Part 1.

Develop a simple recommendation system. The whole solution should consist of several services

1. Generator: The service that generates recommendations.
    1. Single POST method that takes two parameters: model name and viewerid. Each run of the service generates a random number and returns a result of the type: `{“reason”: **<MODELNAME>**, “result”: **<RANDOMNUMBER>**}`
2. Invoker: The service that returns recommendations.
    1. Method `recommend()` that checks the cache for the given user. If there is no data in the cache, it calls the `runcascade()` function. The merged result of the cascade runs is saved in the cache and returned in the response;
    2. Method `runcascade()` makes 5 parallel calls to the **`GENERATOR`** service with different **`<MODELNAME>`** parameters;

Implement two levels of caching: local with a TTL of 10 seconds (limited to 3 keys) and Redis. If there is no data for the given user in the local cache, take the cache from Redis.

All the services have to be packed as docker containers. All the third-party services you need also have to be packed into docker image. Whole solution has to be deployed with docker compose.

Documentation has to be provided in the Readme file in the git repo.

Part 2.

Fill the gaps in the code.

Python

```
#############
#
# Cheap Crowdfunding Problem
#
# There is a crowdfunding project that you want to support. This project
# gives the same reward to every supporter, with one peculiar condition:
# the amount you pledge must not be equal to any earlier pledge amount.
#
# You would like to get the reward, while spending the least amount > 0.
#
# You are given a list of amounts pledged so far in an array of integers.
# You know that there is less than 100,000 of pledges and the maximum
# amount pledged is less than $1,000,000.
#
# Implement a function find_min_pledge(pledge_list) that will return
# the amount you should pledge.
#
#############

def find_min_pledge(pledge_list):
    pass

assert find_min_pledge([1, 3, 6, 4, 1, 2]) == 5

assert find_min_pledge([1, 2, 3]) == 4

assert find_min_pledge([-1, -3]) == 1
```

Python

```
############
#
# Extract Titles from RSS feed
#
# Implement get_headlines() function. It should take a url of an RSS feed
# and return a list of strings representing article titles.
#
############

google_news_url="https://news.google.com/news/rss"


def get_headlines(rss_url):
    """
    @returns a list of titles from the rss feed located at `rss_url`
    """
    return []

print(get_headlines(google_news_url))
```

Python

```
############
#
# Streaming Payments Processor
#
# The function `process_payments()` is processing a large, but finite
# amount of payments in a streaming fashion.
#
# It relies on two library functions to do its job. The first function
# `stream_payments_to_storage(storage)` reads the payments from a payment
# processor and writes them to storage by calling `storage.write(buffer)`
# on it's `storage` argument. The `storage` argument is supplied by
```

```
# calling `get_payments_storage()` function. The API is defined below.
#
# TODO: Modify `process_payments()` to print a checksum of bytes written
# by `stream_payments_to_storage()`. The existing functionality
# should be preserved.
#
# The checksum is implemented as a simple arithmetic sum of bytes.
#
# For example, if bytes([1, 2, 3]) were written, you should print 6.
#
#
# NOTE: you need to take into account the following restrictions:
# - You are allowed only one call each to `get_payments_storage()` and
#       to `stream_payments_to_storage()`
# - You can not read from the storage.
# - You can not use disk as temporary storage.
# - Your system has limited memory that can not hold all payments.
#
#############


# This is a library function, you can't modify it.
def get_payments_storage():
    """
    @returns an instance of
    https://docs.python.org/3/library/io.html#io.BufferedWriter
    """
    # Sample implementation to make the code run in coderpad.
    # Do not rely on this exact implementation.
    return open('/dev/null', 'wb')


# This is a library function, you can't modify it.
def stream_payments_to_storage(storage):
    """
    Loads payments and writes them to the `storage`.
```

```
    Returns when all payments have been written.

    @parameter `storage`: is an instance of
    https://docs.python.org/3/library/io.html#io.BufferedWriter
    """
    # Sample implementation to make the code run in coderpad.
    # Do not rely on this exact implementation.
    for i in range(10):
        storage.write(bytes([1, 2, 3, 4, 5]))


def process_payments():
    """
    TODO:
    Modify `process_payments()` to print the checksum of data
    generated by the `stream_payments_to_storage()` call.
    """
    stream_payments_to_storage(get_payments_storage())
    # Here print the check sum of all of the bytes written by
    # `stream_payments_to_storage()`

process_payments()
```

Python

```
#############
# Streaming Payments Processor, two vendors edition.
#
# We decided to improve the payment processor from the previous
# exercise and hired two vendors. One was to implement `stream_payments()`
# function, and another `store_payments()` function.
#
# The function `process_payments_2()` is processing a large, but finite
```

```
# amount of payments in a streaming fashion.
#
# Unfortunately the vendors did not coordinate their efforts, and delivered
# their functions with incompatible APIs.
#
# TODO: Your task is to analyse the APIs of `stream_payments()` and
# `store_payments()` and to write glue code in `process_payments_2()`
# that allows us to store the payments using these vendor functions.
#
# NOTE: you need to take into account the following restrictions:
# - You are allowed only one call each to `stream_payments()` and
#      to `store_payments()`
# - You can not read from the storage.
# - You can not use disk as temporary storage.
# - Your system has limited memory that can not hold all payments.
#
#############

import io


# This is a library function, you can't modify it.
def stream_payments(callback_fn):
    """
    Reads payments from a payment processor and calls `callback_fn(amount)`
    for each payment.

    Returns when there is no more payments.
    """
    # Sample implementation to make the code run in coderpad.
    # Do not rely on this exact implementation.
    for i in range(10):
        callback_fn(i)


# This is a library function, you can't modify it.
```

```python
def store_payments(amount_iterator):
    """
    Iterates over the payment amounts from amount_iterator
    and stores them to a remote system.
    """
    # Sample implementation to make the code run in coderpad.
    # Do not rely on this exact implementation.
    for i in amount_iterator:
        print(i)


def callback_example(amount):
    print(amount)
    return True


def process_payments_2():
    """
    TODO:
    Modify `process_payments_2()`, write glue code that enables
    `store_payments()` to consume payments produced by `stream_payments()`
    """
    stream_payments(callback_example)


process_payments_2()
```

Python

```python
#############
#
# Code Review
#
# Please do a code review for the following snippet.
```

```
# Add your review suggestions inline as python comments
#
#############


def get_value(data, key, default, lookup=None, mapper=None):
    """
    Finds the value from data associated with key, or default if the
    key isn't present.
    If a lookup enum is provided, this value is then transformed to its
    enum value.
    If a mapper function is provided, this value is then transformed
    by applying mapper to it.
    """
    return_value = data[key]
    if return_value is None or return_value == "":
        return_value = default
    if lookup:
        return_value = lookup[return_value]
    if mapper:
        return_value = mapper(return_value)
    return return_value


def ftp_file_prefix(namespace):
    """
    Given a namespace string with dot-separated tokens, returns the
    string with
    the final token replaced by 'ftp'.
    Example: a.b.c => a.b.ftp
    """
    return ".".join(namespace.split(".")[:-1]) + '.ftp'


def string_to_bool(string):
    """
```

```
    Returns True if the given string is 'true' case-insensitive,
    False if it is
     'false' case-insensitive.
    Raises ValueError for any other input.
    """
    if string.lower() == 'true':
        return True
    if string.lower() == 'false':
        return False
    raise ValueError(f'String {string} is neither true nor false')


def config_from_dict(dict):
    """
    Given a dict representing a row from a namespaces csv file,
    returns a DAG configuration as a pair whose first element is the
    DAG name
    and whose second element is a dict describing the DAG's properties
    """
    namespace = dict['Namespace']
    return (dict['Airflow DAG'],
            {"earliest_available_delta_days": 0,
             "lif_encoding": 'json',
             "earliest_available_time":
                 get_value(dict, 'Available Start Time', '07:00'),
             "latest_available_time":
                 get_value(dict, 'Available End Time', '08:00'),
             "require_schema_match":
                 get_value(dict, 'Requires Schema Match', 'True',
                           mapper=string_to_bool),
             "schedule_interval":
                 get_value(dict, 'Schedule', '1 7 * * * '),
             "delta_days":
                 get_value(dict, 'Delta Days', 'DAY_BEFORE',
                           lookup=DeltaDays),
             "ftp_file_wildcard":
```

```
            get_value(dict, 'File Naming Pattern', None),
        "ftp_file_prefix":
            get_value(dict, 'FTP File Prefix',
                      ftp_file_prefix(namespace)),
        "namespace": namespace
        }
    )
```