץ"
status	"CLOSE"

---

5. Output Specifications

Test Case Name	Description	Expected Outcome
test_initial_update	Set initial max_GSR from 0 to 5	max_GSR == 5.
test_update_with_higher_value	Update max_GSR from 10 to 15	max_GSR == 15.
test_update_with_lower_value	Attempt to update max_GSR from 20 to 10 (lower value)	max_GSR == 20 (unchanged).
test_update_with_equal_value	Attempt to update max_GSR with a value equal to the current (25 == 25)	max_GSR == 25 (unchanged).

---

6. Environmental Needs

- Python 3.x.
- Unit test framework (e.g., unittest)
- Class ConversationHistoryRecord must have max_GSR and method update_maxGSR().

---

7. Procedures

Test Case 1: test_initial_update

- Create new record with default max_GSR..
- Call update_maxGSR(5).
- Assert max_GSR == 5.

Test Case 2: test_update_with_higher_value.

- Set max_GSR = 10.
- Call update_maxGSR(15).
- Assert max_GSR == 15.

Test Case 3: test_update_with_lower_value.

- Set max_GSR = 20.
- Call update_maxGSR(10).
- Assert max_GSR == 20 (unchanged).

Test Case 4: test_update_with_equal_value.

- Set max_GSR = 25.
- Call update_maxGSR(25).
- Assert max_GSR == 25 (unchanged).

---

8. Special Procedural Requirements

None.

---

9. Intercase Dependencies

None – each test case is standalone.

Test Case 15: Partial Update of ConversationHistoryRecord Fields

1. Purpose

To verify that the update() method in ConversationHistoryRecord:

- Correctly updates fields from another record.
- Skips fields that are None or "undefined".
- Updates existing values when valid.
- Preserves old values when new ones are invalid or missing.

---

2. Requirements Addressed

- **REQ-HIST-005**: Records can be updated with another record's non-null, defined values.
- **REQ-HIST-006**: Fields with None or "undefined" must not override existing values.

---

3. Test Items

- ConversationHistoryRecord.update(other_record: ConversationHistoryRecord).

---

4. Input Specifications

Each test uses two ConversationHistoryRecord objects with overlapping fields and different field values. Some fields will intentionally be None or "undefined".

---

5. Output Specifications

Test Case Name	Description	Expected Outcome
test_update_GSR	Valid update: GSR is overwritten with a lower value	record1.GSR == 5.0.
test_ignore_none_and_undefined_values	Ignore update fields that are None or "undefined"	duration and source remain as in record1.
test_partial_update	Mixed update: one valid, one invalid field	duration is updated, source remains unchanged.

---

6. Environmental Needs

- Python 3.x.
- Unit test framework (unittest).
- ConversationHistoryRecord class with proper update() logic.

---

7. Procedures

Test Case 1: test_update_GSR

1. Create record1 with GSR = 10.0.
2. Create record2 with GSR = 5.0.
3. Call record1.update(record2).
4. Assert record1.GSR == 5.0.

Test Case 2: test_ignore_none_and_undefined_values

1. Set record1.duration = 693443 and source = "SHARK".
2. Set record2.duration = None and source = "undefined".
3. Call record1.update(record2).
4. Assert no change in record1.duration and record1.source.

Test Case 3: test_partial_update

1. record1.duration = 693443, record1.source = "SHARK".
2. record2.duration = 500000, record2.source = None.
3. Call record1.update(record2).
4. Assert record1.duration == 500000, record1.source == "SHARK".

---

8. Special Procedural Requirements

None.

---

9. Intercase Dependencies

None – all test cases are isolated and do not depend on each other.

Front-end testing:

Test Case 1: Inserting conversation with high level of suicide risk (high GSR)

Objective: To verify that a conversation flagged with a high level of suicide risk is:

1. Displayed correctly in the Active Conversations page.
2. Triggers an alert notification on the front-end interface.

Preconditions:

- The back-end system is running , accessible and configured to analyze GSR (suicide risk levels).
- Frontend is actively polling or responding to open call list updates.

Steps:

1.**Generate a Conversation**
Simulate conversation with high suicide risk (high GSR score).

2.**Insert into Open Calls List**
Add this conversation to the back-end system's open calls list.

3.**Trigger API to Send Open Calls**
Perform an API call to fetch the list of open conversations.

4.**Verify Front-End Display**
Confirm that the new high-risk conversation appears in the Active Conversations list on the front end.

5.**Check for Alert**
Confirm that a visible alert is shown, indicating a high-risk case has been detected.

Expected Result:

- The new conversation appears in the front-end Active Conversations list.
- An alert is shown to the user clearly indicating a high-risk conversation.

Test Case 2: Inserting conversations

Objective: To verify that all conversations present in the backend system are correctly displayed on the Active Conversations page of the front end.

Preconditions:

- The back-end system is running , accessible and configured to analyze GSR (suicide risk levels).
- Frontend is actively polling or responding to open call list updates.

Steps:

1.**Generate Conversations**

Simulate multiple conversations, each with varying GSR scores.

2.**Insert into Open Calls List**

Add all generated conversations to the back-end system's open calls list.

3.**Trigger API to Send Open Calls**

Perform an API call to fetch the list of open conversations.

4. **Verify Front-End Display**

Confirm that each inserted conversation appears in the Active Conversations list shown on the frontend interface.

Expected Result: All conversations inserted into the backend open calls list appear in the frontend Active Conversations page.

Test Case 3: Inserting conversations to the conversations' history

Objective: To verify that all conversations present in the backend closed calls list are correctly displayed on the History Conversations page of the front end, according to the correct time frame provided by the front-end system.

Preconditions:

- The back-end system is running , accessible and configured to analyze GSR (suicide risk levels).
- Frontend is actively polling or triggering API calls to retrieve closed conversations (history).
- Time frame filter or date range is defined and passed by the front-end to the backend.

Steps:

1. **Generate Conversations**
Simulate several conversations, each with varying GSR scores.

2. **Insert into Closed Calls List**
Add these conversations to the backend's closed calls list.

3. **Trigger API to Send Open Calls**
From the frontend or test client, send an API request with a defined time frame.

4. **Verify Front-End Display**
Check that each relevant conversation appears in the History Conversations page on the frontend, and that conversations outside the specified time frame are excluded.

Expected Result:

- All conversations added to the backend closed list within the requested time frame are displayed on the front-end's History Conversations page.
- Conversations outside the selected time frame are not shown.

Test Case 4 – Load Testing : Checking the Front-End with Multiple Users Simultaneously

Objective: To verify that the front-end system can handle multiple users simultaneously and respond with conversation data within an acceptable time frame (3–5 seconds).

Preconditions:

- The back-end system is running , accessible and configured to analyze GSR (suicide risk levels).
- Frontend is connected, actively polling or making API calls to retrieve history or open conversations.
- At least 4 registered users exist in the system.
- A timing mechanism (e.g., stopwatch or timer utility) is available for performance measurement.
- Selenium or a similar automation tool is set up to simulate multiple concurrent user sessions.

Steps:

1. **Generate Conversations**
Simulate 10 conversations with varying GSR scores (low, medium, high). Insert them into the open list at the back end.

2. **Started Timing (Stopper)**
Begin time measurement just before the users log in or hit the dashboard.

3.**Log in Users Simultaneously**
Use Selenium to simulate 4 users logging into the system at the same time and navigating to the Active or History Conversations page.

4.**Measure the time for conversation**

When the first conversation appears on the frontend interface for any user, stop the timer and record the response time.

Expected Result:

- The first conversation appears on the screen of all active users within 3–5 seconds.
- No frontend hangs, crashes, or unresponsive behavior is observed.
- Conversations are identical and consistent across all sessions.