# Code Documentation

## Webhook (src/webhook_receiver.py)

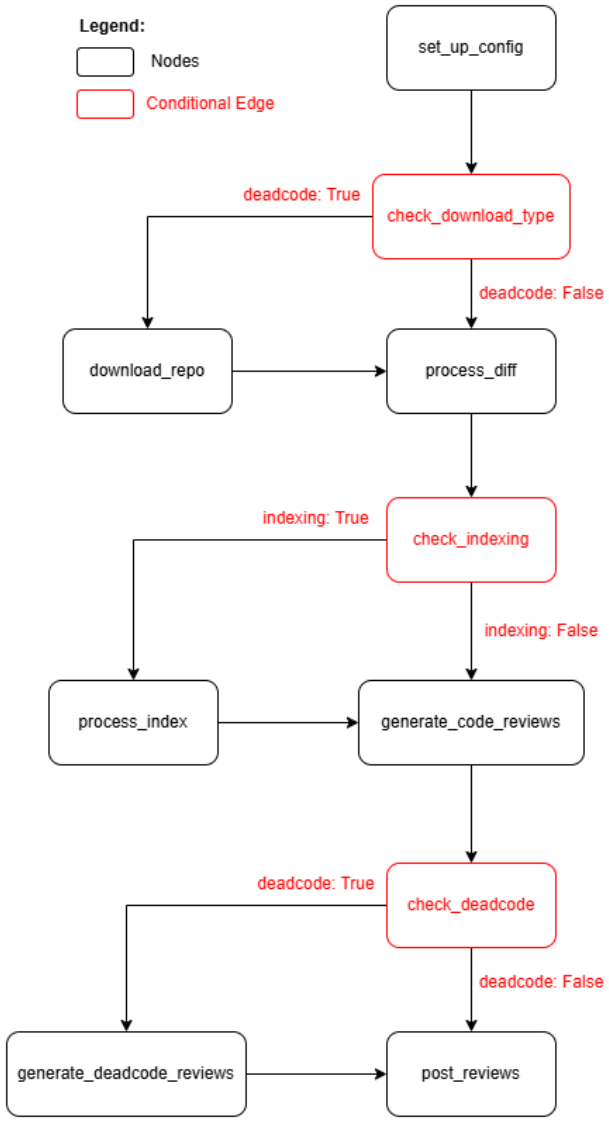
**Endpoint '/':**

1. Whenever a pull request is opened or modified, Bitbucket posts requests to webhook that calls ***trigger_review***.
    - *C*hecks if post request is coming from `pr:opened` or `pr:from_ref_updated` event:
        - If it is, the post request is queued for processing and webhook returns Status 202.
        - If it is not, the post request is ignored and webhook returns Status 200.
2. On webhook startup, 3 ***process_review_queue*** tasks are created. Maximum of 3 review processes can be run at the same time.
    - Dequeue post request for processing:
        - Extract payload information from post request.
        - Build LangGraph using payload information to run review process.

**Endpoint '/feedback':**

1. Calls ***save_feedback*** when '/feedback' endpoint is accessed.
    - Connect to Postgres table `feedback` to save feedback information.
2. Directs user to a HTML page that updates user on whether feedback was saved successfully and then automatically redirects user back to pull request.

# LangGraph Flow (src/review_code.py)




## Node Details

### Error handling in all nodes:

- Stops entire review process if any Exceptions are raised.
- Post a Bitbucket comment to update user that Sentinel code review failed with an error message.

### Metrics logging in all nodes: *log_metrics*

- Log task with pull request details in log file.
- If called at the 'else' clause of the node:

- Calculate duration that node took to complete. Add the duration to a list `duration` to calculate total duration at the end of the entire LangGraph flow.
- Log duration and total tokens used in the node (if tokens are being used) to log file along with the task completion message and pull request details

***set_up_config:***

1. *process_config_file:* Download and read values from sentinel-config file in Bitbucket repository.
2. Download .agents/.instructions file from the `doc_folder` set in the sentinel-config file.
3. Extra error handling:
    - Stops entire review process if `language` is not set to Python in sentinel-config.yaml file.

***process_diff:***

1. *get_diff:* Extract diffs from the pull request using Bitbucket API.
    - *diff_dict*: Dictionary to track modified line numbers in a file
        - Key: File
        - Values: Line number added, Line number removed
2. *get_modified_functions:* Identify modified functions in the pull request.
    - Using the diffs, download relevant files and their corresponding test files.
    - Parse through the downloaded files and find modified functions based on the line numbers added/removed.
    - *modified_func_dict:* Dictionary to track modified functions in a file
        - Key: File
        - Values: List of functions modified in the specific file
3. Extra error handling:
    - If `test_folder` not set in sentinel-config.yaml file, default to downloading test files from 'tests/' folder.
    - If default test file download fails, check for the source of error – missing sentinel-config.yaml OR missing `test_folder` value in sentinel-config.yaml file.
    - Stops entire review process if test files cannot be downloaded.

***download_repo:***

1. *download_all_files*: Download ALL files in the repository to prepare for dead code review.

***process_index:***

1. *embed_and_store_codebase*: Create/Update Qdrant with embeddings to prepare for semantic search during review process.
    - Check if a Qdrant collection exists for the repository.
    - If it does not exist, *create_index* is called to embed the entire repository.

- If it exists, *update_index* is called to update embeddings for modified functions/files only.

2. *create_index* process:
   - Download ALL files in the repository.
   - Create a Qdrant collection with formatted name `embeddings_for_<project>_<repo>`.
   - Get relevant information to prepare for embedding process.
     - Get full project description from README and use LLM to summarise the project description.
     - Get project structure (i.e. how the folders and files are structured in the repository).
   - Run **embedding process** for each downloaded file:
     - Generate function and file descriptions using LLM. Project description and structure is provided as context to generate more code-aware descriptions.
       - If the file is a yaml file, the yaml content is provided to LLM as dictionary to generate file description.
       - If the file has functions, generate individual function description for each function. Collate these individual function descriptions to generate file description.
       - If the file does not have functions, break the file code into smaller code chunks to generate descriptions for each smaller code chunks. Collate these smaller code chunk descriptions to generate file description.
     - Embed and store into Qdrant.
       - If the file is a yaml file or does not have functions, embed the file code into vector `code` and the file description into vector `description`. Store these 2 vectors and the payload (file, code, file_description) into Qdrant.
       - If the file has functions, embed the function code into vector `code` and the function + file description into vector `description`. Store these 2 vectors and the payload (file, function, code, file_description, func_description) into Qdrant.
     - Each point stored in Qdrant has a unique point_id. Save these point_id into a json file `qdrant_id_<project>_<repo>` for easy access during future updates of the collection.

3. *update_index* process:
   - Iterate through all modified functions in *modified_func_dict.*
     - Get point_id of the function from the corresponding `qdrant_id_<project>_<repo>` json file.
     - Account for different cases:
       - point_id exist → **Case 1:** Modified functions existed before the current PR and function code is modified.
       - point_id does not exist → Modified function is a new function added.
         - **Case 2:** Function added in an existing file.

        - **Case 3:** Function added in a new file.
    - Case 1:
        - Generate updated function and file descriptions using LLM.
        - Embed these updated descriptions into vector `description` and the updated code into vector `code`.
        - Update Qdrant with the new vectors and payload.
    - Case 2:
        - Generate function description using LLM. Get existing file description from Qdrant and generate an updated file description using LLM.
        - Embed these updated descriptions into vector `description` and the function code into vector `code`.
        - Store the new vectors and payload into Qdrant.
    - Case 3:
        - Run embedding process for the new file (refer to *create_index* process).

***generate_code_reviews:***

1. *review_documentation*: Generate file-level and function-level documentation reviews.
    - Iterate through all modified files in *modified_func_dict* to *review_documentation_by_file*
        - *review_file_name*:
            - Enhance original prompt to review file with additional information from .agents/.instructions file and with context from semantic search in Qdrant if indexing: True.
            - Use the enhanced prompt to generate file name reviews.
            - Add file name reviews into *file_level_review*.
        - *review_file_docstrings:*
            - If file has existing docstring, enhance original prompt to review file docstring with additional information from .agents/.instructions file and with context from semantic search in Qdrant if indexing: True then use the enhanced prompt to generate file docstring reviews.
            - If file does not have existing docstring, enhance original prompt to suggest file docstring with additional information from .agents/.instructions file and with context from semantic search in Qdrant if indexing: True then use the enhanced prompt to generate file docstring reviews.
            - Add file docstring reviews into *file_level_review*.
        - Store into a dictionary *file_review_dict* with key *file* and value *file_level_review.*
    - Iterate through all modified functions in *modified_func_dict* to *review_documentation_by_function*
        - *review_func_name:*

            - Enhance original prompt to review function name with additional information from .agents/.instructions file and with context from semantic search in Qdrant if indexing: True then use the enhanced prompt to generate function name reviews.
            - Add function name reviews into *function_level_review.*
        - *review_var_name:*
            - Enhance original prompt to review variable names with additional information from .agents/.instructions file and with context from semantic search in Qdrant if indexing: True then use the enhanced prompt to generate variable names reviews.
            - Validate the variable name reviews to check if the suggested renaming are significant and meaningful.
            - Add variables name reviews into *function_level_review.*
        - *review_func_docstrings:*
            - If function has existing docstring, enhance original prompt to review function docstring with additional information from .agents/.instructions file and with context from semantic search in Qdrant if indexing: True then use the enhanced prompt to generate function docstring reviews.
            - If function does not have existing docstring, enhance original prompt to suggest function docstring with additional information from .agents/.instructions file and with context from semantic search in Qdrant if indexing: True then use the enhanced prompt to generate function docstring reviews.
            - Add function docstring reviews into *function_level_review*.
        - Store into another dictionary *func_review_dict* with key *function* and value *function_level_review.*
    - Returns *file_review_dict* and *func_review_dict*.
2. *review_test*: Generate existing unit tests review (function-level) and missing unit tests review (file-level).
    - Iterate through all the modified file in *modified_func_dict*
        - Find the relevant test files for the modified file.
        - Call *get_test_cases* to get ALL test functions in the test file and *get_fixtures* to get ALL fixtures in the test file.
        - Iterate through modified functions in the file
            - Get all the test functions relevant to the function. Collate all these test functions into one *merged_function*.
                - *merged_function.func_code* is the code of all the test functions.
                - *merged_function.dependencies* is the dependencies of all the test functions.
            - *review_tested_functions:*
                - *generate_review_for_tested:*
                    - Extract context of the test functions – fixtures & dependencies

                - Enhance original prompt to review unit tests with the extracted context above, additional information from .agents/.instructions file and context from semantic search in Qdrant if indexing: True.
                - Use the enhanced prompt to generate unit test reviews.
                - Store into a dictionary *existing_test_review* with function as key and unit test review as value.
            - If there are test functions relevant to the modified function, add the modified function into *modified_and_tested* to keep track of functions that have existing unit tests.
    - *generate_review_for_untested:*
        - Compare *modified_and_tested* with *modified_func_dict* to get functions that have missing unit tests.
        - Generate a missing unit test review for these functions.
        - Store into another dictionary *missing_test_review* with file as key and missing test review as value.
    - Return *existing_test_review* and *missing_test_review* dictionaries.
3. *review_logic*: Generate individual commit review + Jira ticket review + Confluence review.
    - *set_up*:
        - Get PR description using Bitbucket API.
        - Get Jira tickets linked to the PR using Bitbucket API.
        - Get Confluence links in the Jira ticket using Jira API. Use these Confluence links to get the Confluence content using Confluence API.
        - Get the commits in the PR and then the diffs in each commit using Bitbucket API.
            - Diffs includes *diff_dict* (file as key, raw diff as value), *removed_files* and *added_files*.
            - Download the files in *diff_dict*.
    - Main logic review:
        - For each commit:
            - Identify the purpose of each file's modification.
                - Enhance the original prompt with additional information from .agents/.instructions file and with context from semantic search in Qdrant if indexing: True.
                - Use the enhanced prompt to identify the purpose of the file's modification.
            - Consolidate the purpose of each file's modification into *file_modification_purpose*.
            - Using the consolidated *file_modification purpose*, generate commit purpose and store into a list *purpose*.
            - If the commit message has a linked Jira ticket, generate individual commit reviews against the Jira ticket's summary and description.
                - Enhance the original prompt with additional information from .agents/.instructions file.

            - Use the enhanced prompt to generate individual commit review.
    - Consolidate all the commit purposes in the PR from the populated *purpose* list.
        - Use the consolidated commit purposes to generate Jira ticket review.
            - Enhance the original prompt with additional information from .agents/.instructions file.
            - Use the enhanced prompt to generate overall Jira ticket review.
        - If Confluence content is found, use the consolidated commit purposes to generate Confluence review.
            - Enhance the original prompt with additional information from .agents/.instructions file.
            - Use the enhanced prompt to generate Confluence review.
        - If Confluence content is not found, use the consolidated commit purposes and the PR description to generate points that should be included in a Confluence page as Confluence review.
            - Enhance the original prompt with additional information from .agents/.instructions file.
            - Use the enhanced prompt to generate points to include in Confluence page.
    - Consolidate individual reviews, Jira ticket review and Confluence review into comments that are below the word count limit. Return these comments in a list.

***generate_deadcode_reviews***:

1. *find_unused_code:*
    - 2 types of unused code: Unused pipelines/nodes & Dead code.
    - Unusued pipelines/nodes is specific to a personal repository and is skipped in most repositories.
2. Finding unused/undefined pipelines:
    - Extract and parse through all the pipeline files in the repository.
    - Find *all_pipelines* and *called_pipelines* in the pipeline files.
    - Unusued pipelines are pipelines that are in *all_pipelines* but not in *called_pipelines.* Store this into *unused_pipelines_review* list.
    - Undefined pipelines are pipelines that are in *called_pipelines* but not in *all_pipelines*. Store this into *undefined_pipelines_review* list.
3. Finding unusued/undefined nodes:
    - Find *nodes_called* using *all_pipelines* and *called_pipelines*.
    - Extract and parse through node files in the repository.
    - Find *all_nodes* in the node files.
    - Unusued nodes are nodes that are in *all_nodes* but not in *nodes_called.* Store this into *unused_nodes_review* list.

    - Undefined nodes are nodes that are in *nodes_called* but not in *all_nodes*. Store this into *undefined_nodes_review* list.
4. Finding deadcode:
    - Create a temporary file to simulate call of the function defined in *nodes_called*.
    - Run open-source deadcode library on the entire downloaded repository. Sort the reviews by folders from the most deadcode instances to the least deadcode instances.
    - Store the sorted dead code review into *deadcode_review* list.
5. Consolidate reviews:
    - Consolidate all the reviews – unused pipelines, undefined pipelines, unused nodes, undefined nodes, dead code – into comments that are below the word count limit.
    - Return the consolidated comments in a list.

***post_reviews:***

1. Consolidate documentation and unit test review into function-level and file-level reviews.
    - File-level: *file_level_review* in documentation review + *missing_test_review* in unit test review
    - Function-level: *function_level_review* in documentation review + *existing_test_review* in unit test review
2. Post Bitbucket comments in this sequence:
    - Dead code review
    - Consolidated documentation and test review
    - Logic review
3. Each comment is embedded with unique feedback links that points to the webhook's '/feedback' endpoint for user to rate the review comment.
4. Save the review metrics (project, repo, pr_id, duration, tokens, num_files, indexing, deadcode) into Postgres table *review_metrics*.

# Context Retrieval

**Vector Database: Qdrant**

- Collection consists of two 768-dimensional vectors – *code* and *description*
- Each point in the collection has the following payload/information:
    - File
    - Function
    - File Description
    - Function Description
    - Code

**Main semantic search function – *get_context (src/reviewer.py):***

1. Use function/file as query.

2. Embed query and use the embedded query to search for top 5 results in each of the *code* and *description* vectors.
3. Compare *code_hits* and *description_hits* to find overlapping hits. Description hits are preferred over code hits because the embedding model works better with natural language than with code.
    - If number of overlaps < 5, add in the remaining *description_hits* then return the payload of the overlaps + remaining *description_hits* as context.
    - If number of overlaps = 5, return the payload of the overlaps as context.