

ACID Evolution

Python Codes

Main Code

```
1 import os
2 import excavator
3 import constants
4 import pandas as pd
5 import pickle
6 import time
7 import datetime
8 import sys
9 import git
10 import nltk
11 nltk.download('punkt')
12 nltk.download('punkt_tab')
13 ...
14 ...
15 This script goes to each repo and mines
commits and commit messages and then get the defect category
16 ...
17 def getBranchName(path):
18     try:
19         repo = git.Repo(path)
20         default_branch_ref =
repo.git.symbolic_ref('refs/remotes/origin/HEAD')
21         default_branch =
default_branch_ref.replace('refs/remotes/origin/', '')
22         return default_branch
23     except Exception as e:
24         print(f"Error selecting Branch repo
{path}: {e}")
25     return None
26
27 def giveTimeStamp():
28     tsObj = time.time()
29     strToret =
datetime.datetime.fromtimestamp(tsObj).strftime('%Y-%m-%d
%H:%M:%S')
30     return strToret
31
32 if __name__ == '__main__':
33
34     t1 = time.time()
35     print('Started at:', giveTimeStamp())
36     print('*'*100)
37
38 flag_arg = sys.argv[sys.argv.index('--'
flag_arg) + 1]
39 if flag_arg == '-x':
40     orgName='EXTRA'
41     print('ACID will now run on extra
testing repos')
42     out_fil_nam = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY.PKL'
43     out_csv_file = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY_CATEG_OUTPUT_FINAL.csv'
44     out_pkl_file = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY_CATEG_OUTPUT_FINAL.PKL'
45 elif flag_arg == "-replication":
46     orgName = 'PIPR-replication'
47     print('ACID will now run on PIPr
Replication repos')
48     out_fil_nam = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY.PKL'
49     out_csv_file = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY_CATEG_OUTPUT_FINAL.csv'
50     out_pkl_file = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY_CATEG_OUTPUT_FINAL.PKL'
51 elif flag_arg == "-t":
52     orgName='TEST'
53     print('ACID will now run on default
testing repos')
54     out_fil_nam = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/TEST2_ONLY.PKL'
55     out_csv_file = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/TEST2_ONLY_CATEG_OUTPUT_FINAL.csv'
56     out_pkl_file = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/TEST2_ONLY_CATEG_OUTPUT_FINAL.PKL'
57 else:
58     orgName = flag_arg
59     print(f'ACID will now run on
{flag_arg} repos')
```

```

60         output_location =
os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
61         out_fil_nam = output_location +
f'{flag_arg}_ONLY.PKL'
62         out_csv_fil = output_location +
f'{flag_arg}_CATEG_OUTPUT_FINAL.csv'
63         out_pkl_fil = output_location +
f'{flag_arg}_ONLY_CATEG_OUTPUT_FINAL.PKL'
64
65         if orgName == 'EXTRA' or orgName == "TEST"
or orgName == "PIPR-replication":
66             csv_replication = None
67             csv_default = None
68         else:
69             csv_replication =
os.path.abspath(sys.argv[sys.argv.index("--csv-replication") +
1])
70             csv_default =
os.path.abspath(sys.argv[sys.argv.index("--csv-default") + 1])
71
72             script_dir =
os.path.dirname(os.path.abspath(__file__))
73             pathRepo = script_dir + "/" +
constants.DATASET_DIR + "/" + orgName + "/"
74             fileName = pathRepo + "/" +
constants.REPO_FILE_LIST
75             eligibleRepos =
excavator.getEligibleProjects(fileName)
76             dic = {}
77             categ = []
78             for proj_ in eligibleRepos:
79                 try:
80                     if proj_ ==
constants.REPO_FILE_LIST:
81                         continue
82                     path_proj = pathRepo + proj_
83                     branchName =
getBranchName(path_proj)
84                     per_proj_commit_dict,
per_proj_full_defect_list = excavator.runMiner(orgName, proj_,
branchName, csv_file_path=csv_replication,
csv_default=csv_default)
85                     categ = categ +
per_proj_full_defect_list
86                     # print proj_
len(per_proj_full_defect_list)
87                     print('Finished analyzing:',
proj_)
88                     dic[proj_] =
(per_proj_commit_dict, per_proj_full_defect_list)
89                     # print(dic[proj_])
90                     except Exception as e:
91                         print(e)
92                         print('='*50)
93
94                     all_proj_df = pd.DataFrame(categ)
95                     all_proj_df.to_csv(out_csv_fil, header=
['HASH', 'CATEG', 'REPO', 'TIME'], index=False)
96
97                     with open(out_pkl_fil, 'wb') as fp_:
98                         pickle.dump(dic, fp_)
99                         print('*'*100)
100                        print('Ended at:', giveTimeStamp())
101                        print('*'*100)
102                        t2 = time.time()
103                        time_diff = round( (t2 - t1) / 60, 5)
104                        print("Duration: {}"
minutes".format(time_diff))
105                        print('*'*100)

```

Main Concurrent Code

```

1 import os
2 import excavator
3 import constants
4 import pandas as pd
5 import pickle
6 import time
7 import datetime
8 import sys
9 import git
10 import concurrent.futures
11 import nltk
12 nltk.download('punkt')
13 nltk.download('punkt_tab')
14
15 ...
16 This script goes to each repo and mines
commits and commit messages and then get the defect category
17 ...
18 def getBranchName(path):
19     try:
20         repo = git.Repo(path)
21         default_branch_ref =
repo.git.symbolic_ref('refs/remotes/origin/HEAD')
22         default_branch =

```

```

default_branch_ref.replace('refs/remotes/origin/', '')
23     return default_branch
24 except Exception as e:
25     print(f"Error selecting Branch repo
{path}): {e}")
26     return None
27
28 def giveTimeStamp():
29     tsObj = time.time()
30     strToret =
datetime.datetime.fromtimestamp(tsObj).strftime('%Y-%m-%d
%H:%M:%S')
31     return strToret
32
33 if __name__=='__main__':
34
35     t1 = time.time()
36     print('Started at:', giveTimeStamp())
37     print('*'*100)
38
39 flag_arg = sys.argv[sys.argv.index("--"
flag_arg") + 1]
40     if flag_arg == '-x':
41         orgName='EXTRA'
42         print('ACID will now run on extra
testing repos')
43         out_fil_nam = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY.PKL'
44         out_csv_fil = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY_CATEG_OUTPUT_FINAL.csv'
45
46         out_pkl_fil = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/EXTRA2_TEST_ONLY_CATEG_OUTPUT_FINAL.PKL'
47
48     elif flag_arg == "-replication":
49         orgName = 'PIPR-replication'
50         print('ACID will now run on PIPr
Replication repos')
51         out_fil_nam = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY.PKL'
52         out_csv_fil = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY_CATEG_OUTPUT_FINAL.csv'
53
54         out_pkl_fil = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/REPLICATION_ONLY_CATEG_OUTPUT_FINAL.PKL'
55
56     elif flag_arg == "-t":
57         orgName='TEST'
58         print('ACID will now run on default
testing repos')
59         out_fil_nam = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/TEST2_ONLY.PKL'
60         out_csv_fil = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/TEST2_ONLY_CATEG_OUTPUT_FINAL.csv'
61         out_pkl_fil = '/home/aluno/ACID-
dataset/ARTIFACT/OUTPUT/TEST2_ONLY_CATEG_OUTPUT_FINAL.PKL'
62
63     else:
64         orgName = flag_arg
65         print(f'ACID will now run on
{flag_arg} repos')
66         output_location =
os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
67         out_fil_nam = output_location +
f'{flag_arg}_ONLY.PKL'
68         out_csv_fil = output_location +
f'{flag_arg}_CATEG_OUTPUT_FINAL.csv'
69         out_pkl_fil = output_location +
f'{flag_arg}_ONLY_CATEG_OUTPUT_FINAL.PKL'
70
71     def process_project(orgName, proj_,
pathRepo, dic, categ, csv_replication, csv_default):
72         try:
73             if proj_ == constants.REPO_FILE_LIST:
74                 per_proj_commit_dict,
75                 per_proj_full_defect_list = excavator.runMiner(orgName, proj_,
branchName, csv_replication, csv_default)
76                 categ += per_proj_full_defect_list
77                 dic[proj_] = (per_proj_commit_dict,
per_proj_full_defect_list)
78
79                 print(f'Finished analyzing: {proj_}')
80                 print("="*50)
81             except Exception as e:
82                 print(f'Error processing project
{proj_}): {e}')
83
84             def run_in_parallel(orgName, eligibleRepos,
pathRepo, dic, categ, csv_replication, csv_default):
85                 with

```

```

concurrent.futures.ThreadPoolExecutor() as executor:
    86         futures =
[executor.submit(process_project, orgName, proj_, pathRepo,
dic, categ, csv_replication, csv_default) for proj_ in
eligibleRepos]
    87
    88         # Wait for all tasks to complete and
handle exceptions
    89         for future in
concurrent.futures.as_completed(futures):
    90             try:
    91                 future.result() # Retrieve
the result of the task (or raise any exceptions)
    92             except Exception as e:
    93                 print(f"Task raised an
exception: {e}")
    94
    95             if orgName == 'EXTRA' or orgName == "TEST"
or orgName == "PIPR-replication":
    96                 csv_replication = None
    97                 csv_default = None
    98             else:
    99                 csv_replication =
os.path.abspath(sys.argv[sys.argv.index("--csv-replication") +
1])
    100                csv_default =
os.path.abspath(sys.argv[sys.argv.index("--csv-default") + 1])
    101
    102                script_dir =
os.path.dirname(os.path.abspath( file_))
    103                pathRepo = script_dir + "/" +
constants.DATASET_DIR + "/" + orgName + "/"
    104                fileName = pathRepo + "/" +
constants.REPO_FILE_LIST
    105                eligibleRepos =
excavator.getEligibleProjects(fileName)
    106                dic = {}
    107                categ = []
    108                run_in_parallel(orgName, eligibleRepos,
pathRepo, dic, categ, csv_replication, csv_default)
    109
    110                all_proj_df = pd.DataFrame(categ)
    111                all_proj_df.to_csv(out_csv_fil, header=
['HASH', 'CATEG', 'REPO', 'TIME'], index=False)
    112
    113                with open(out_pkl_fil, 'wb') as fp_:
    114                    pickle.dump(dic, fp_)
    115                    print('***100')
    116                    print('Ended at:', giveTimeStamp())
    117                    print('***100')
    118                    t2 = time.time()
    119                    time_diff = round( (t2 - t1) / 60, 5)
    120                    print("Duration: {}"
minutes".format(time_diff))
    121                    print('***100')

```

Excavator Code

```

1  from   nltk.tokenize import sent_tokenize
2  from   git           import Repo
3  import os
4  import csv
5  import numpy as np
6  import sys
7  import subprocess
8  import constants
9  import classifier
10 import ast
11 csv.field_size_limit(sys.maxsize)
12
13
14 def getEligibleProjects(fileNameParam):
15     repo_list = []
16     with open(fileNameParam,
constants.FILE_READ_MODE) as f:
17         reader = csv.reader(f)
18         for row in reader:
19             repo_list.append(row[0])
20     return repo_list
21
22 def
getPuppetFilesOfRepo(repo_dir_absolute_path):
23     pp_, non_pp = [], []
24     for root_, dirs, files_ in
os.walk(repo_dir_absolute_path):
25         for file_ in files_:
26             full_p_file = os.path.join(root_,
file_)
27             if((os.path.exists(full_p_file)) and
(constants.AST_PATH not in full_p_file) ):
28                 if
(full_p_file.endswith(constants.PP_EXTENSION)):
29                     if any(full_p_file.endswith(ext) for
ext in constants.IAC_FILES):
30                         pp_.append(full_p_file)
31     return pp_

```

```

    32
    33     def getRelPathOfFile(all_pp_param,
repo_dir_absolute_path):
    34         common_path = repo_dir_absolute_path
    35         files_relative_paths =
os.path.relpath(path, common_path) for path in all_pp_param]
    36             return files_relative_paths
    37
    38     def
getPuppRelatedCommits(repo_dir_absolute_path, ppListinRepo,
branchName=constants.MASTER_BRANCH):
    39         mappedPuppetList=[]
    40         track_exec_cnt = 0
    41         repo_ = Repo(repo_dir_absolute_path)
    42         all_commits =
list(repo_.iter_commits(branchName))
    43             for each_commit in all_commits:
    44                 track_exec_cnt = track_exec_cnt + 1
    45
    46             cmd_of_interest1 =
constants.CHANGE_DIR_CMD + repo_dir_absolute_path + " ; "
    47             cmd_of_interest2 =
constants.GIT_COMM_CMD_1 + str(each_commit) +
constants.GIT_COMM_CMD_2
    48             cmd_of_interest = cmd_of_interest1 +
cmd_of_interest2
    49             commit_of_interest =
str(subprocess.check_output([constants.BASH_CMD,
constants.BASH_FLAG, cmd_of_interest])) #in Python 3
subprocess.check_output returns byte
    50
    51             for ppFile in ppListinRepo:
    52                 if ppFile in commit_of_interest:
    53                     file_with_path =
os.path.join(repo_dir_absolute_path, ppfile)
    54                     mapped_tuple = (file_with_path,
each_commit)
    55                     mappedPuppetList.append(mapped_tuple)
    56
    57         return mappedPuppetList
    58
    59     def IacRelatedCommits(repo_dir_absolute_path,
iac_list_repo, branchName=constants.MASTER_BRANCH):
    60         mapped_iac_list = []
    61         track_exec_cnt = 0
    62         repo_ = Repo(repo_dir_absolute_path)
    63         all_commits =
list(repo_.iter_commits(branchName))
    64             for each_commit in all_commits:
    65                 track_exec_cnt = track_exec_cnt + 1
    66
    67                 cmd_of_interest1 =
constants.CHANGE_DIR_CMD + repo_dir_absolute_path + " ; "
    68                 cmd_of_interest2 =
constants.GIT_COMM_CMD_1 + str(each_commit) +
constants.GIT_COMM_CMD_2
    69                 cmd_of_interest = cmd_of_interest1 +
cmd_of_interest2
    70                 commit_of_interest =
str(subprocess.check_output([constants.BASH_CMD,
constants.BASH_FLAG, cmd_of_interest])) #in Python 3
subprocess.check_output returns byte
    71
    72                 for iac_file in iac_list_repo:
    73                     if iac_file in commit_of_interest:
    74                         file_with_path =
os.path.join(repo_dir_absolute_path, iac_file)
    75                         mapped_tuple = (file_with_path,
each_commit)
    76                         mapped_iac_list.append(mapped_tuple)
    77
    78         return mapped_iac_list
    79
    80     def getDiffStr(repo_path_p, commit_hash_p,
file_p):
    81         cdCommand = f'{constants.CHANGE_DIR_CMD}{repo_path_p} ; '
    82         theFile = os.path.relpath(file_p,
repo_path_p)
    83
    84         diffCommand = f'{constants.GIT_SHOW_CMD}{commit_hash_p} -- {theFile}'
    85         command2Run = cdCommand + diffCommand
    86
    87         diff_output = subprocess.check_output(
constants.BASH_CMD,
    88             [constants.BASH_CMD, command2Run]
    89             ).decode('utf-8')
    90         return diff_output
    91
    92     def makeDepParsingMessage(m_, i_):
    93         upper, lower = 0, 0
    94         lower = i_- constants.STR_LIST_BOUNDS
    95         upper = i_+ constants.STR_LIST_BOUNDS
    96         if upper > len(m_):
    97             upper = -1
    98         if lower < 0:

```

```

    99      lower = 0
100     return constants.WHITE_SPACE.join(m_[i_ -
constants.STR_LIST_BOUNDS : i_ + constants.STR_LIST_BOUNDS])
101
102     def processMessage(indi_comm_mess):
103         splitted_messages = []
104         # original
105         # if ('*' in indi_comm_mess):
106         if ('*' in indi_comm_mess) or (';' in
indi_comm_mess):
107             splitted_messages =
indi_comm_mess.split('*')
108             splitted_messages =
indi_comm_mess.split(';')
109             else:
110                 splitted_messages =
sent_tokenize(indi_comm_mess)
111             return splitted_messages
112
113     def analyzeCommit(repo_path_param,
iac_commits_mapping):
114         verbose = False # For oracle dataset it is
True (later), otherwise it is False
115         pupp_bug_list = []
116         all_commit_file_dict = {}
117         all_defect_categ_list = []
118         hash_tracker = []
119         for tuple_ in iac_commits_mapping:
120
121             file_ = tuple_[0]
122             commit_ = tuple_[1]
123             msg_commit = commit_.message
124
125             msg_commit = msg_commit.replace('\n',
constants.WHITE_SPACE)
126             msg_commit = msg_commit.replace(',', ' ')
127             msg_commit = msg_commit.replace('\t',
constants.WHITE_SPACE)
128             msg_commit = msg_commit.replace('&', ' ')
129             msg_commit = msg_commit.replace('#',
constants.WHITE_SPACE)
130             msg_commit = msg_commit.replace('=', ,
constants.WHITE_SPACE)
131
132             commit_hash = commit_.hexsha
133
134             # '''
135             # for testing purpose , uncomment only for
tool accuracy purpose
136             # '''
137             # if commit hash in
constants.ORACLE_HASH_CHECKLIST:
138                 # verbose = True
139                 # else:
140                 # verbose = False
141                 # '''
142                 # '''
143
144             timestamp_commit =
commit_.committed_datetime
145             str_time_commit =
timestamp_commit.strftime(constants.DATE_TIME_FORMAT) ## date
with time
146
147             ##### categorization zone
148             per_commit_defect_categ_list = []
149             if (commit_hash not in hash_tracker):
150                 bug_status, index_status =
classifier.detectBuggyCommit(msg_commit, verbose)
151                 # print bug_status
152                 #if commit_hash ==
"1f03639bcddb66031b16ed6cfdf91f2bbdeca6c8":
153                     #bug_status = True
154                     if (bug_status) or
(classifier.detectRevertedCommit(msg_commit) ):
155                         processed_message =
processMessage(msg_commit)
156                         # each commit has multiple messages,
need to merge them together in one list here, not in
classifier
157                         for tokenized_msg in
processed_message:
158                             diff_content_str =
getDiffStr(repo_path_param, commit_hash, file_)
159                             bug_categ =
classifier.detectCateg(tokenized_msg, diff_content_str,
verbose)
160                             # bug_categ =
classifier.detectCategHashFounder(tokenized_msg,
diff_content_str, verbose, hash=commit_hash)
161                             if len(bug_categ) == 0:
162
per_commit_defect_categ_list.append(constants.BUGGY_COMMIT)
163                             else:
164                                 per_commit_defect_categ_list +=
bug_categ
165                             else:

```

```

166         per_commit_defect_categ_list =
[constants.NO_DEFECT_CATEG]
167
168         bug_categ_list =
np.unique(per_commit_defect_categ_list)
169         ...
170         for testing purpose , uncomment only for
tool accuracy purpose
171         ...
172         # if verbose:
173         #   print bug_categ_list
174         ...
175         ...
176         if (len(bug_categ_list) > 0):
177             for bug_categ_ in bug_categ_list:
178                 tup_ = (commit_hash, bug_categ_,
file_, str_time_commit)
179                 all_defect_categ_list.append(tup_)
180                 # -- my debug
181                 #if (tup_[1] !=
constants.NO_DEFECT_CATEG):
182                     # print(tup_[1])
183                     # ---
184                     # print tup_[0], tup_[1], tup_[2],
tup_[3]
185                     # print '-'*25
186             else:
187                 tup_ = (commit_hash,
constants.NO_DEFECT_CATEG, file_, str_time_commit)
188                 all_defect_categ_list.append(tup_)
189
190                 hash_tracker.append(commit_hash)
191                 ##### file to hash mapping zone
192                 if commit_hash not in
all_commit_file_dict:
193                     all_commit_file_dict[commit_hash] =
[file_]
194                 else:
195                     all_commit_file_dict[commit_hash] =
all_commit_file_dict[commit_hash] + [file_]
196
197                 return all_commit_file_dict,
all_defect_categ_list
198
199     def getIacFilesOfRepo(repo_id, csv_file_path =
constants.CSV_REPLICATION, csv_default =
constants.CSV_DEFAULT_PATH):
200         with open(csv_file_path, 'r',
encoding='utf-8') as csv_file:
201             reader = csv.DictReader(csv_file)
202             for row in reader:
203                 if row['id'] == str(repo_id):
204                     # Converter strings de listas
para listas reais
205                     iac_paths =
ast.literal_eval(row['iac_paths'])
206                     related_files =
ast.literal_eval(row['related_files'])
207                     result = iac_paths +
related_files
208                     result =
[item.replace(csv_default + "/" + repo_id + '/', '') for item
in result]
209                     return result
210             return None, None
211
212     def getId(path):
213         return path.split('/')[-1]
214
215     def runMiner(orgParamName, repo_name_param,
branchParam, csv_file_path = None, csv_default = None):
216         script_dir =
os.path.dirname(os.path.abspath(__file__))
217         repo_path = script_dir + "/" +
constants.DATASET_DIR + "/" + orgParamName + "/" +
repo_name_param
218         repo_branch = branchParam
219
220         repo_id = getId(repo_path)
221         all_iac_files_in_repo =
getIacFilesOfRepo(repo_id, csv_file_path, csv_default)
222         iac_commits_in_repo =
IacRelatedCommits(repo_path, all_iac_files_in_repo,
repo_branch)
223         commit_file_dict, categ_defect_list =
analyzeCommit(repo_path, iac_commits_in_repo)
224
225         return commit_file_dict, categ_defect_list
226
227
228
229     def dumpContentToFile(strP, fileP):
230         fileToWrite = open(fileP,
constants.FILE_WRITE_MODE)
231         fileToWrite.write(strP)
232         fileToWrite.close()
233         return str(os.stat(fileP).st_size)

```

Difference Parser Code

```

1  #reff:
https://github.com/cscorley/whatthepatch
2  import whatthepatch
3
4  import constants
5
6  from fuzzywuzzy import fuzz
7
8  import re
9
10 # [(1, None, '# == Class cdh4::pig'), (None,
1, '# == Class cdh4::pig'), (2, 2, '#'), (3, None, '# Installs
and configures Apache Pig.'), (None, 3, '# Installs and
configures Apache Pig and Pig DataFu.'), (4, 4, '#'), (5,
None, 'class cdh4::pig {}'), (6, None, " package { 'pig':",
(7, None, "    ensure => 'installed'."), (8, None, ' '),
(None, 5, 'class cdh4::pig()'), (None, 6, "
$pig_properties_template = 'cdh/pig/pig.properties.erb',"),
(None, 7, "    $log4j_template =
'cdn/pig/log4j.properties.erb',"), (None, 8, ')'), (None, 9,
'{'), (None, 10, '    # cdh4::pig requires hadoop client and
configs are installed.'), (None, 11, "    Class['cdh:hadoop']
-> Class['cdh4::pig']).(9, 12, ''), (10, None, "    file {
'/etc/pig/conf/pig.properties':"), (11, None, "        content =>
template('cdh4/pig/pig.properties.erb'),"), (12, None, "
require => Package['pig'].(13, None, ' '),
(None, 13, "    package { 'pig':"),
(None, 14, "        ensure =>
'installed'."), (None, 15, ' '),
(None, 16, "        package {
'pig-udf-datafu':"),
(None, 17, "        ensure =>
'installed'."), (None, 18, ' '),
(None, 19, ' '),
(None, 20, '        $config_directory =
'/etc/pig/conf${cdh:hadoop:cluster_name}"""),
(None, 21, "
# Create the $cluster_name based $config_directory.'),
(None,
22, "        file { $config_directory:"),
(None, 23, "
ensure => 'directory'."), (None, 24, "        require =>
Package['pig']."),
(None, 25, ' '),
(None, 26, "
cdh:alternative { 'pig-conf':"),
(None, 27, "        link
=> '/etc/pig/conf')."),
(None, 28, "        path
=> $config_directory.'),
(None, 29, ' '),
(None, 30, ''),
(None, 31, "        file {
"${config_directory}/pig.properties":"),
(None, 32, "
content => template($pig_properties_template)."),
(None, 33, "
require => Package['pig']."),
(None, 34, ' '),
(None, 35,
'        file { ${config_directory}/log4j.properties':"),
(None,
36, "
content => template($log4j_template)."),
(None,
37, "
require => Package['pig']."),
(None, 38, '
}), (14, 39, ''))
11
12  def parseTheDiff(diff_text):
13      parse_out_dict = {}
14      diff_mess_str = str(diff_text) ## changes
for Python 3 migration
15      for diff_ in
whatthepatch.parse_patch(diff_mess_str):
16          all_changes_line_by_line = diff_[1] ##
diff_ is a tuple, changes is identified by the second index
17          line_numbers_added,
line_numbers_deleted = [], []
18          add_dic, del_dic = {}, {}
19          parse_out_dict = {}
20          for change_tuple in
all_changes_line_by_line:
21              if (change_tuple[0] != None ):
22
line_numbers_added.append(change_tuple[0])
23              add_dic[change_tuple[0]] =
change_tuple[2]
24              if (change_tuple[1] != None ):
25
line_numbers_deleted.append(change_tuple[1])
26              del_dic[change_tuple[1]] =
change_tuple[2]
27              lines_changed =
list(set(line_numbers_added).intersection(line_numbers_deleted))
28
for line_number in lines_changed:
29              if ((line_number in add_dic) and
(line_number in del_dic)):
30                  parse_out_dict[line_number] =
[del_dic[line_number], add_dic[line_number]] ## <removed
content, added content>
31                  #print parse_out_dict
32
return parse_out_dict
33
34  def filterTextList(txt_lis):
35      return_list = []
36      return_list = [x_.lower() for x_ in
txt_lis if constants.HASH_SYMBOL not in x_]
37      return_list = [x_.replace(constants.TAB,
'') for x_ in return_list]
38      return_list =
[x_.replace(constants.NEWLINE, '') for x_ in return_list ]

```

```

39     return_list = [x_ for x_ in return_list if
40         len(x_) > 1 ]
41     return return_list
42 
43     def getAddDelLines(diff_mess):
44         added_text, deleted_text = [], []
45         diff_mess_str = str(diff_mess)
46         try:
47             for diff_ in
whatthepatch.parse_patch(diff_mess_str):
48                 all_changes_line_by_line =
getattr(diff_, "changes", None)
49                 if all_changes_line_by_line:
50                     for change in
all_changes_line_by_line:
51                         if change.new is not None:
52                             added_text.append(change.line)
53                         if change.old is not None:
54                             deleted_text.append(change.line)
55 
56             except Exception as e:
57                 print(f"[ERROR] Error when processing
diff: {e}")
58 
59             return added_text, deleted_text
60 
61     def getSpecialConfigDict(text_str_list,
splitter):
62         dic2ret = {}
63         for x_ in text_str_list:
64             if (splitter in x_):
65                 _key_ =
x_.replace(constants.WHITE_SPACE, '').split(splitter)[0]
66                 _val_ =
x_.replace(constants.WHITE_SPACE, '').split(splitter)[-1]
67                 if _key_ not in dic2ret:
68                     dic2ret[_key_] = _val_
69 
70             # print text_str_list
71             # print dic2ret
72         return dic2ret
73 
74     def filterConfig(oldValue):
75         oldValue = oldValue.replace(" ", "", "")
76         oldValue = oldValue.replace("\t", "", "")
77         oldValue = oldValue.replace("\n", "", "")
78         val_ = oldValue.replace(">", "", "")
79 
80         return val_
81 
82     def getConfigChangeCnt(start_dict, end_dict):
83         tracker = 0
84         track_list = []
85         val_track_list = []
86         for k_, v_ in start_dict.items():
87             if (k_ in end_dict) and (k_ not in
track_list) and (v_ not in val_track_list) and (len(v_) > 1):
88                 oldValue = end_dict[k_]
89                 newValue = v_
90                 # need more pre procession ugh
91                 oldValue = filterConfig(oldValue)
92                 newValue = filterConfig(newValue)
93                 if newValue != oldValue:
94                     # print k
95                     # print oldValue, newValue
96                     tracker = tracker + 1
97                     track_list.append(k_)
98                     val_track_list.append(v_)
99 
# print '>*'5
100    return tracker
101 
102    def checkDiffForConfigDefects(diff_text):
103        added_text , deleted_text = [], []
104        final_flag = False
105        added_text, deleted_text =
getAddDelLines(diff_text)
106        added_text = filterTextList(added_text)
107        deleted_text =
filterTextList(deleted_text)
108        config_change_tracker = 0
109 
110        valu_assi_dict_addi =
getSpecialConfigDict(added_text, constants.VAR_SIGN)
111        valu_assi_dict_deli =
getSpecialConfigDict(deleted_text, constants.VAR_SIGN)
112 
113        attr_assi_dict_addi =
getSpecialConfigDict(added_text, constants.ATTR_SIGN)
114        attr_assi_dict_deli =
getSpecialConfigDict(deleted_text, constants.ATTR_SIGN)
115 
116        # config_change_tracker =
getConfigChangeCnt(valu_assi_dict_addi, valu_assi_dict_deli) +
getConfigChangeCnt(valu_assi_dict_deli, valu_assi_dict_addi) +

```

```
getConfigChangeCnt(attr_assi_dict_addi, attr_assi_dict_deli) +  
getConfigChangeCnt(attr_assi_dict_deli, attr_assi_dict_addi)  
    117     config_change_tracker =  
getConfigChangeCnt(valu_assi_dict_addi, valu_assi_dict_deli) +  
getConfigChangeCnt(attr_assi_dict_addi, attr_assi_dict_deli)  
    118  
    119     if config_change_tracker > 0 :  
    120         final_flag = True  
    121  
    122     return final_flag  
    123  
    124 def checkDiffForDepDefects(diff_text):  
    125     added_text, deleted_text = [], []  
    126     final_flag, final_flag_1, final_flag_2 =  
False, False, False  
    127     added_text, deleted_text =  
getAddDellines(diff_text)  
    128     added_text = filterTextList(added_text)  
    129     deleted_text =  
filterTextList(deleted_text)  
    130     added_text = [x_ for x_ in added_text if  
constants.VAR_SIGN not in x_]  
    131     added_text = [x_ for x_ in added_text if  
constants.ATTR_SIGN not in x_]  
    132     deleted_text = [x_ for x_ in deleted_text if  
constants.VAR_SIGN not in x_]  
    133     deleted_text = [x_ for x_ in  
deleted_text if constants.ATTR_SIGN not in x_]  
    134     deleted_text = [x_ for x_ in  
deleted_text if constants.VAR_SIGN not in x_]  
    135     deleted_text = [x_ for x_ in  
deleted_text if constants.ATTR_SIGN not in x_]  
    136     # print added_text, deleted_text  
    137     added_text = [z_ for z_ in added_text if  
any(x_ in z_ for x_ in constants.diff_depen_code_elems ) ]  
    138     deleted_text = [z_ for z_ in deleted_text if  
any(x_ in z_ for x_ in constants.diff_depen_code_elems ) ]  
    139  
    140     if (len(added_text) > 0 ) and  
(len(deleted_text) > 0 ) :  
    141         final_flag = True  
    142  
    143     return final_flag  
    144  
    145     import re  
    146  
    147     def has_comment(line):  
    148         # ignore if it's a string  
    149         if re.fullmatch(r'\s*["\'].*["\']\s*',  
line.strip()):  
    150             return False  
    151  
    152         # line's entire comment  
    153         if re.search(r'^\s*(#//).+', line):  
    154             return True  
    155  
    156         # inline comment  
    157         if re.search(r'^[^"]*(#//).+', line):  
    158             return True  
    159  
    160         # blocks comment  
    161         if re.search(r'/*.*?\*/', line,  
re.DOTALL):  
    162             return True  
    163  
    164     return False  
    165  
    166     def checkDiffForDocDefects(diff_text):  
    167         lines_changed = []  
    168         final_flag = False  
    169         diff_mess_str = str(diff_text) ## changes  
for Python 3 migration  
    170         for diff_ in  
whatthepatch.parse_patch(diff_mess_str):  
    171             all_changes_line_by_line = diff_[1]  
    172             line_numbers_added,  
line_numbers_deleted = [], []  
    173             if all_changes_line_by_line is not  
None:  
    174                 for change_tuple in  
all_changes_line_by_line:  
    175                     content = change_tuple[2]  
    176                     content =  
content.replace(constants.WHITE_SPACE, '')  
    177                     if change_tuple[0] is not None  
and has_comment(content):  
    178                         line_numbers_added.append(content)  
    179                         if change_tuple[1] is not None  
and has_comment(content):  
    180                             line_numbers_deleted.append(content)  
    181                             lines_changed =  
list(set(line_numbers_added).intersection(line_numbers_deleted))  
    182  
    183             # print lines_changed  
    184             lines_changed = [x_ for x_ in  
lines_changed if len(x_) > 1 ]  
    185             if len(lines_changed) > 0:  
    186                 final_flag = True  
    187  
    188             return final_flag
```

```

185         final_flag = True
186     return final_flag
187
188 def checkDiffForNetwork(diff_text):
189     added_text, deleted_text = [], []
190     final_flag = False
191     added_text, deleted_text =
getAddDelLines(diff_text)
192
193     added_text = filterTextList(added_text)
194     deleted_text =
filterTextList(deleted_text)
195
196     added_text = [line for line in added_text
if any(keyword in line for keyword in
constants.diff_network_elems)]
197     deleted_text = [line for line in deleted_text
if any(keyword in line for keyword in
constants.diff_network_elems)]
198
199     if added_text or deleted_text:
200         final_flag = True
201
202     return final_flag
203
204 def checkDiffForCredentials(diff_text):
205     added_text, deleted_text = [], []
206     final_flag = False
207     added_text, deleted_text =
getAddDelLines(diff_text)
208
209     added_text = filterTextList(added_text)
210     deleted_text =
filterTextList(deleted_text)
211
212     added_text = [line for line in added_text
if any(keyword in line for keyword in
constants.diff_credentials_kw_list)]
213     deleted_text = [line for line in deleted_text
if any(keyword in line for keyword in
constants.diff_credentials_kw_list)]
214
215     if added_text or deleted_text:
216         final_flag = True
217
218     return final_flag
219
220 def checkDiffForLogicDefects(diff_text):
221     added_text , deleted_text = [], []
222     final_flag, final_flag_1, final_flag_2 =
False , False, False
223     added_text, deleted_text =
getAddDelLines(diff_text)
224     added_text = filterTextList(added_text)
225     deleted_text =
filterTextList(deleted_text)
226     added_text = [x_ for x_ in added_text if
constants.VAR_SIGN not in x_ ]
227     added_text = [x_ for x_ in added_text if
constants.ATTR_SIGN not in x_ ]
228
229     deleted_text = [x_ for x_ in
deleted_text if constants.VAR_SIGN not in x_ ]
230     deleted_text = [x_ for x_ in
deleted_text if constants.ATTR_SIGN not in x_ ]
231     # print added_text, deleted_text
232     added_text = [z_ for z_ in added_text if
any(x_ in z_ for x_ in constants.diff_logic_code_elems ) ]
233     deleted_text = [z_ for z_ in deleted_text
if any(x_ in z_ for x_ in constants.diff_logic_code_elems ) ]
234
235     if (len(added_text) > 0 ) or
(len(deleted_text) > 0 ) :
236         final_flag = True
237
238     return final_flag
239
240 def checkDiffForSecurityDefects(diff_text):
241     final_flag = False
242     added_text , deleted_text = [], []
243
244     added_text, deleted_text =
getAddDelLines(diff_text)
245     added_text = filterTextList(added_text)
246     deleted_text =
filterTextList(deleted_text)
247     added_text = [x_ for x_ in added_text if
constants.VAR_SIGN in x_ ]
248     added_text = [x_ for x_ in added_text if
constants.ATTR_SIGN in x_ ]
249
250     deleted_text = [x_ for x_ in deleted_text
if constants.VAR_SIGN in x_ ]
251     deleted_text = [x_ for x_ in deleted_text
if constants.ATTR_SIGN in x_ ]
252
253     added_text =
[x_.split(constants.VAR_SIGN)

```

```

[0].replace(constants.WHITE_SPACE, '') for x_ in added_text]
253     added_text =
[x_.split(constants.ATTR_SIGN)
[0].replace(constants.WHITE_SPACE, '') for x_ in deleted_text]
254     deleted_text =
[x_.split(constants.VAR_SIGN)
[0].replace(constants.WHITE_SPACE, '') for x_ in deleted_text]
255     deleted_text =
[x_.split(constants.ATTR_SIGN)
[0].replace(constants.WHITE_SPACE, '') for x_ in deleted_text]
256     deleted_text =
[x_.split(constants.VAR_SIGN)
[0].replace(constants.WHITE_SPACE, '') for x_ in deleted_text]
257     deleted_text =
any(x_ in z_ for x_ in constants.diff_secu_code_elems )
258     added_text = [z_ for z_ in added_text if
any(x_ in z_ for x_ in constants.diff_secu_code_elems )
259     deleted_text = [z_ for z_ in deleted_text if
any(x_ in z_ for x_ in constants.diff_secu_code_elems )
260     # print added_text, deleted_text
261     if (len(added_text) > 0) or
(len(deleted_text) > 0):
262         final_flag = True
263     return final_flag
264
265 def checkDiffForServiceDefects(diff_text):
266     final_flag = False
267     added_text, deleted_text = [], []
268
269     added_text, deleted_text =
getAddDellines(diff_text)
270     added_text = filterTextList(added_text)
271     deleted_text =
filterTextList(deleted_text)
272     added_text = [x_ for x_ in added_text if
constants.VAR_SIGN not in x_]
273     added_text = [x_.lower() for x_ in
added_text if constants.ATTR_SIGN not in x_]
274
275     deleted_text = [x_ for x_ in deleted_text if
constants.VAR_SIGN not in x_]
276     deleted_text = [x_.lower() for x_ in
deleted_text if constants.ATTR_SIGN not in x_]
277
278     added_text = [z_ for z_ in added_text if
any(x_ in z_ for x_ in constants.diff_service_code_elems )]
279     deleted_text = [z_ for z_ in deleted_text if
any(x_ in z_ for x_ in constants.diff_service_code_elems )]
280
281     if (len(added_text) > 0) and
(len(deleted_text) > 0) :
282         final_flag = True
283     return final_flag
284
285 def matchStringsFuzzily(add_str_lis,
del_str_lis):
286     # takes two string as input, returns
levenshtein's ratio, ref:
https://www.datacamp.com/community/tutorials/fuzzy-string-python
287     add_str =
constants.WHITE_SPACE.join(add_str_lis)
288     del_str =
constants.WHITE_SPACE.join(del_str_lis)
289     lower_add_str = add_str.lower()
290     lower_del_str = del_str.lower()
291     lev_str_ratio = fuzz.token_sort_ratio(
lower_add_str, lower_del_str ) ## this is levenshtein ratio,
in a sorted manner
292     return lev_str_ratio
293
294
295 def checkDiffForSyntaxDefects(diff_text):
296     final_flag = False
297     added_text, deleted_text = [], []
298     attr_added_text, attr_deleted_text = [],
[]
299     var_added_text, var_deleted_text = [], []
300
301     added_text, deleted_text =
getAddDellines(diff_text)
302     added_text = filterTextList(added_text)
303     deleted_text =
filterTextList(deleted_text)
304
305     ...
306     look for variable name change
307     ...
308     attr_added_text = [x_.lower() for x_ in
added_text if constants.ATTR_SIGN in x_]
309     var_added_text =
[x_.lower().replace(constants.WHITE_SPACE, '') for x_ in
added_text if constants.VAR_SIGN in x_]
310
311     attr_deleted_text = [x_.lower() for x_ in
deleted_text if constants.ATTR_SIGN in x_]
312     var_deleted_text =
[x_.lower().replace(constants.WHITE_SPACE, '') for x_ in
deleted_text if constants.VAR_SIGN in x_]
313     ...

```

```

314     Now compare
315     ...
316
317     # if (len(added_text)) and
318     # if ((len(attr_added_text)) ==
319     # (len(attr_deleted_text)) or (len(var_added_text) ==
320     # len(var_deleted_text) ) : ## right logic , same number of
321     # additions and deletions for variables
322     final_flag = True
323
324     elif (
325     (matchStringsFuzzily(attr_added_text, attr_deleted_text) >
326      constants.lev_cutoff ) or (matchStringsFuzzily(var_added_text,
327      var_deleted_text) > constants.lev_cutoff ) );
328
329     # Why does the original author uses
330     minus?
331
332     # final_flag - True
333     final_flag = True
334
335
336
337
338     return final_flag
339
340
341
342
343
344     def checkDiffForIdempotenceDefects(diff_text):
345         final_flag = False
346         added_text , deleted_text = [], []
347
348         added_text, deleted_text =
349         getAddDelLines(diff_text)
350         added_text = filterTextList(added_text)
351         deleted_text =
352         filterTextList(deleted_text)
353
354         if(len(deleted_text) > len(added_text)):
355             for text_ in added_text:
356                 for elem in
357                 constants.diff_extra_idem_elems:
358                     if elem in text_:
359                         flag_list.append(True)
360
361         if (len(flag_list) > 0):
362             final_flag = True
363
364     return final_flag

```

Classifier Code

```

1  import constants
2  import diff_parser
3  import re
4  import spacy
5  spacy_engine =
spacy.load(constants.SPACY_ENG_DICT)
6  from nltk.stem.porter import *
7  stemmerObj = PorterStemmer()
8
9
10
11 def checkForNum(str_par):
12     return any(char_.isdigit() for char_ in
str_par)
13
14 def filterCommitMessage(msg_par):
15     temp_msg_ = msg_par.lower()
16     splitted_msg =
temp_msg_.split(constants.WHITE_SPACE)
17     splitted_msg = [stemmerObj.stem(x_) for x_
in splitted_msg] ##porter stemming , x_ is a string
18     splitted_msg = [x_ for x_ in splitted_msg
if len(x_) > 1 ] ## remove special characterers , x_ is a
string
19     # splitted_msg = [x_ for x_ in
splitted_msg if x_.isalnum() ] ## remove special characterers
, x_ is a string
20     filtered_msg = [x_ for x_ in splitted_msg
if checkForNum(x_) == False ] ## remove alphanumeric
characters , x_ is a string

```

```

21     return filtered_msg
22
23
24 def doDepAnalysis(msg_par):
25     msg_to_analyze = []
26     filtered_msg =
filterCommitMessage(msg_par)
27     unicode_msg_ =
constants.WHITE_SPACE.join(filtered_msg)
28     try:
29         unicode_msg_ = str(unicode_msg_,
constants.UTF_ENCODING)
30     except:
31         unicode_msg_ = unicode_msg_
# print unicode_msg_
32     spacy_doc = spacy_engine(unicode_msg_)
33     for token in spacy_doc:
34         if (token.dep_ ==
constants.ROOT_TOKEN):
35             for x_ in token.children:
36                 msg_to_analyze.append(x_.text)
37
38     return
constants.WHITE_SPACE.join(msg_to_analyze)
39
40
41 def doTempCleanUp(msg_str):
42     msg_ = msg_str.replace(constants.CLOSE_KW,
constants.WHITE_SPACE)
43     msg_ = msg_.replace(constants.MERGE_KW,
constants.WHITE_SPACE)
44     msg_ = msg_.replace(constants.DFLT_KW,
constants.WHITE_SPACE)
45
46     return msg_
47
48 def detectBuggyCommit(msg_, verboseFlag =
False):
49     flag2ret = False
50     index2ret = 0
51     msg_ = msg_.lower()
52
53     if (constants.IDEM_XTRA_kw in msg_) or
(constants.SYNTAX_XTRA_kw in msg_):
54         msg_ = doTempCleanUp(msg_)
55
56     if(any(x_ in msg_ for x_ in
constants.prem_bug_kw_list)) and ( constants.DFLT_kw not in
msg_) and (constants.MERGE_kw not in msg_):
57         str2see = [y_ for y_ in
constants.prem_bug_kw_list][0]
58         index2ret = msg_.find( str2see )
59         flag2ret = True
60
61     return flag2ret, index2ret
62
62 def detectRevertedCommit(msg_):
63     flag2ret = False
64     msg_ = msg_.lower()
65     revert_matches =
re.findall(constants.REVERT_REGEX, msg_)
66     if(len(revert_matches) > 0):
67         flag2ret = True
68
69     return flag2ret
70
70 def categ_check(key_words=[], msg=None,
diff_function=None, diff=None,
classification=constants.NO_DEFECT_CATEG):
71     if any(kw in msg for kw in key_words) or
(diff_function != None and diff_function(diff)):
72         return classification
73
74 ...
75 detectCateg takes a sentence and a diff from a
commit message as input , and return a defect category (single
value)
76 ...
77 def detectCateg(msg_, diff_,
verboseFlag=False,hash=None):
78     temp_msg_ = '' ## for oracle dataset
79     defect_categ_to_ret = set()
80
81     if (len(diff_) > 0):
82         temp_msg_list =
filterCommitMessage(msg_) # for extra false negative rules
83         temp_msg_ =
constants.WHITE_SPACE.join(temp_msg_list) # for extra false
negative rules
84         msg_ = doDepAnalysis(msg_)
## depending on results, this extra step of dependency parsing
may change
85
86         for classification, key_words,
diff_function in constants.CLASSIFICATION_PARSE:
87             categ_check_classification =
categ_check(msg=msg_, diff=diff_, key_words=key_words,
diff_function=diff_function, classification=classification)
88             if categ_check_classification !=
constants.NO_DEFECT_CATEG and categ_check_classification !=

```

```

constants.BUGGY_COMMIT:
    89
defect_categ_to_ret.add(categ_check_classification)
    90
    91      # These are the extra rules that the
author selected
    92
    93      # extra rule for idempotence
    94      if ( constants.IDEM_XTRA_KW in
temp_msg_ ) and ( constants.EXTRA_FIX_KEYWORD in temp_msg_ )
or any(_ in temp_msg_ for _ in constants.idem_defect_kw_list):
    95
defect_categ_to_ret.add(constants.IDEM_DEFECT_CATEG)
    96
    97      # extra rule for conditional
    98      if (( constants.LOGIC_XTRA_KW1 in
temp_msg_ ) or ( constants.LOGIC_XTRA_KW2 in temp_msg_ ) or (
constants.LOGIC_XTRA_KW3 in temp_msg_ ) ) and (
constants.EXTRA_FIX_KEYWORD in temp_msg_ ):
    99
defect_categ_to_ret.add(constants.CONDI_DEFECT_CATEG )
100
101      # extra rule for syntax
102      if any(kw in temp_msg_ for kw in
constants.syntax_xtra_kw_list) and (((
constants.EXTRA_FIX_KEYWORD in temp_msg_ ) )):
103
defect_categ_to_ret.add(constants.SYNTAX_DEFECT_CATEG)
104
105      # extra rule for doc
106      if ( constants.DOC_XTRA_KW in
temp_msg_ ) and ( constants.EXTRA_FIX_KEYWORD in temp_msg_ ):
107
defect_categ_to_ret.add(constants.DOC_DEFECT_CATEG)
108
109      if any(kw in temp_msg_ for kw in
constants.dep_xtra_kw_list) and (((
constants.EXTRA_FIX_KEYWORD in temp_msg_ ) )):
110
defect_categ_to_ret.add(constants.DEP_DEFECT_CATEG)
111
112      if any(kw in temp_msg_ for kw in
constants.resource_xtra_kw_list) and (((
constants.EXTRA_FIX_KEYWORD in temp_msg_ ) )):
113
defect_categ_to_ret.add(constants.SERVICE_RESOURCE_DEFECT_CATEG)
114
115      if any(kw in temp_msg_ for kw in
constants.network_extra_kw_list) and (((
constants.EXTRA_FIX_KEYWORD in temp_msg_ ) )):
116
defect_categ_to_ret.add(constants.NETWORK_DEFECT_CATEG)
117
118      if any(kw in temp_msg_ for kw in
constants.storage_extra_kw_list) and (((
constants.EXTRA_FIX_KEYWORD in temp_msg_ ) )):
119
defect_categ_to_ret.add(constants.STORAGE_DEFECT_CATEG)
120
121      if any(kw in temp_msg_ for kw in
constants.credentials_extra_kw_list) and (((
constants.EXTRA_FIX_KEYWORD in temp_msg_ ) )):
122
defect_categ_to_ret.add(constants.CREDENTIALS_DEFECT_CATEG)
123
124      return list(defect_categ_to_ret)

```

Constants Code

```

1 import diff_parser
2
3 DATASET_DIR = 'dataset'
4 REPO_FILE_LIST = 'eligible_repos.csv'
5 MASTER_BRANCH = 'main'
6 FILE_READ_MODE = 'r'
7 AST_PATH = 'EXTRA_AST'
8 PP_EXTENSION = '.pp'
9 IAC_FILES = [
10     "Pulumi.yaml", "Pulumi.yml", "cdk.json",
"cdktf.json",
11     ".py", ".go", ".js", ".ts", ".java",
".tf",
12     ".cs", ".fs", ".vb", ".cpp", ".kt",
".php", ".rb", ".swift", ".abap", ".edn"
13 ]
14 DATE_TIME_FORMAT = "%Y-%m-%dT%H-%M-%S"
15 WHITE_SPACE = ' '
16 TAB = '\t'
17 NEWLINE = '\n'
18 HASH_SYMBOL = '#'
19 comments_elems = ['#', '//', '/*', '*'']
20 CSV_REPLICATION = '/home/aluno/ACID-
dataset/ARTIFACT/Iac_Defect_Categ_Revamp/replication/iac_time_period.csv'
21 CSV_DEFAULT_PATH = '/home/aluno/filtered-

```

```

repositories-iac-criteria/criteria4/
22
23 CHANGE_DIR_CMD = 'cd '
24 GIT_COMM_CMD_1 = "git show --name-status "
25 GIT_COMM_CMD_2 = " | awk
'/(Pulumi\\\.yaml|Pulumi\\\.yml|cdk\\\.json|cdktf\\\.json|\\.py|\\.go|\\.js|\\.ts|\\.java|\\.tf|\\.cs|\\.ts
{print $2}'"
26 BASH_CMD = 'bash'
27 BASH_FLAG = '-c'
28 GIT_SHOW_CMD = "git show"
29 GIT_DIFF_CMD = " git diff"
30 HG_REV_SPEC_CMD = " ; hg log -p -r "
31
32 ENCODING = 'utf8'
33 UTF_ENCODING = 'utf-8'
34 FILE_WRITE_MODE = 'w'
35 SPACY_ENG_DICT = 'en_core_web_sm'
36 ROOT_TOKEN = 'ROOT'
37
38 STR_LIST_BOUNDS = 3 # tri-grams
39 NO_DEFECT_CATEG = 'NO_DEFECT'
40 BUGGY_COMMIT = 'BUGGY_COMMIT'
41 prem_bug_kw_list = ['error', 'bug',
'fix', 'issu', 'mistake', 'incorrect', 'fault', 'defect',
'flaw', 'solve']
42
43 CONFIG_DEFECT_CATEG = 'CONFIG_DATA_DEFECT'
44 config_defect_kw_list = ['value',
'config', 'option', 'setting', 'hiera', 'data']
45
46 DEP_DEFECT_CATEG = 'DEP_DEFECT'
47 dep_defect_kw_list = ['requir',
'depend', 'relation', 'order', 'sync', 'compatibil', 'ensur',
'inherit']
48 dep_defect_kw_list += ['version',
'deprecat', 'packag', 'path', 'modul', 'upgrad', 'updat']
49 dep_xtra_kw_list = ['module']
50
51 # Retirado import por causa de port em network
52 # These are used on diff_parser
53 VAR_SIGN = '='
54 ATTR_SIGN = '>='
55 diff_depen_code_elems = ['>', '::',
'include', 'packag', 'exec', 'require', 'import', 'version']
56
57 DOC_DEFECT_CATEG = 'DOC_DEFECT'
58 # origal author doc_defect_kw_list =
['doc', 'comment', 'spec', 'license', 'copyright', 'notice',
'header', 'readme']
59 # changed made removing 'spec' and header
60 doc_defect_kw_list = ['doc', 'comment',
'licens', 'copyright', 'notic', 'readm']
61 doc_defect_kw_list += ['descript']
62
63 IDEM_DEFECT_CATEG = 'IDEM_DEFECT'
64 idem_defect_kw_list = ['idempot']
65 idem_defect_kw_list += ['determin']
66
67 diff_idem_code_elem = 'class'
68 diff_idem_removal_cnt = 10
69
70 CONDI_DEFECT_CATEG = 'CONDITIONAL_DEFECT'
71 logic_defect_kw_list = ['logic',
'condition', 'bool']
72
73 diff_logic_code_elems = ['if', 'unless',
'els', 'case']
74 diff_logic_code_elems+= ['while', 'elif']
75
76 SECU_DEFECT_CATEG =
'SECURITY_DEFECT'
77 secu_defect_kw_list = ['vulner',
'ssl', 'secri', 'authent', 'password', 'security', 'cve']
78 # secu_defect_kw_list += ['cert',
'firewall', 'encrypt', 'protect']
79 # adding access to control
80 secu_defect_kw_list += ['cert',
'firewall', 'encrypt', 'protect', 'access']
81
82 diff_secu_code_elems = ['tls', 'cert',
'cred', 'ssl', 'password', 'pass', 'pwd']
83
84 NETWORK_DEFECT_CATEG =
'CD_NETWORK_DEFECT'
85 # removing address that was from the original
author
86 # network_defect_kw_list = ['network',
'address', 'port', 'tcp', 'dhcp', 'ssh', 'gateway', 'connect']
87 network_defect_kw_list = ['network',
'port', 'tcp', 'dhcp', 'ssh', 'gateway', 'connect']
88 network_defect_kw_list += ['rout']
89 diff_network_elems = ['url', 'vpc',
'subnet', 'endpoint']
90 network_extra_kw_list = ['gateway']
91 # ip tem que sair por causa de descript em doc
92
93 STORAGE_DEFECT_CATEG =

```

```

'CD_STORAGE_DEFECT'
    94 storage_defect_kw_list      = ['sql', 'db',
'database', 'disk']
    95 storage_extra_kw_list      = ['disk']
    96 # retirar database por causa de data em
configuration
    97
    98 CACHE_DEFECT_CATEG      =
'CD_CACHE_DEFECT'
    99 cache_defect_kw_list      = ['cach',
'memory', 'buffer', 'evict', 'ttl']
   100
   101 CREDENTIALS_DEFECT_CATEG      =
'CD_CREDENTIAL_DEFECT'
   102 credentials_defect_kw_list  = ['polic',
'credential', 'iam', 'role', 'token', 'user', 'username',
'password']
   103 credentials_extra_kw_list  = ['polic']
   104 diff_credentials_kw_list  = ['polic',
'credential']
   105
   106 FILE_SYSTEM_DEFECT_CATEG      =
'CD_FILE_SYSTEM_DEFECT'
   107 file_system_defect_kw_list  = ['file',
'permisss']
   108
   109 SYNTAX_DEFECT_CATEG      = 'SYNTAX_DEFECT'
   110 syntax_defect_kw_list      = ['compil', 'lint',
'warn', 'typo', 'spell', 'indent', 'regex', 'duplicat',
'variabl', 'whitespac']
   111 syntax_defect_kw_list += ['type', 'format',
'naming', 'casing', 'styl', 'comma', 'pattern', 'quot']
   112 # retirar name por causa de username
   113
   114 SERVICE_RESOURCE_DEFECT_CATEG      =
'SERVICE_RESOURCE_DEFECT'
   115 resource_defect_kw_list      = ['servic',
'server', 'location', 'resourc', 'provi', 'cluster']
   116 resource_xtra_kw_list      = ['kube', 'cloud']
   117 diff_service_code_elems      = ['service']
   118
   119 SERVICE_PANIC_DEFECT_CATEG      =
'SERVICE_PANIC_DEFECT'
   120 panic_defect_kw_list      = ['check', 'deploy',
'reboot', 'build', 'mount', 'kernel', 'extran', 'bypass']
   121
   122 CLASSIFICATION_PARSE = [
   123     (CONDI_DEFECT_CATEG,
logic_defect_kw_list,
diff_parser.checkDiffForLogicDefects),
   124     (IDEM_DEFECT_CATEG,
idem_defect_kw_list,
None),
   125     (DOC_DEFECT_CATEG,
doc_defect_kw_list,
diff_parser.checkDiffForDocDefects),
   126     (SYNTAX_DEFECT_CATEG,
syntax_defect_kw_list,
None),
   127     (SECU_DEFECT_CATEG,
secu_defect_kw_list,
diff_parser.checkDiffForSecurityDefects),
   128     (DEP_DEFECT_CATEG,
dep_defect_kw_list,
diff_parser.checkDiffForDepDefects),
   129     (CONFIG_DEFECT_CATEG,
config_defect_kw_list,
diff_parser.checkDiffForConfigDefects),
   130     (NETWORK_DEFECT_CATEG,
network_defect_kw_list,
diff_parser.checkDiffForNetwork),
   131     (STORAGE_DEFECT_CATEG,
storage_defect_kw_list,
None),
   132     (CACHE_DEFECT_CATEG,
cache_defect_kw_list,
None),
   133     (FILE_SYSTEM_DEFECT_CATEG,
file_system_defect_kw_list,
None),
   134     (CREDENTIALS_DEFECT_CATEG,
credentials_defect_kw_list,
diff_parser.checkDiffForCredentials),
   135     (SERVICE_RESOURCE_DEFECT_CATEG,
resource_defect_kw_list,
diff_parser.checkDiffForServiceDefects),
   136     (SERVICE_PANIC_DEFECT_CATEG,
panic_defect_kw_list,
None)
   137 ]
   138
   139 EXTRA_FIX_KEYWORD      = 'fix'
   140 EXTRA_BUG_KEYWORD      = 'bug'
   141
   142 DFLT_KW      = 'default'
   143 CLOSE_KW      = 'closes-bug'
   144 MERGE_KW      = 'merge'
   145 REVERT_KW      = 'revert'
   146 REVERT_REGEX  = r'^revert.*\.*$'
   147
   148 IDEM_XTRA_KW      = 'idempot' # for
detectBuggyCommit()
   149
   150 LOGIC_XTRA_KW1      = 'condit'

```

```

151 LOGIC_XTRA_KW2 = 'logic'
152 LOGIC_XTRA_KW3 = 'bool'
153
154 SYNTAX_XTRA_KW1 = 'lint'
155 SYNTAX_XTRA_KW2 = 'typo'
156 SYNTAX_XTRA_KW4 = 'syntax'
157 syntax_xtra_kw_list = ['lint', 'typo',
'syntax', 'type']
158 DOC_XTRA_KW = 'notice'
159 # DEPEND_XTRA_KW = 'override'
160 # NETWORK_XTRA_KW = 'provis'
161
162 diff_config_code_elems = ['hiera', 'hash',
'parameter']
163
164 diff_extra_idem_elems = ['ensure', 'unless',
'creates', 'replace']
165
166 lev_cutoff = 75
167
168 ...
169 Oracle dataset work
170 ...
171 ORACLE_HASH_CHECKLIST =
['75e460ab929a76e9e4a8d42740a529b3a476e952',
172 '9a5a54073bf887f87886ae4f9f52d5ade1b26bc7',
173 '0d834093814b3d184eff36b2835530a847ee6421',
174 '854e0e7b9fc339dc56bf3e2b3de7107c3f35b835',
175 'a7dedf197a24bf8a3fad00d1d1f58eede2f43057',
176 '114536ef2e7c569300019844e0ca57d278e27791'
177 ]

```

Criteria Code

```

1 import os
2 import sys
3 import subprocess
4 import pandas as pd
5 from concurrent.futures import
ThreadPoolExecutor
6
7 iac_extensions = [".tf", "Pulumi.yaml",
"Pulumi.yml", "cdk.json", "cdktf.json", ".edn"]
8
9 def is_not_fork(repo_path):
10     config_file = os.path.join(repo_path,
".git", "config")
11     if not os.path.exists(config_file):
12         print(f"[WARNING] {repo_path}:
.git/config não encontrado.")
13         return None
14     with open(config_file, "r") as f:
15         is_fork = "fork = true" not in
f.read()
16         print(f"[INFO] {repo_path}: Fork?
{'Não' if is_fork else 'Sim'}")
17         return is_fork
18
19 def iac_percentage(repo_path):
20     total_files = 0
21     iac_files = 0
22     iac_directories = set()
23     for root, _, files in os.walk(repo_path):
24         has_iac = any(file_.endswith(ext) for
file_ in files for ext in iac_extensions)
25         if has_iac:
26             iac_directories.add(root)
27             total_files += len(files)
28             for iac_dir in iac_directories:
29                 iac_files += sum(len(files) for _, _,
files in os.walk(iac_dir))
30             percentage = (iac_files / total_files) *
100 if total_files > 0 else 0
31             print(f"[INFO] {repo_path}: IaC =
{iac_files}/{total_files} arquivos ({percentage:.2f}%)")
32             return percentage
33
34 def commits_per_month(repo_path):
35     result = subprocess.run(
36         ["git", "log", "--date=format:%Y-%m",
"--pretty=format:%ad"],
37         cwd=repo_path,
38         stdout=subprocess.PIPE,
39         text=True
40     )
41     dates = result.stdout.splitlines()
42     unique_months = set(dates)
43     cpm = len(dates) / len(unique_months) if
unique_months else 0
44     print(f"[INFO] {repo_path}: {len(dates)} commits em
{len(unique_months)} meses = {cpm:.2f}")

```

```

commits/mês")
45     return cpm
46
47 def num_contributors(repo_path):
48     result = subprocess.run(
49         ["git", "log", "--pretty=format:%ae"],
50         cwd=repo_path,
51         stdout=subprocess.PIPE,
52         text=True
53     )
54     all_emails =
set(result.stdout.splitlines())
55     filtered = {email for email in all_emails
if not email.endswith("@github.com")}
56     print(f"[INFO] {repo_path}:
{len(filtered)} contribuidores (sem bots/github)")
57     return len(filtered)
58
59 def analyze_repo(repo, dataset_dir, input_dir,
output_dir, filters):
60     repo_path = os.path.join(dataset_dir,
repo)
61     if not
os.path.isdir(os.path.join(repo_path, ".git")):
62         print(f"[WARNING] {repo_path} não é um
repositório Git válido. Ignorando.")
63         return {"repo": repo, "status": "Not a
Git repo"}
64
65     input_repos = os.listdir(input_dir) if
input_dir else []
66     is_input = repo in input_repos
67
68     results = {"repo": repo}
69     print(f"[INFO] Analisando repositório:
{repo}")
70
71     if filters["-fork"]:
72         results["is_not_fork"] =
is_not_fork(repo_path)
73         if filters["-iac-percentage"]:
74             results["iac_percentage"] =
iac_percentage(repo_path)
75             if filters["-commits-per-month"]:
76                 results["commits_per_month"] =
commits_per_month(repo_path)
77                 if filters["-num-contributors"]:
78                     results["num_contributors"] =
num_contributors(repo_path)
79
80             passed = True
81             if filters["-fork"] and not
results.get("is_not_fork", False):
82                 passed = False
83             if filters["-iac-percentage"] and
(results.get("iac_percentage") or 0) < 11:
84                 passed = False
85             if filters["-commits-per-month"] and
(results.get("commits_per_month") or 0) < 2:
86                 passed = False
87             if filters["-num-contributors"] and
(results.get("num_contributors") or 0) < 10:
88                 passed = False
89
90         results["passed"] = passed
91         print(f"[INFO] {repo}: {'PASSOU' if passed
else 'NÃO passou'} nos filtros.")
92
93         if passed and is_input:
94             target_path = os.path.join(output_dir,
repo)
95             try:
96                 if not
os.path.exists(target_path):
97                     os.symlink(os.path.abspath(repo_path),
target_path,
target_is_directory=True)
98                     results["link_created"] = True
99                     print(f"[INFO] Link simbólico
criado para {repo}")
100                else:
101                    results["link_created"] =
False
102
103            except Exception as e:
104                results["link_created"] = False
105                results["error"] = str(e)
106                print(f"[ERROR] Falha ao criar
link para {repo}: {e}")
107
108        return results
109        if __name__ == "__main__":
110            if "-dataset" not in sys.argv or "--
output" not in sys.argv:
111                print("Usage: python3 criterias.py --
dataset path --input path --output path [--fork] [--iac-

```

```

percentage] [--commits-per-month] [--num-contributors] [--csv
path/to/file.csv")
    112         sys.exit(1)
    113
    114     dataset_dir =
os.path.abspath(sys.argv[sys.argv.index("--dataset") + 1])
    115     output_dir =
os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
    116     input_dir =
os.path.abspath(sys.argv[sys.argv.index("--input") + 1]) if --
" in sys.argv else None
    117     os.makedirs(output_dir, exist_ok=True)
    118
    119     # Define caminho do CSV
    120     if "--csv" in sys.argv:
    121         csv_path =
os.path.abspath(sys.argv[sys.argv.index("--csv") + 1])
    122         else:
    123             csv_path = os.path.join(output_dir,
"criterias_results.csv")
    124
    125     filters = {
    126         "--fork": "--fork" in sys.argv,
    127         "--iac-percentage": "--iac-percentage"
in sys.argv,
    128         "--commits-per-month": "--commits-per-
month" in sys.argv,
    129         "--num-contributors": "--num-
contributors" in sys.argv
    130     }
    131
    132     repos = os.listdir(dataset_dir)
    133     print(f"[INFO] Iniciando análise de
{len(repos)} repositórios...")
    134
    135     results = []
    136     with ThreadPoolExecutor() as executor:
    137         futures =
[executor.submit(analyze_repo, repo, dataset_dir, input_dir,
output_dir, filters) for repo in repos]
    138         for i, future in enumerate(futures,
1):
    139             result = future.result()
    140             results.append(result)
    141             print(f"[INFO] {i}/{len(repos)}"
repositórios processados")
    142
    143     # Salva CSV
    144     fieldnames = set()
    145     new_df = pd.DataFrame(results)
    146
    147     if os.path.exists(csv_path):
    148         print(f"[INFO] CSV já existe,
atualizando resultados: {csv_path}")
    149
    150         # Carrega CSV existente
    151         existing_df = pd.read_csv(csv_path)
    152         # Junta com base na coluna "repo",
mantendo dados anteriores e atualizando os novos
    153         existing_df["repo"] =
existing_df["repo"].astype(str)
    154         new_df["repo"] =
new_df["repo"].astype(str)
    155         merged_df = pd.merge(existing_df,
new_df, on="repo", how="outer", suffixes=('', '_new'))
    156
    157         # Atualiza os campos com os dados
novos (colunas *_new), se existirem
    158         for col in new_df.columns:
    159             if col != "repo":
    160                 new_col = col + "_new"
    161                 if new_col in
merged_df.columns:
    162                     merged_df[col] =
merged_df[new_col].combine_first(merged_df[col])
    163                     merged_df.drop(columns=
[new_col], inplace=True)
    164
    165         # Salva de volta no CSV
    166         merged_df.to_csv(csv_path,
index=False)
    167         print(f"[INFO] Resultados atualizados
no CSV.")
    168     else:
    169         print(f"[INFO] Criando novo CSV em:
{csv_path}")
    170         new_df.to_csv(csv_path, index=False)
    171         print(f"[INFO] {len(results)}"
repositórios registrados no novo CSV.")

```

Criteria Frequency Code

```

1 import os
2 import csv
3 from concurrent.futures import
ThreadPoolExecutor

```

```

4   import sys
5
6
7   def
classify_technology_in_directory(repo_path):
8       """
9           Classify the technology based on files in
the repository directory.
10          Priority: Pulumi > Terraform > AWS CDK
11          """
12          for root, _, files in os.walk(repo_path):
13              for f in files:
14                  if f.endswith("Pulumi.yaml") or
f.endswith("Pulumi.yml"):
15                      return "Pulumi"
16                  elif f.endswith("cdktf.json") or
f.endswith(".tf"):
17                      return "Terraform"
18                  elif f.endswith("cdk.json"):
19                      return "AWS CDK"
20                  elif f.endswith(".edn"):
21                      return "NUBANK"
22          return "NOTFOUND"
23
24
25      def process_criteria(criteria_dir,
output_dir):
26          """
27              Process a single criteria directory to
classify technologies.
28          """
29          technology_counts = {"Pulumi": 0,
"Terraform": 0, "AWS CDK": 0, "NUBANK": 0, "NOTFOUND": 0}
30          results = []
31
32          # Iterate over the subdirectories in the
criteria directory
33          for repo_id in os.listdir(criteria_dir):
34              repo_path = os.path.join(criteria_dir,
repo_id)
35
36          # Only process if it's a directory
37          if os.path.isdir(repo_path):
38              tech_classification =
classify_technology_in_directory(repo_path)
39
technology_counts[tech_classification] += 1
40
results.append([repo_id,
tech_classification])
41
42          # Write results to a CSV
43          output_csv = os.path.join(output_dir, f"{os.path.basename(criteria_dir.rstrip('/'))}_output.csv")
44          os.makedirs(output_dir, exist_ok=True)
45          with open(output_csv, "w", newline="") as
file:
46              writer = csv.writer(file)
47              writer.writerow(["ID", "Technology"])
48              writer.writerows(results)
49
50          # Print counts
51          string = f"Counts for {criteria_dir}:\n"
52          for tech, count in
technology_counts.items():
53              string += f"{tech}: {count}\n"
54          print(string)
55
56          return output_csv
57
58
59      def
process_directories_in_parallel(criteria_dirs, output_dir):
60          """
61              Process all criteria directories in
parallel.
62          """
63          with ThreadPoolExecutor() as executor:
64              futures = [
65                  executor.submit(process_criteria,
criteria_dir, output_dir)
66                  for criteria_dir in criteria_dirs
67              ]
68              for future in futures:
69                  print(f"Output CSV generated:
{future.result()}")
70
71
72      if __name__ == "__main__":
73          if not '--input' in sys.argv or not '--
output' in sys.argv:
74              print("Usage: python3 criterias-
frequency.py --input path1,path2,path3,path4 --output path")
75              sys.exit(1)
76
77      criteria_dirs = [os.path.abspath(path) for
path in sys.argv[sys.argv.index('--input') + 1].split(',')]
78      print(f"Executing the frequency of the

```

```

following paths: {criteria_dirs})
79     output_dir =
os.path.abspath(sys.argv[sys.argv.index('--output') + 1])
80
81
process_directories_in_parallel(criteria_dirs, output_dir)

```

Related Files Finder Code

```

1  import os
2  import csv
3  import json
4  from concurrent.futures import
ThreadPoolExecutor
5  import sys
6
7  csv.field_size_limit(sys.maxsize)
8
9  # Definição dos arquivos e extensões
# permitidas por tecnologia
10 IAC_FILES = {
11     "Pulumi": {
12         "patterns": ["Pulumi.yaml",
"*.yml"],

13         "extensions": [".js", ".ts", ".py",
".go", ".cs", ".fs", ".vb", ".java"]
14     },
15     "Terraform": {
16         "patterns": ["cdktf.json", ".tf"],
17         "extensions": [".ts", ".py", ".java",
".cs", ".go"]
18     },
19     "AWS CDK": {
20         "patterns": ["cdk.json"],

21         "extensions": [".cpp", ".go", ".java",
".js", ".kt", ".cs", ".ts", ".php", ".py", ".rb", ".rs",
".swift", ".abap"]
22     },
23     "NUBANK": {
24         "patterns": [".edn"],

25         "extensions": [".edn"]
26     }
27 }
28
29 def process_directory(parent_dir,
subdir_name):
30     """Processa um diretório pai e identifica
os arquivos IaC e vizinhos compatíveis."""
31     subdir_path = os.path.join(parent_dir,
subdir_name)
32     iac_data = {
33         "id": subdir_name,
34         "iac_type": None,
35         "iac_paths": [],
36         "related_files": []
37     }
38
39     print(f"[DEBUG] Processando diretório:
{subdir_path}")
40
41     # Percorre arquivos no diretório pai
42     for dirpath, _, filenames in
os.walk(subdir_path):
43         for iac_type, details in
IAC_FILES.items():
44             found_iac_files = [
45                 f for f in filenames
46                 if any(f == pattern or
f.endswith(pattern) for pattern in details["patterns"])
47             ]
48             if found_iac_files:
49                 # Define o tipo de IaC
# identificado e adiciona seus arquivos
50                 iac_data["iac_type"] =
iac_type
51                 iac_data["iac_paths"].extend(
[os.path.join(dirpath, f)
for f in found_iac_files]
52             )
53
54             # Captura arquivos vizinhos
# compatíveis apenas com a tecnologia correspondente
55             allowed_extensions =
details["extensions"]
56             neighbor_files = [
57                 os.path.join(dirpath, f)
58                 for f in filenames
59                 if os.path.splitext(f)[1]
60             in allowed_extensions
61             ]
62
iac_data["related_files"].extend(neighbor_files)
63
64     print(f"[DEBUG] Resultados para ID
'{subdir_name}': {iac_data}")
65     return iac_data

```

```

66     66     def
67     67         find_iac_files_with_neighbors_parallel(root_dir,
68             MAX_THREADS=8):
69             """
70                 """Procura arquivos IaC e seus vizinhos
71                     usando paralelização."""
72             iac_results = []
73             parent_dirs = [d for d in
74                 os.listdir(root_dir) if os.path.isdir(os.path.join(root_dir,
75                     d))]
76             print(f"[DEBUG] Diretórios identificados:
77 {parent_dirs}")
78             with
79                 ThreadPoolExecutor(max_workers=MAX_THREADS) as executor:
80                     tasks =
81                     [executor.submit(process_directory, root_dir, subdir) for
82                         subdir in parent_dirs]
83                     for future in tasks:
84                         iac_results.append(future.result())
85                     print(f"[DEBUG] Total de diretórios
86 processados: {len(iac_results)}")
87             return iac_results
88
89     83     def save_to_csv(data, output_file):
90         """
91             """Salva os resultados em um arquivo
92                 CSV."""
93             print(f"[DEBUG] Salvando resultados no
94 arquivo: {output_file}")
95             with open(output_file, mode="w",
96                 newline="", encoding="utf-8") as csvfile:
97                 writer = csv.DictWriter(csvfile,
98                     fieldnames=["id", "iac_type", "iac_paths", "related_files"])
99                 writer.writeheader()
100                for entry in data:
101                    writer.writerow({
102                        "id": entry["id"],
103                        "iac_type": entry["iac_type"]
104                        if entry["iac_type"] else "None",
105                        "iac_paths": json.dumps(entry["iac_paths"]),
106                        "related_files": json.dumps(entry["related_files"])
107                    })
108                print(f"[DEBUG] Resultados salvos com
109 sucesso em {output_file}")
110
111    98    if __name__ == "__main__":
112        if "-input" not in sys.argv or "--output"
113            not in sys.argv:
114                print("Usage: python3 1-related-files-
115 generator.py --input path --output path -t number_threads")
116                sys.exit(1)
117
118    100    root_dir =
119        os.path.abspath(sys.argv[sys.argv.index("--input") + 1])
120        output =
121        os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
122
123    105    if "-t" in sys.argv:
124        n_threads =
125        int(sys.argv[sys.argv.index("-t") + 1])
126        iac_data =
127        find_iac_files_with_neighbors_parallel(root_dir, n_threads)
128        else:
129        iac_data =
130        find_iac_files_with_neighbors_parallel(root_dir)
131
132    112    save_to_csv(iac_data, output)

```

Commits Counter Code

```

1  import os
2  import subprocess
3  import csv
4  import sys
5  from concurrent.futures import
ThreadPoolExecutor, as_completed
6
7  csv.field_size_limit(sys.maxsize)
8
9  def is_git_repo(path):
10     return os.path.exists(os.path.join(path,
11 ".git"))
12
13  def count_commits_for_files(repo_path,
14     file_paths):
15      """
16          Conta o número de commits relacionados a
17          uma lista de arquivos em um repositório Git.
18      """
19      unique_commits = set()

```

```

    17     for file_path in file_paths:
    18         relative_path =
os.path.relpath(file_path, repo_path)
    19         cmd = ["git", "-C", repo_path, "log",
    20             "-pretty=%H", "--", relative_path]
    21         try:
    22             result = subprocess.run(cmd,
capture_output=True, text=True, check=True)
    23             commits =
result.stdout.strip().split("\n")
    24             unique_commits.update(commits)
    25         except subprocess.CalledProcessError:
    26             print(f"[ERROR] Falha ao executar
git log para arquivo {relative_path} em {repo_path}")
    27             continue
    28     return len(unique_commits)
    29
    30     def count_total_commits(repo_path):
    31         """
    32             Conta o total de commits do repositório.
    33         """
    34         cmd = ["git", "-C", repo_path, "log", "--"
pretty=%H"]
    35         try:
    36             result = subprocess.run(cmd,
capture_output=True, text=True, check=True)
    37             return
    38         except subprocess.CalledProcessError:
    39             print(f"[ERROR] Falha ao executar git
log no repositório {repo_path}")
    40             return None
    41
    42     def process_repository_row(row, dataset_dir):
    43         """
    44             Processa uma linha do CSV (um repositório)
e retorna a linha com os campos de commits preenchidos.
    45         """
    46         repo_id = row["id"]
    47         try:
    48             iac_paths = eval(row["iac_paths"])
            related_files =
eval(row["related_files"])
    49             except Exception as e:
    50                 print(f"[ERROR] Erro ao processar
paths do repositório {repo_id}: {e}")
    51             row["commit_count"] = ""
    52             row["total_commit_count"] = ""
    53             return row
    54
    55             iac_paths = [path for path in iac_paths if
path]
    56             related_files = [path for path in
related_files if path]
    57
    58             if not iac_paths and not related_files:
    59                 print(f"[INFO] Ignorando repositório
{repo_id} porque não há arquivos válidos.")
    60                 row["commit_count"] = ""
    61                 row["total_commit_count"] = ""
    62                 return row
    63
    64             repo_path = os.path.join(dataset_dir,
repo_id)
    65
    66             if not is_git_repo(repo_path):
    67                 print(f"[WARNING] Diretório
'{repo_path}' não é um repositório Git.")
    68                 row["commit_count"] = ""
    69                 row["total_commit_count"] = ""
    70                 return row
    71
    72             file_paths = iac_paths + related_files
            commit_count =
count_commits_for_files(repo_path, file_paths)
    74             total_commits =
count_total_commits(repo_path)
    75
    76             row["commit_count"] = commit_count
    77             row["total_commit_count"] = total_commits
    78             if total_commits is not None else ""
    79             return row
    80
    81     def
process_repositories_and_commits(input_csv, output_csv,
dataset_dir):
    82         """
    83             Processa o CSV de entrada de forma
concorrente, contando commits relacionados aos arquivos IaC
e salvando o resultado em um CSV de saída.
    84         """
    85         with open(input_csv, mode="r") as infile,
open(output_csv, mode="w", newline="") as outfile:
    86             reader = csv.DictReader(infile)
    87             fieldnames = reader.fieldnames +
["commit_count", "total_commit_count"]
    88             writer = csv.DictWriter(outfile,

```

```

fieldnames=fieldnames)
89         writer.writeheader()
90
91     rows = list(reader)
92
93     with ThreadPoolExecutor(max_workers=8)
94         futures =
95     {executor.submit(process_repository_row, row, dataset_dir):
96      row for row in rows}
97
98     for future in
99     as_completed(futures):
100        result_row = future.result()
101        writer.writerow(result_row)
102
103    # Entry point
104    if __name__ == "__main__":
105        if "-input" not in sys.argv or "--output"
106            not in sys.argv or "--dataset-dir" not in sys.argv:
107                print("Usage: python3 2-commits-
108 count.py --input path --output path --dataset-dir path")
109                sys.exit(1)
110
111    input_path =
112    os.path.abspath(sys.argv[sys.argv.index("--input") + 1])
113    output =
114    os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
115    dataset_dir =
116    os.path.abspath(sys.argv[sys.argv.index("--dataset-dir") + 1])
117
118 process_repositories_and_commits(input_path, output,
119 dataset_dir)

```

Time Period Code

```

1 import os
2 import subprocess
3 import csv
4 from datetime import datetime
5 import sys
6 from concurrent.futures import
ThreadPoolExecutor, as_completed
7
8 csv.field_size_limit(sys.maxsize)
9
10 def get_commit_time_period(repo_path,
file_paths):
11     if not
os.path.exists(os.path.join(repo_path, ".git")):
12         print(f"[WARNING] Diretório
'{repo_path}' não é um repositório Git. Ignorando.")
13         return None, None
14
15     commit_dates = []
16     for file_path in file_paths:
17         relative_path =
os.path.relpath(file_path, repo_path)
18         cmd = ["git", "-C", repo_path, "log",
"--pretty=%ci", "...", relative_path]
19         try:
20             result = subprocess.run(cmd,
capture_output=True, text=True, check=True)
21             dates =
result.stdout.strip().split("\n")
22             commit_dates.extend(dates)
23         except subprocess.CalledProcessError
as e:
24             print(f"[ERROR] git log falhou
para '{relative_path}' em '{repo_path}': {e}")
25             return None, None
26
27     valid_dates = []
28     for date in commit_dates:
29         try:
30             valid_dates.append(datetime.strptime(date, "%Y-%m-%d %H:%M:%S
%z"))
31         except ValueError as e:
32             print(f"[WARNING] Data inválida
'{date}' em '{repo_path}': {e}")
33             continue
34
35     if not valid_dates:
36         print(f"[WARNING] Nenhuma data válida
em '{repo_path}' .")
37         return None, None
38
39     return min(valid_dates), max(valid_dates)
40
41
42 def process_row(row, dataset_dir):
43     repo_id = row["id"]
44     iac_paths = eval(row["iac_paths"])
45     repo_path = os.path.join(dataset_dir,
repo_id)

```

```

46     try:
47         oldest_commit, newest_commit =
get_commit_time_period(repo_path, iac_paths)
48             if oldest_commit and newest_commit:
49                 commit_time_period =
(newest_commit - oldest_commit).days
50                     print(f"[INFO] {repo_id}: Período
de commits = {commit_time_period} dias")
51             else:
52                 commit_time_period = None
53                     print(f"[INFO] {repo_id}: Sem
dados de commit válidos.")
54
55             row["oldest_commit"] =
oldest_commit.strftime("%Y-%m-%d %H:%M:%S %z") if
oldest_commit else "N/A"
56             row["newest_commit"] =
newest_commit.strftime("%Y-%m-%d %H:%M:%S %z") if
newest_commit else "N/A"
57             row["commit_time_period"] =
commit_time_period
58
59     except Exception as e:
60         print(f"[ERROR] Erro ao processar
'{repo_id}': {e}")
61             row["oldest_commit"] = "N/A"
62             row["newest_commit"] = "N/A"
63             row["commit_time_period"] = "N/A"
64
65     return row
66
67
68
69
70 def process_time_period(input_csv, output_csv,
dataset_dir):
71     with open(input_csv, mode="r") as infile:
72         reader = csv.DictReader(infile)
73         fieldnames = reader.fieldnames +
["oldest_commit", "newest_commit", "commit_time_period"]
74         rows = list(reader)
75
76     print(f"[INFO] Iniciando processamento de
{len(rows)} repositórios...")
77
78     results = []
79     with ThreadPoolExecutor() as executor:
80         future_to_row =
{executor.submit(process_row, row, dataset_dir): row for row
in rows}
81         for i, future in
enumerate(as_completed(future_to_row), 1):
82             result = future.result()
83             results.append(result)
84             print(f"[INFO] Processados
{i}/{len(rows)} repositórios")
85
86         with open(output_csv, mode="w",
newline="") as outfile:
87             writer = csv.DictWriter(outfile,
fieldnames=fieldnames)
88             writer.writeheader()
89             writer.writerows(results)
90
91     print(f"[DONE] Todos os repositórios foram
processados com sucesso.")
92     print(f"[DONE] Resultados salvos em:
{output_csv}")
93
94
95     if __name__ == "__main__":
96         if not "--input" in sys.argv or not "--
output" in sys.argv or not "--dataset-dir" in sys.argv:
97             print("Usage: python3 3-time-period.py
--input path --output path --dataset-dir path")
98             sys.exit(1)
99
100    input_path =
os.path.abspath(sys.argv[sys.argv.index("--input") + 1])
101    output =
os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
102    dataset_dir =
os.path.abspath(sys.argv[sys.argv.index("--dataset-dir") + 1])
103    process_time_period(input_path, output,
dataset_dir)

```

Analyzer Code

```

1 import csv
2 from datetime import datetime
3 import os
4 import sys
5 import traceback
6
7 csv.field_size_limit(sys.maxsize)
8

```

```

9  def analyze_csv(file_path, output_csv):
10    results = {
11      "TOTAL": {"repos": 0, "commits": 0},
12      "Terraform": {"repos": 0, "commits": 0},
13      "Pulumi": {"repos": 0, "commits": 0},
14      "AWS CDK": {"repos": 0, "commits": 0},
15      "NUBANK": {"repos": 0, "commits": 0},
16    }
17
18    time_periods = {"TOTAL": [], "Terraform": [],
19      "Pulumi": [], "AWS CDK": [], "NUBANK": []}
20
21    with open(file_path, newline='') as csvfile:
22      reader = csv.DictReader(csvfile)
23      for i, row in enumerate(reader, 1):
24        try:
25          iac_paths =
26          related_files =
27          iac_files_count =
28          total_commits =
29          related_commits_count =
30          first_commit_date =
31          last_commit_date =
32          repo_name =
33          category =
34          datetime.strptime(row["iac_paths"].strip("[]").split(", "),
35            "%Y-%m-%d %H:%M:%S %z")
36          datetime.strptime(row["related_files"].strip("[]").split(", "),
37            "%Y-%m-%d %H:%M:%S %z")
38          int(row["total_commit_count"])
39          int(row["oldest_commit"])
40          int(row["newest_commit"])
41          if "iac_type" in row:
42            first_commit_date = row["iac_type"]
43            last_commit_date = row["iac_type"]
44            category = "TOTAL"
45            if "terraform" == row["iac_type"]:
46              category = "Terraform"
47            elif "pulumi" == row["iac_type"]:
48              category = "Pulumi"
49            elif "aws cdk" == row["iac_type"]:
50              category = "AWS CDK"
51            elif "nubank" == row["iac_type"]:
52              category = "NUBANK"
53            print(f"[INFO] Linha {i}:
54              # Atualizar métricas
55              results[category]["repos"] += 1
56              results[category]["commits"] += 1
57              results[category]["iac_files"] += 1
58              results["TOTAL"]["iac_files"] += 1
59              results["TOTAL"]["repos"] += 1
60              results["TOTAL"]["commits"] += 1
61              results["TOTAL"]["iac_files"] += 1
62              time_periods[category].append(first_commit_date)
63              time_periods[category].append(last_commit_date)
64              time_periods[category].append(first_commit_date)
65              time_periods[category].append(last_commit_date)
66              time_periods[category].append(first_commit_date)
67              time_periods[category].append(last_commit_date)
68              time_periods[category].append(first_commit_date)
69              time_periods[category].append(last_commit_date)
70              time_periods[category].append(first_commit_date)
71              time_periods[category].append(last_commit_date)
72              time_periods[category].append(first_commit_date)
73              time_periods[category].append(last_commit_date)
74              time_periods[category].append(first_commit_date)
75              time_periods[category].append(last_commit_date)
76              time_periods[category].append(first_commit_date)
77              time_periods[category].append(last_commit_date)
78              time_periods[category].append(first_commit_date)
79              time_periods[category].append(last_commit_date)
80              time_periods[category].append(first_commit_date)
81              time_periods[category].append(last_commit_date)
82              time_periods[category].append(first_commit_date)
83              time_periods[category].append(last_commit_date)
84              time_periods[category].append(first_commit_date)
85              time_periods[category].append(last_commit_date)
86              time_periods[category].append(first_commit_date)
87              time_periods[category].append(last_commit_date)
88              time_periods[category].append(first_commit_date)
89              time_periods[category].append(last_commit_date)
90              time_periods[category].append(first_commit_date)
91              time_periods[category].append(last_commit_date)
92              time_periods[category].append(first_commit_date)
93              time_periods[category].append(last_commit_date)
94              time_periods[category].append(first_commit_date)
95              time_periods[category].append(last_commit_date)
96              time_periods[category].append(first_commit_date)
97              time_periods[category].append(last_commit_date)
98              time_periods[category].append(first_commit_date)
99              time_periods[category].append(last_commit_date)
100             except Exception as e:
101               print(f"[ERROR] Erro na linha {i}: {e}")
102               traceback.print_exc()
103               continue
104             # Calcular o período de tempo para cada categoria
105             for category, times in time_periods.items():
106               if times:
107                 results[category]["time_period"] =

```

```

f"{min(times).strftime('%Y-%m-%d')} - 
{max(times).strftime('%Y-%m-%d')}"
    71      # Escrever resultados no CSV
    72      with open(output_csv, mode='w',
newline='', encoding='utf-8') as csvfile:
    73          fieldnames = ["category", "type",
"repos", "total_commits", "iac_files", "iac_commits",
"time_period"]
    74          writer = csv.DictWriter(csvfile,
fieldnames=fieldnames)
    75
    76          writer.writeheader()
    77          for category, data in results.items():
    78              writer.writerow({
    79                  "category": category,
    80                  "type": "GIT" if category !=
    81 "TOTAL" else "",
    82                  "repos": data["repos"],
    83                  "total_commits": data["commits"],
    84                  "iac_files": data["iac_files"],
    85                  "iac_commits": data["iac_commits"],
    86                  "time_period": data["time_period"] if data["time_period"] else "N/A"
    87              })
    88
    89          if __name__ == "__main__":
    90              if "--input" not in sys.argv or "--output" not in sys.argv:
    91                  print("Usage: python3 4-analyze.py --
input path --output path")
    92                  sys.exit(1)
    93
    94          input_csv =
os.path.abspath(sys.argv[sys.argv.index("--input") + 1])
    95          output_csv =
os.path.abspath(sys.argv[sys.argv.index("--output") + 1])
    96
    97          analyze_csv(input_csv, output_csv)
    98
    99          print(f"Resultados armazenados em:
{output_csv}")

```

Script Shell

Repository Cloner Code

```

1 #!/bin/bash
2
3 # Initial configurations
4 default_dir="dataset"
5 log_file="clone_logs.csv"
6 repos_file="repos_list.txt" # File with the
list of repositories
7 threads=10 # Number of threads
8 start_line=2 # Default starting line
9
10 function usage() {
11     echo "Usage: $0 [-d directory] [-f
repos_file] [-s start_line] [-t threads] [-c credential]"
12     echo "Options:"
13     echo " -d Destination directory for
cloning repositories (default: dataset)"
14     echo " -f File containing the list of
repositories (one per line) (default: repos.list.txt)"
15     echo " -s Starting line to continue
cloning (default: 2)"
16     echo " -t Number of simultaneous clones
(default: 10)"
17     echo " -c Credential type: ssh or token"
18     exit 1
19 }
20
21 # Process arguments
22 while getopts "d:f:s:t:c:" opt; do
23     case "$opt" in
24         d) target_dir="$OPTARG" ;;
25         f) repos_file="$OPTARG" ;;
26         s) start_line="$OPTARG" ;;
27         t) threads="$OPTARG" ;;
28         c) credential_type="$OPTARG" ;;
29         *) usage ;;
30     esac
31 done
32
33 [[ -z "$target_dir" ]] &&
target_dir="$default_dir"
34 [[ -z "$credential_type" ]] &&
credential_type="ssh"
35
36 # Initial validations

```

```

      37  if [[ ! -f "$repos_file" ]]; then
      38    echo "Error: Repositories file ($repos_file)
not found."
      39    exit 1
      40  fi
      41
      42  if [[ "$credential_type" != "ssh" &&
"$credential_type" != "token" ]]; then
      43    echo "Error: Invalid credential type. Use
'ssh' or 'token'."
      44    exit 1
      45  fi
      46
      47  # The following command can prevent
parallelism errors
      48  # It increases the limit of open files
      49  ulimit -n 4096
      50
      51  mkdir -p "$target_dir"
      52  echo "Destination directory created at
($target_dir)"
      53
      54  if [[ ! -f "$log_file" ]]; then
      55    echo "Repository,Status,Message" >
"$log_file"
      56  fi
      57  echo "Log file created at ($log_file)"
      58
      59  clone_repo() {
      60    local repo_url="$1"
      61    local repo_name=$(basename -s .git
"$repo_url")
      62
      63    if [[ "$credential_type" == "token" &&
"$repo_url" == https://* ]]; then
      64      repo_url=$(echo "$repo_url" | sed
"s|https://|https://$github_token@|")
      65    fi
      66
      67    echo "Cloning $repo_name..."
      68
      69    if git clone "$repo_url"
"$target_dir/$repo_name" &>/dev/null; then
      70      echo "$repo_url,Success," >> "$log_file"
      71      echo "[OK] $repo_name cloned
successfully."
      72    else
      73      echo "$repo_url,Error,Failed to clone" >>
"$log_file"
      74      echo "[ERROR] Failed to clone $repo_name."
      75    fi
      76  }
      77
      78  export -f clone_repo
      79  export credential_type
      80  github_token=""
      81
      82  if [[ "$credential_type" == "token" ]]; then
      83    echo "Enter the GitHub token: "
      84    read -s github_token
      85    export github_token
      86  fi
      87  export target_dir
      88  export log_file
      89
      90  # Process repositories in parallel
      91  repos_to_clone=$(tail -n +$start_line
"$repos_file")
      92  echo "$repos_to_clone" | xargs -P $threads -n
1 -I {} bash -c 'clone_repo "$1"' _ {}
      93
      94  mkdir -p csv
      95  mv '$log_file' "csv/$log_file"
      96
      97  find "$target_dir" -type d -exec chmod +x {}
\\;
      98
      99  echo "Process completed. Logs saved in
csv/$log_file."

```

Criteria Applier Code

```

      1  #!/bin/bash
      2
      3  log_file=" criterias.log"
      4  default_dir="dataset"
      5  python_cmd="python3"
      6
      7  function usage() {
      8    echo "Usage: $0 [-d directory] [-p
python_interpreter]"
      9    echo "Options:"
     10   echo " -d Specify the directory of the
repositories cloned (default: dataset)"
     11   echo " -p Specify the Python interpreter
(default: python3)"

```

```

12   exit 1
13 }
14
15 while getopts "d:p:" opt; do
16   case "$opt" in
17     d) target_dir="$OPTARG" ;;
18     p) python_cmd="$OPTARG" ;;
19     *) usage ;;
20   esac
21 done
22
23 if [[ -z "$target_dir" ]]; then
24   target_dir="$default_dir"
25 fi
26
27 echo "Execution of the apply criterias script
started" > $log_file
28
29 function handle_error() {
30   echo "[ERROR] $1. Exiting." | tee -a
$log_file
31   exit 1
32 }
33
34 mkdir -p csv/criterias-output || handle_error
"Failed to create CSV directories"
35
36 echo "Creating criterias directories..." | tee
-a $log_file
37 mkdir -p criterias/criterial
criterias/criteria2 criterias/criteria3 criterias/criteria4 || handle_error "Failed to create criterias directories"
38
39 echo "Running the first filtering process..."
| tee -a $log_file
40 $python_cmd replication/criterias.py --dataset
$target_dir --input $target_dir --output criterias/criterial -
-iac-percentage --csv csv/criterias-
output/criterias_results.csv 2>&1 | tee -a $log_file ||
handle_error "First filtering process failed"
41
42 echo "Running the second filtering process..."
| tee -a $log_file
43 $python_cmd replication/criterias.py --dataset
$target_dir --input criterias/criterial --output
criterias/criteria2 --fork --csv csv/criterias-
output/criterias_results.csv 2>&1 | tee -a $log_file ||
handle_error "Second filtering process failed"
44
45 echo "Running the third filtering process..."
| tee -a $log_file
46 $python_cmd replication/criterias.py --dataset
$target_dir --input criterias/criteria2 --output
criterias/criteria3 --commits-per-month --csv csv/criterias-
output/criterias_results.csv 2>&1 | tee -a $log_file ||
handle_error "Third filtering process failed"
47
48 echo "Running the fourth filtering process..."
| tee -a $log_file
49 $python_cmd replication/criterias.py --dataset
$target_dir --input criterias/criteria3 --output
criterias/criteria4 --num-contributors --csv csv/criterias-
output/criterias_results.csv 2>&1 | tee -a $log_file ||
handle_error "Fourth filtering process failed"
50
51 echo "Creating CSV directories..." | tee -a
$log_file
52 mkdir -p csv/criterias-output/criterias-
frequency || handle_error "Failed to create CSV directories"
53
54 echo "Generating the CSV with related
files..." | tee -a $log_file
55 $python_cmd replication/1-related-files-
generator.py --input $target_dir --output csv/criterias-
output/csv1_files_with_neighbors.csv 2>&1 | tee -a $log_file ||
handle_error "CSV related files generation failed"
56
57 echo "Generating the CSV with the commits
summary..." | tee -a $log_file
58 $python_cmd replication/2-commits-count.py --
input csv/criterias-output/csv1_files_with_neighbors.csv --
output csv/criterias-output/csv2_iac_commits_summary.csv --
dataset-dir $target_dir 2>&1 | tee -a $log_file ||
handle_error "Commits summary CSV generation failed"
59
60 echo "Generating the CSV with the time
period..." | tee -a $log_file
61 $python_cmd replication/3-time-period.py --
input csv/criterias-output/csv2_iac_commits_summary.csv --
output csv/criterias-output/csv3_iac_criterias_output.csv --
dataset-dir $target_dir 2>&1 | tee -a $log_file ||
handle_error "Time period CSV generation failed"
62
63 echo "Generating the CSV with frequency..." |
tee -a $log_file
64 $python_cmd replication/4-analyze.py --input
csv/criterias-output/csv3_iac_criterias_output.csv --output

```

```

csv/criterias-output/csv4_iac_output_frequency.csv 2>&1 | tee
-a $log_file || handle_error "Frequency CSV generation failed"
    65
    66 echo "Generating criterias frequency csv..." |
tee -a $log_file
    67 $python_cmd replication/criteria-frequency.py
--input
criterias/criterial,criterias/criteria2,criterias/criteria3,criterias/criteria4,$target_dir
--output csv/criterias-output/criterias-frequency 2>&1 | tee
-a $log_file || handle_error "Criterias frequency CSV
generation failed"
    68
    69 echo "Criterias execution completed. Logs
saved to $log_file." | tee -a $log_file

```

ACID Runner Code

```

1 #!/bin/bash
2
3 log_file="run_acid.log"
4 source_dir="dataset"
5 flag_arg="REPLICATION"
6 target_dir="ACID/dataset/$flag_arg"
7 output_dir="csv/acid-output"
8 script_to_run="ACID/main.py"
9 python_cmd="python3"
10
11 function usage() {
12   echo "Usage: $0 [-c] [-p
<python_interpreter>]"
13   echo "Options:"
14   echo " -c      Use 'main-concurrent.py'
instead of 'main.py'"
15   echo " -p      Specify the Python interpreter
(default is 'python3')"
16   exit 1
17 }
18
19 echo "Starting run-acid.sh" | tee -a
"$log_file"
20
21 while getopts "cp:" opt; do
22   case "$opt" in
23     c)
24       script_to_run="ACID/main-concurrent.py"
25       echo "[INFO] Using concurrent script:
$script_to_run" | tee -a "$log_file"
26       ;;
27     p)
28       python_cmd="$OPTARG"
29       echo "[INFO] Using Python interpreter:
$python_cmd" | tee -a "$log_file"
30       ;;
31     *)
32       usage
33       ;;
34   esac
35 done
36
37 echo "[INFO] Creating directories..." | tee -a
"$log_file"
38 mkdir -p "ACID/dataset" | tee -a "$log_file"
39 mkdir -p "$target_dir" | tee -a "$log_file"
40 mkdir -p "$output_dir" | tee -a "$log_file"
41
42 echo "[INFO] Creating symbolic links from
$source_dir to $target_dir..." | tee -a "$log_file"
43 for dir in "$source_dir"/*; do
44   if [[ -d "$dir" ]]; then
45     dir_abs_path=$(realpath "$dir")
46     dir_name=$(basename "$dir")
47
48     ln -s "$dir_abs_path"
"$target_dir/$dir_name" 2>>"$log_file"
49
50   echo "[INFO] Link created:
$target_dir/$dir_name -> $dir_abs_path" | tee -a "$log_file"
51 fi
52 done
53
54 echo "[INFO] Generating eligible repositories
CSV at $target_dir/eligible_repos.csv..." | tee -a "$log_file"
55 ls '$target_dir' >
"$target_dir/eligible_repos.csv" 2>>"$log_file"
56 if [[ $? -eq 0 ]]; then
57   echo "[SUCCESS] CSV generated successfully."
| tee -a "$log_file"
58 else
59   echo "[ERROR] Failed to generate CSV." | tee
-a "$log_file"
60   exit 1
61 fi
62
63 echo "[INFO] Running $script_to_run with
$python_cmd..." | tee -a "$log_file"
64 $python_cmd "$script_to_run" --flag-arg

```

```
$flag_arg --csv-replication csv/criterias-
output/csv3_iac_criterias_output.csv --csv-default
"$source_dir" --output "$output_dir" 2>>"$log_file"
65  if [[ $? -eq 0 ]]; then
66    echo "[SUCCESS] $script_to_run executed
successfully." | tee -a "$log_file"
67  else
68    echo "[ERROR] $script_to_run failed." | tee
-a "$log_file"
69  exit 1
70 fi
71
72 echo "[INFO] Script completed. Logs saved to
$log_file." | tee -a "$log_file"
```