📄 LLM Pipeline Summary

🧾 Description

Develop a billing calculator for cloud usage

✅ Subtasks

1. Project Initiation
2. Research and Analysis
3. System Design
4. Technology Stack Selection
5. Development Environment Setup
6. Frontend Development
7. Backend Development
8. Database Design and Implementation
9. Integration and Testing
10. Performance Optimization
11. Security Measures Implementation
12. Error Handling and Reporting
13. User Testing
14. Documentation
15. Quality Assurance (QA) Testing
16. Deployment Planning
17. Production Deployment
18. Maintenance and Support
19. Performance Monitoring and Logging
20. Scalability Planning
21. Disaster Recovery and Business Continuity Planning
22. Security Audit and Compliance
23. User Feedback Collection and Iteration
24. Marketing and Promotion
25. Analytics Integration
26. User Onboarding
27. Customer Support
28. Compliance with Data Privacy Laws
29. Continuous Improvement
30. Project Closure

💻 Developer Subtasks

1. Design the structure for CloudUsage data model (classes or structures)
2. Implement method to add new usage record (timestamp, resource type, and usage amount)
3. Create a method to retrieve total usage for a specific resource type
4. Implement a method to calculate daily billing for each resource type
5. Write a function to calculate monthly billing for each resource type
6. Develop a function to calculate yearly billing for each resource type
7. Create a method to calculate the total billing for all resources combined
8. Implement a method to display the detailed usage and billing report (by resource type and total)
9. Write a function to save the usage records in a persistent storage (file or database)
10. Develop a method to load usage records from persistent storage
11. Create a function to validate the input usage record before adding it to the storage
12. Implement a method for handling errors during data loading/saving process
13. Write a test case for adding new usage records
14. Test the function to retrieve total usage for a specific resource type
15. Develop and run tests for daily, monthly, and yearly billing calculations
16. Test the function to calculate the total billing for all resources combined
17. Create and implement a test case for detailed usage and billing report generation
18. Write a test case for saving/loading usage records from persistent storage
19. Test the validation of input usage record before adding it to the storage
20. Test error handling during data loading/saving process
21. Implement user interface for entering and viewing the usage records (if applicable)
22. Write code to connect to the database if using a relational database system
23. Create SQL statements for creating tables, inserting, updating, and querying data in the database (if applicable)
24. Test all the database-related operations with unit tests or integration tests
25. Develop a function to handle user input and call the appropriate business logic functions (if applicable)
26. Implement error handling for invalid user input (if applicable)
27. Write code for formatting and displaying the output in an easy-to-read format (if applicable)
28. Test the user interface functionality with a variety of test cases
29. Optimize the performance of resource usage calculations, especially for large datasets (if necessary)
30. Implement caching or lazy loading techniques to improve app performance (if applicable)

31. Add comments and documentation throughout the codebase to explain what each function does
32. Write unit tests for each function to ensure they are working as intended
33. Refactor the code to follow best practices and make it more maintainable
34. Review and test the refactored code thoroughly
35. Prepare release notes detailing changes made in the new version of the billing calculator
36. Create a build script or automate the build process using tools like Maven, Gradle, or others (if applicable)
37. Test the build script and ensure that it works as expected with different configurations
38. Deploy the billing calculator to a staging environment for testing before release
39. Test the deployed application in the staging environment and fix any issues found
40. Prepare and release the final version of the billing calculator to production.

📂 Code Files

`create_a_method_to_retrieve_total_usage_for_a_specific_resource`

```python
class ResourceUsageTracker:
    def __init__(self):
        self.usage = {}

    def add_usage(self, resource_type, usage_amount):
        if resource_type not in self.usage:
            self.usage[resource_type] = 0
        self.usage[resource_type] += usage_amount

    def get_total_usage(self, resource_type):
        if resource_type not in self.usage:
            return None
        return self.usage[resource_type]
```

This code defines a class `ResourceUsageTracker` that tracks the total usage of different resources. You can create an instance of this class and use the `add_usage()` method to add usage for a specific resource type, and `get_total_usage()` method to retrieve the total usage for a given resource type. If the resource type does not exist in the tracker, the `get_total_usage()` method will return None.

### `implement_error_handling_for_invalid_user_input_if_applicable_.py`

```python
```

~~~
```python
def get_valid_input(prompt):
    while True:
        try:
            user_input = input(prompt)
            if user_input.isnumeric():
                return int(user_input)
            else:
                print("Invalid input. Please enter a number.")
        except ValueError:
            print("Invalid input. Please enter a number.")
~~~

This function will keep prompting the user for an input until they provide a valid numeric input. It uses a try-except block to handle invalid inputs and provide an error message.

~~~
### `create_a_method_to_calculate_the_total_billing_for_all_resources_comb

```python
```python
class Resource:
    def __init__(self, cost):
        self.cost = cost

    def get_cost(self):
        return self.cost

class Bill:
    def __init__(self, resources):
        self.resources = resources

    def calculate_total(self):
        total = 0
        for resource in self.resources:
            total += resource.get_cost()
        return total
~~~

Example usage:

~~~
resource1 = Resource(10)
resource2 = Resource(5)
bill = Bill([resource1, resource2])
total = bill.calculate_total()
print(total)  # Outputs: 15


### `prepare_and_release_the_final_version_of_the_billing_calculator_to_pr

```python
```python
import sys
~~~

~~~~
from pathlib import Path
import setuptools

here = Path(__file__).parent

setuptools.setup(
    name='billing_calculator',
    version='1.0.0',
    description='Billing Calculator',
    author='Your Name',
    author_email='your.email@example.com',
    url='https://github.com/YourGithubUsername/billing_calculator',
    packages=setuptools.find_packages(),
    scripts=[str(here / 'scripts' / 'billing_calculator')],
    install_requires=[
        'numpy>=1.19.0',
        'pandas>=1.0.1',
        'pytz>=2018.9'
    ],
    entry_points={
        'console_scripts': [
            'billing_calculator = scripts.billing_calculator:main'
        ]
    },
    classifiers=[
        'Programming Language :: Python :: 3',
        'Operating System :: OS Independent',
    ],
)

# Prepare and release to production
subprocess.check_call([sys.executable, '-m', 'build', '--format=src'], cwd
subprocess.check_call([sys.executable, '-m', 'sdist', '--formats', 'gztar'
subprocess.check_call([sys.executable, '-m', 'upload', '--repository', 'py
~~~~

This code uses the `setuptools` package to prepare and distribute the billing calculator as a Python package on PyPI (Python Package Index). The script assumes that you have already set up your PyPI credentials for uploading packages. Adjust the author, email, url, install_requires, classifiers, and scripts fields accordingly.

~~~~
### `test_the_user_interface_functionality_with_a_variety_of_test_cases.py

```python
```python
import pytest
from your_ui_module import YourUIClass

@pytest.mark.parametrize("input, expected", [
    ("Test Input 1", "Expected Output 1"),
~~~~

````
    ("Test Input 2", "Expected Output 2"),
    # Add more test cases as needed
])
def test_your_ui_functionality(input, expected):
    ui = YourUIClass()
    result = ui.your_ui_function(input)
    assert result == expected

@pytest.mark.parametrize("button, should_be_enabled", [
    (None, False),
    (ui.Button1, True),
    # Add more test cases as needed
])
def test_button_states(button, should_be_enabled):
    ui = YourUIClass()
    actual = button.isEnabled() if button else None
    assert actual == should_be_enabled
````

### `deploy_the_billing_calculator_to_a_staging_environment_for_testing_be

~~~
```python
```python
import os
from fabric import Connection

def deploy_billing_calculator():
    repo_url = "https://github.com/your-organization/billing-calculator.gi
    staging_server = "staging.example.com"
    user = "deployuser"

    with Connection(host=staging_server, user=user) as conn:
        conn.run(f'mkdir -p ~/apps/{os.path.basename(repo_url)}')
        conn.run(f'cd ~/apps/{os.path.basename(repo_url)}; git clone {repo
        # Assuming you have a custom script to install and run the billing
        # Replace 'start-billing-calculator.sh' with your actual script pa
        conn.run(f'cd ~/apps/{os.path.basename(repo_url)} && ./start-billi

deploy_billing_calculator()
~~~

This script assumes you have Fabric installed and configured for the staging server, and a script named 'start-billing-calculator.sh' in the root directory of your billing calculator repository to handle installation and running the application.

### `develop_a_method_to_load_usage_records_from_persistent_storage.py`

~~~
```python
```python
import pickle
import os
~~~

~~~
def load_usage_records(filepath):
    if not os.path.exists(filepath):
        return []

    with open(filepath, 'rb') as f:
        usage_records = pickle.load(f)

    return usage_records


### `create_a_function_to_validate_the_input_usage_record_before_adding_it

```python
```python
def validate_usage(record):
    try:
        device_id = int(record['device_id'])
        timestamp = float(record['timestamp'])
        data_consumed = float(record['data_consumed'])

        if 0 <= device_id < sys.maxsize and timestamp >= 0 and data_consum
            return True
    except (KeyError, ValueError):
        return False
    return False
~~~

In this code, the validate_usage function takes a dictionary as an argument that contains 'device_id', 'timestamp' and 'data_consumed'. It checks if the values are integers for device_id and floats for timestamp and data_consumed. Additionally, it verifies that the values meet certain conditions (non-negative for device_id and data_consumed > 0). If all checks pass, it returns True indicating valid input; otherwise False.

~~~
### `develop_and_run_tests_for_daily_monthly_and_yearly_billing_calculatio

```python
```python
import unittest
from datetime import timedelta, date

class BillingTest(unittest.TestCase):
    def setUp(self):
        self.billing_rate = 10.0
        self.base_day_duration = timedelta(hours=24)

    def daily_billing(self, start_date, end_date):
        days = (end_date - start_date).days + 1
        return days * self.billing_rate

    def monthly_billing(self, start_date, end_date):
~~~

~~~
        first_day = date(start_date.year, start_date.month, 1)
        last_day = end_date if (end_date.month == start_date.month and end
        days_in_month = (last_day - first_day).days + 1
        return self.daily_billing(first_day, last_day) * days_in_month * s

    def yearly_billing(self, start_date, end_date):
        start_year = start_date.year
        end_year = end_date.year if end_date > start_date else start_date.
        return sum([self.monthly_billing(date(start_year, month=i+1, day=1

    def test_daily_billing(self):
        start_date = date(2022, 1, 1)
        end_date = date(2022, 1, 5)
        expected = 5 * self.billing_rate
        self.assertEqual(self.daily_billing(start_date, end_date), expecte

    def test_monthly_billing(self):
        start_date = date(2022, 1, 1)
        end_date = date(2022, 2, 1)
        expected = (date(2022, 2, 1) - date(2022, 1, 1)).days * self.billi
        self.assertEqual(self.monthly_billing(start_date, end_date), expec

    def test_yearly_billing(self):
        start_date = date(2021, 10, 5)
        end_date = date(2022, 9, 30)
        expected = sum([self.monthly_billing(date(2021, month=i+1, day=1),
        self.assertEqual(self.yearly_billing(start_date, end_date), expect

if __name__ == '__main__':
    unittest.main()


### `develop_a_function_to_handle_user_input_and_call_the_appropriate_busi

```python
```python
def get_user_input():
    user_choice = input("Enter your choice (add, remove, view): ")
    if user_choice in ["add", "remove", "view"]:
        return user_choice
    else:
        print("Invalid choice. Please enter add, remove or view.")
        return get_user_input()

def business_logic(choice):
    if choice == "add":
        # Implement add logic here
    elif choice == "remove":
        # Implement remove logic here
    elif choice == "view":
        # Implement view logic here
~~~

### `write_code_for_formatting_and_displaying_the_output_in_an_easy_to_rea

~~~
```python
```python
def format_output(data):
    header = "{0:<20} {1:<10} {2:<10}\n".format("Name", "Age", "Score")
    lines = []
    for name, age, score in data:
        line = "{0:<20} {1:<10} {2:<10}\n".format(name, age, score)
        lines.append(line)
    return header + "".join(lines)
~~~

Assuming the input data is in a list of tuples:

```
data = [("Alice", 25, 80), ("Bob", 30, 90), ("Charlie", 19, 75)]
```

You can call the function like this:

```
print(format_output(data))
```

### `refactor_the_code_to_follow_best_practices_and_make_it_more_maintaina

~~~
```python
```python
import re
from typing import List, Tuple

def parse_line(line: str) -> Tuple[str, int]:
    pattern = r'^(\w+) (\d+)$'
    match = re.match(pattern, line)
    if not match:
        raise ValueError("Invalid format")
    return match.groups()

def process_file(filename: str) -> List[str]:
    with open(filename, 'r') as file:
        lines = [line.strip() for line in file]
        parsed_lines = [parse_line(line) for line in lines if line]
        return parsed_lines

def calculate_totals(data: List[Tuple[str, int]]) -> Tuple[int, int]:
    total_elf_calories = 0
    current_elf_calories = 0
    for name, calories in data:
        if calories == "":
            total_elf_calories += current_elf_calories
            current_elf_calories = 0
        else:
            current_elf_calories += int(calories)
~~~

~~~
    total_elf_calories += current_elf_calories
    return total_elf_calories, max(data, key=lambda x: x[1])[1]
~~~

### `write_a_test_case_for_saving_loading_usage_records_from_persistent_st

~~~
```python
```python
import unittest
from datetime import datetime
import json
from my_module import UsageRecord, Storage

class TestStorage(unittest.TestCase):
    def setUp(self):
        self.storage = Storage('test_data.json')
        self.record1 = UsageRecord(resource="CPU", usage=12345, timestamp=
        self.record2 = UsageRecord(resource="RAM", usage=67890, timestamp=

    def test_save(self):
        self.storage.save(self.record1)
        self.storage.save(self.record2)
        saved = self.storage.load()
        self.assertEqual(len(saved), 2)

    def test_load(self):
        self.storage.save(self.record1)
        self.storage.save(self.record2)
        loaded = self.storage.load()
        self.assertCountEqual([r.resource for r in loaded], ['CPU', 'RAM']
        self.assertAlmostEqual([r.usage for r in loaded], [12345, 67890])
        self.assertEqual(len([r.timestamp for r in loaded]), 2)

    def tearDown(self):
        try:
            open('test_data.json', 'r').close()
        except FileNotFoundError:
            pass
        else:
            open('test_data.json', 'w').close()

if __name__ == "__main__":
    unittest.main()
~~~

Here, `my_module` is a hypothetical Python module that contains the `UsageRecord` and `Storage` classes with appropriate implementations for persistently saving/loading usage records in JSON format. The code above assumes that those classes have already been implemented.

### `test_the_function_to_retrieve_total_usage_for_a_specific_resource_typ

~~~
```python
```python
import boto3

def get_resource_usage(resource_type):
    session = boto3.Session(profile_name='your_aws_profile')
    client = session.client('cloudwatch', region_name='us-west-2') # chang

    namespace = 'AWS/EC2'
    metric = f'CPUUtilization' if resource_type == 'EC2 instance' else 'El

    if not metric:
        return None

    start_time = int(datetime.datetime.now(tz=pytz.utc) - datetime.timedel
    end_time = int(datetime.datetime.now(tz=pytz.utc).timestamp())

    response = client.get_metric_statistics(
        Namespace=namespace,
        MetricName=metric,
        Dimensions=[{'Name': 'ResourceId', 'Value': 'your_resource_id'}],
        StartTime=start_time,
        EndTime=end_time,
        Period=86400, # 1 day in seconds
        Stat='SampleCount'
    )

    total = response['Datapoints'][0]['SampleCount'] if response['Datapoin
    return total


### `test_the_function_to_calculate_the_total_billing_for_all_resources_co

```python
```python
class Resource:
    def __init__(self, price):
        self.price = price

    def get_billing(self):
        return self.price

def total_billing(resources):
    billing_sum = 0
    for resource in resources:
        billing_sum += resource.get_billing()
    return billing_sum

# Example usage:
resource1 = Resource(10)
resource2 = Resource(20)
~~~

~~~
all_resources = [resource1, resource2]
total_bill = total_billing(all_resources)
print(total_bill)  # Outputs: 30


### `implement_caching_or_lazy_loading_techniques_to_improve_app_performan

```python
```python
from functools import lru_cache

@lru_cache(maxsize=128)
def expensive_function(arg):
    # Your expensive function implementation here
    pass

# Usage of the lazy-loaded function
expensive_function('some_arg')
~~~

In this example, `expensive_function` uses the LRU (Least Recently Used) cache from the `functools` module. This means that the most recently used cache entries are kept as long as possible and the least recently used entries are removed first when the cache exceeds its maximum size. The maximum size is set to 128 in this example, but it can be adjusted according to your needs.

~~~
### `add_comments_and_documentation_throughout_the_codebase_to_explain_wha

```python
```python
"""
This module contains utility functions for data processing.

:author: Your Name
:date: DD/MM/YYYY
"""

# Function to calculate the average of a list of numbers
def calc_average(numbers: List[float]) -> float:
    """
    Calculates the average of a given list of numbers.

    :param numbers: A list of floating point numbers.
    :return: The average value as a floating point number.
    """
    total = sum(numbers)
    return total / len(numbers)

# Function to check if a given string is a palindrome
def is_palindrome(string: str) -> bool:
    """
~~~

~~~
    Checks if the given string is a palindrome (reads the same forwards an

    :param string: A string to be checked.
    :return: True if the string is a palindrome, False otherwise.
    """
    return string == string[::-1]


### `optimize_the_performance_of_resource_usage_calculations_especially_fo

```python
```python
import numpy as np

def batch_resource_usage(data, batch_size=10000):
    total = 0
    for i in range(0, len(data), batch_size):
        batch = data[i:i+batch_size]
        resources = calculate_resources(batch)
        total += resources
    return total

def calculate_resources(data):
    # Optimized resource usage calculation for large datasets (e.g., using
    # Replace this function with your custom implementation
    pass
~~~

In the provided code, the `batch_resource_usage` function calculates the total resource usage of a large dataset by processing it in batches. This approach can significantly reduce memory usage when dealing with very large datasets, as only small portions of data need to be loaded into memory at any given time.

The `calculate_resources` function is where you should implement your custom optimization for resource usage calculation. It takes a batch of data and returns the corresponding resource usage. The example I provided leaves this function empty, waiting for your implementation.

~~~
### `implement_a_method_to_display_the_detailed_usage_and_billing_report_b

```python
```python
from collections import defaultdict

class ResourceUsageReport:
    def __init__(self):
        self.usage_report = defaultdict(lambda: 0)
        self.billing_report = defaultdict(float)

    def log_usage(self, resource_type, hours_used):
        self.usage_report[resource_type] += hours_used
~~~

~~~
    def log_bill(self, resource_type, cost_per_hour):
        self.billing_report[resource_type] += cost_per_hour * self.usage_r

    def display_report(self):
        print("\nDetailed Usage and Billing Report")
        print("-----------------------------------")
        for resource_type, usage in sorted(self.usage_report.items()):
            bill = self.billing_report[resource_type]
            print(f"{resource_type}: {usage} hours used ({'$' + str(bill)}


### `write_unit_tests_for_each_function_to_ensure_they_are_working_as_inte

```python
```python
import unittest
from functions import function1, function2, function3

class TestFunctions(unittest.TestCase):
    def test_function1(self):
        self.assertEqual(function1([1, 2, 3]), [3, 2, 1])
        self.assertRaises(TypeError, function1, "string")

    def test_function2(self):
        self.assertEqual(function2("Hello"), "olleH")
        self.assertRaises(ValueError, function2, None)

    def test_function3(self):
        self.assertEqual(function3([1, 2, 3]), [True, False, True])
        self.assertRaises(TypeError, function3, "list")

if __name__ == "__main__":
    unittest.main()
~~~

In this code, I assumed you have a module named `functions.py` containing the functions: function1, function2, and function3. Replace the implementation of these functions according to your requirements. The test cases check if the functions return the expected results for valid inputs and if they raise appropriate exceptions for invalid inputs.

~~~
### `write_a_test_case_for_adding_new_usage_records.py`

```python
```python
import unittest
from your_module import UsageRecord, RecordsCollection

class TestUsageRecords(unittest.TestCase):
    def setUp(self):
~~~

~~~~
        self.records = RecordsCollection()

    def test_add_new_usage_record(self):
        record = UsageRecord("2022-12-31", "device1", 5)
        self.records.add_record(record)
        self.assertEqual(len(self.records._records), 1)
        self.assertEqual(self.records.get_usage_record_by_date("2022-12-31

if __name__ == '__main__':
    unittest.main()


### `create_sql_statements_for_creating_tables_inserting_updating_and_quer

```python
```python
import psycopg2

# Create connection to the PostgreSQL database
conn = psycopg2.connect(
    dbname="mydb",
    user="username",
    password="password",
    host="localhost"
)

cur = conn.cursor()

# Create table for users
cur.execute("""
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL
);
""")

# Insert user into users table
cur.execute("INSERT INTO users (username, email, password) VALUES (%s, %s,

# Update user's password in the users table
cur.execute("UPDATE users SET password=%s WHERE username=%s", ("new_passwo

# Query data from the users table
cur.execute("SELECT * FROM users")
rows = cur.fetchall()
for row in rows:
    print(row)

# Commit and close connection
conn.commit()
~~~~

~~~
cur.close()
conn.close()


### `test_the_validation_of_input_usage_record_before_adding_it_to_the_sto

```python
```python
class UsageRecord:
    def __init__(self, user_id, operation, timestamp):
        self.user_id = int(user_id)
        self.operation = operation.lower()
        self.timestamp = timestamp

    @staticmethod
    def validate_input(user_id, operation, timestamp):
        if not isinstance(user_id, int) or user_id < 0:
            return False

        if len(operation) < 3 or len(operation) > 20:
            return False

        try:
            datetime.strptime(timestamp, "%Y-%m-%d %H:%M:%S")
        except ValueError:
            return False

        return True

    def add_to_storage(self):
        if self.validate_input(self.user_id, self.operation, self.timestam
            # Add to storage logic here
            pass
        else:
            raise ValueError("Invalid usage record")


### `test_all_the_database_related_operations_with_unit_tests_or_integrati

```python
```python
import unittest
from mock import patch
from your_database_module import DatabaseConnection, DatabaseOperation

class TestDatabase(unittest.TestCase):
    @patch('your_database_module.DatabaseConnection')
    def test_connection(self, mock_conn):
        mock_conn.return_value.connect.return_value = None
        db_conn = DatabaseConnection()
        self.assertIsNotNone(db_conn)
~~~

````
    @patch('your_database_module.DatabaseOperation')
    def test_create(self, mock_op):
        mock_op.execute.return_value = True
        op = DatabaseOperation(db_conn=None)
        self.assertTrue(op.create('table'))

    @patch('your_database_module.DatabaseConnection')
    def test_connection_error(self, mock_conn):
        mock_conn.return_value.connect.return_value = Exception('Error con
        db_conn = DatabaseConnection()
        self.assertRaises(Exception, db_conn)

if __name__ == '__main__':
    unittest.main()
````

Replace `your_database_module` with the actual module name where the `DatabaseConnection` and `DatabaseOperation` classes are defined. The test code mocks the database connection to simulate different scenarios and tests various operations like creating a table using unit tests.

~~~
### `review_and_test_the_refactored_code_thoroughly.py`

```python
```python
import unittest
from refactored_module import FunctionA, FunctionB

class TestRefactoredCode(unittest.TestCase):
    def setUp(self):
        self.func_a = FunctionA()
        self.func_b = FunctionB()

    def test_function_a(self):
        self.assertEqual(self.func_a.process([1, 2, 3]), [2, 4, 8])
        self.assertEqual(self.func_a.process([]), [])
        self.assertEqual(self.func_a.process([0]), [0])

    def test_function_b(self):
        self.assertEqual(self.func_b.process('hello world'), 'olleh dlrow'
        self.assertEqual(self.func_b.process(''), '')
        self.assertEqual(self.func_b.process('xyz'), 'zyx')

if __name__ == '__main__':
    unittest.main()
~~~

Assuming that the `refactored_module.py` contains the following code:

```
class FunctionA:
    def process(self, numbers):
        return [x**2 for x in numbers]
```

~~~~
class FunctionB:
    def process(self, string):
        return string[::-1]


### `create_a_build_script_or_automate_the_build_process_using_tools_like_

```python
```python
import os
import shutil
from subprocess import call

# Define paths and project name
project_name = "my_project"
src_dir = f"src/{project_name}"
build_dir = f"{project_name}/build"
dist_dir = f"{project_name}/dist"

# Ensure directories are created
os.makedirs(build_dir, exist_ok=True)
os.makedirs(dist_dir, exist_ok=True)

# Copy source files to build directory
shutil.copytree(src_dir, build_dir)

# Run build command (replace with appropriate command for your tool)
call(["gradle", "build"], cwd=build_dir)

# Copy built files to distribution directory
shutil.copytree(f"{build_dir}/build/outputs/jar", dist_dir)
~~~~

This code uses the `os`, `shutil`, and `subprocess` modules in Python 3. It assumes you are using Gradle as your build tool, but you can replace the "gradle build" command with the appropriate one for other tools like Maven or others.

~~~~
### `create_and_implement_a_test_case_for_detailed_usage_and_billing_repor

```python
```python
import unittest
from datetime import datetime, timedelta

class TestDetailedUsageReport(unittest.TestCase):
    def setUp(self):
        self.start_date = datetime(2021, 1, 1)
        self.end_date = datetime(2021, 1, 7)
        self.usage_data = [
            ("John Doe", "api_calls", 500, self.start_date),
            ("Jane Smith", "api_calls", 800, self.start_date + timedelta(d
~~~~

```
            ("John Doe", "storage", 100, self.start_date + timedelta(days=
            ("John Doe", "api_calls", 300, self.start_date + timedelta(day
            ("Jane Smith", "storage", 250, self.end_date),
        ]

    def test_generate_report(self):
        from report import DetailedUsageReport
        report = DetailedUsageReport(self.start_date, self.end_date)
        report.add_usage(self.usage_data)
        expected_output = [
            ("John Doe", "api_calls", 800, self.start_date),
            ("Jane Smith", "api_calls", 0, self.start_date + timedelta(day
            ("John Doe", "storage", 100, self.start_date + timedelta(days=
            ("Jane Smith", "storage", 250, self.end_date),
        ]
        self.assertEqual(report.generate(), expected_output)

class TestBillingReport(unittest.TestCase):
    def setUp(self):
        self.start_date = datetime(2021, 1, 1)
        self.end_date = datetime(2021, 1, 7)
        self.usage_data = [
            ("John Doe", "api_calls", 500, self.start_date),
            ("Jane Smith", "api_calls", 800, self.start_date + timedelta(d
            ("John Doe", "storage", 100, self.start_date + timedelta(days=
            ("John Doe", "api_calls", 300, self.start_date + timedelta(day
            ("Jane Smith", "storage", 250, self.end_date),
        ]

    def test_generate_report(self):
        from report import BillingReport
        rate = {
            ("John Doe", "api_calls"): 0.1,
            ("Jane Smith", "api_calls"): 0.2,
            ("John Doe", "storage"): 5,
            ("Jane Smith", "storage"): 4,
        }
        report = BillingReport(self.start_date, self.end_date, rate)
        report.add_usage(self.usage_data)
        expected_output = [
            ("John Doe", "api_calls", 80, self.start_date),
            ("Jane Smith", "api_calls", 160, self.start_date + timedelta(d
            ("John Doe", "storage", 500, self.start_date + timedelta(days=
            ("Jane Smith", "storage", 1000, self.end_date),
        ]
        self.assertEqual(report.generate(), expected_output)
```

Assuming the `report.py` file contains the following code:

```
class DetailedUsageReport():
    def __init__(self, start_date, end_date):
        self.start_date = start_date
```

~~~
        self.end_date = end_date
        self.data = []

    def add_usage(self, usage_data):
        for item in usage_data:
            self.data.append((item[0], item[1], item[2], max(self.start_da

    def generate(self):
        return sorted(self.data, key=lambda x: x[3])

class BillingReport():
    def __init__(self, start_date, end_date, rate):
        self.start_date = start_date
        self.end_date = end_date
        self.rate = rate
        self.data = []

    def add_usage(self, usage_data):
        for item in usage_data:
            self.data.append((item[0], item[1], item[2] * self.rate[(item[

    def generate(self):
        return sorted(self.data, key=lambda x: x[3])


### `implement_method_to_add_new_usage_record_timestamp_resource_type_and_

```python
```python
class UsageRecord:
    def __init__(self, timestamp, resource_type, usage_amount):
        self.timestamp = timestamp
        self.resource_type = resource_type
        self.usage_amount = usage_amount

def add_usage_record(records, timestamp, resource_type, usage_amount):
    new_record = UsageRecord(timestamp, resource_type, usage_amount)
    records.append(new_record)
    return records
~~~

You can use this class and method to create a new UsageRecord object with given timestamp, resource_type, and usage_amount, and add it to the list of records. This implementation assumes that you have an initial empty list to store the usage records. For example:

```
records = []
add_usage_record(records, "2022-01-01 10:00:00", "CPU", 50)
add_usage_record(records, "2022-01-01 11:00:00", "RAM", 100)
```

```
### `test_error_handling_during_data_loading_saving_process.py`
```

~~~
```python
```python
import csv

class DataLoader:
    def __init__(self, file_path):
        self.file_path = file_path

    def load(self):
        try:
            with open(self.file_path, 'r') as file:
                reader = csv.reader(file)
                data = list(reader)
            return data
        except FileNotFoundError:
            print("File not found")
        except Exception as e:
            print(f"Unexpected error occurred during loading data: {e}")

    def save(self, data):
        try:
            with open(self.file_path, 'w', newline='') as file:
                writer = csv.writer(file)
                writer.writerows(data)
        except FileNotFoundError:
            print("File not found")
        except Exception as e:
            print(f"Unexpected error occurred during saving data: {e}")


### `write_a_function_to_calculate_monthly_billing_for_each_resource_type.

```python
```python
def calculate_monthly_billing(usage):
    resource_rates = {
        "data": 0.1,
        "computation": 0.075,
        "storage": 0.03
    }

    billing = {}

    for resource_type, usage_amount in usage.items():
        billing[resource_type] = resource_rates.get(resource_type, 0) * us

    return billing


### `implement_a_method_to_calculate_daily_billing_for_each_resource_type.

```python
~~~

~~~
```python
class Resource:
    def __init__(self, usage, rate):
        self.usage = usage
        self.rate = rate

class Billing:
    def __init__(self, resources):
        self.resources = resources

    def daily_billing(self):
        total_bill = 0
        for resource in self.resources:
            total_bill += resource.usage * resource.rate
        return total_bill

# Example usage
data = [Resource(10, 5), Resource(20, 7)]  # First element is usage, secon
billing_calculator = Billing(data)
print(billing_calculator.daily_billing())


### `write_a_function_to_save_the_usage_records_in_a_persistent_storage_fi

```python
```python
import json

def save_usage_records(records, filename='usage_records.json'):
    try:
        with open(filename, 'w') as f:
            json.dump(records, f)
    except Exception as e:
        print(f"Error saving usage records: {e}")

def load_usage_records(filename='usage_records.json'):
    try:
        with open(filename, 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        return {}


### `develop_a_function_to_calculate_yearly_billing_for_each_resource_type

```python
```python
def calculate_yearly_billing(resource_type, usage, price):
    def daily_billing(usage, price):
        return usage * price

    def yearly_billing(days_in_year=365):
~~~

~~~
        return daily_billing(usage, price) * days_in_year

    return {resource_type: yearly_billing}


### `implement_user_interface_for_entering_and_viewing_the_usage_records_i

```python
```python
import tkinter as tk
from tkinter import messagebox

class UsageRecordUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Usage Record UI")
        self.frame = tk.Frame(self.root)
        self.frame.pack()

        self.entry_id = tk.Entry(self.frame, width=20)
        self.entry_id.grid(row=0, column=0, padx=10, pady=10)

        self.entry_date = tk.Entry(self.frame, width=20)
        self.entry_date.grid(row=1, column=0, padx=10, pady=10)

        self.entry_usage = tk.Entry(self.frame, width=20)
        self.entry_usage.grid(row=2, column=0, padx=10, pady=10)

        self.btn_add = tk.Button(self.frame, text="Add Record", command=se
        self.btn_add.grid(row=3, column=0, padx=10, pady=10)

        self.listbox = tk.Listbox(self.frame, width=50, height=10)
        self.listbox.grid(row=4, column=0, rowspan=5, padx=10, pady=10)

    def add_record(self):
        id = self.entry_id.get()
        date = self.entry_date.get()
        usage = self.entry_usage.get()

        if not id or not date or not usage:
            messagebox.showerror("Error", "All fields are required.")
            return

        self.listbox.insert(tk.END, f"ID: {id}, Date: {date}, Usage: {usag
        self.clear_entry()

    def clear_entry(self):
        self.entry_id.delete(0, tk.END)
        self.entry_date.delete(0, tk.END)
        self.entry_usage.delete(0, tk.END)

root = tk.Tk()
~~~

```
UsageRecordUI(root)
root.mainloop()
```

This code creates a simple Tkinter GUI for entering and viewing usage records. The user can input an ID, date, and usage in the text boxes, then click "Add Record" to add the record to the listbox below. The entries are automatically cleared after adding a record.

~~~
### `design_the_structure_for_cloudusage_data_model_classes_or_structures_

```python
```python
from datetime import datetime
from peewee import *

db = SqliteDatabase('cloud_usage.db')

class BaseModel(Model):
    class Meta:
        database = db

class CloudProvider(BaseModel):
    provider_id = IntegerField()
    name = CharField()

class Service(BaseModel):
    service_id = IntegerField()
    name = CharField()
    cloud_provider = ForeignKeyField(CloudProvider, backref='services')

class Usage(BaseModel):
    usage_id = IntegerField()
    timestamp = DateTimeField(default=datetime.now)
    service = ForeignKeyField(Service, backref='usages')
    resource = CharField()
    amount = FloatField()
~~~

This code uses the Peewee ORM for SQLite to define a CloudUsage data model. The `CloudProvider` class represents cloud providers with an id and name. The `Service` class represents services offered by a cloud provider, linking back to the `CloudProvider`. The `Usage` class represents usage of a service, recording the timestamp, resource used, and amount. Each Usage is linked to its corresponding Service.

~~~
### `implement_a_method_for_handling_errors_during_data_loading_saving_pro

```python
```python
import sys
~~~

```
import os
from typing import Callable

class DataHandler:
    def __init__(self, load_data: Callable, save_data: Callable):
        self.load_data = load_data
        self.save_data = save_data

    def load(self, filepath: str) -> object:
        try:
            data = self.load_data(filepath)
            return data
        except Exception as e:
            print(f"Error occurred while loading data from {filepath}: {e}
            sys.exit(1)

    def save(self, filepath: str, data: object):
        try:
            self.save_data(filepath, data)
        except Exception as e:
            print(f"Error occurred while saving data to {filepath}: {e}")
            sys.exit(1)
```

This code defines a `DataHandler` class that takes two functions (`load_data` and `save_data`) as arguments during initialization. The `load` method attempts to load data from a given file path, catching any exceptions and exiting the program if an error occurs. Similarly, the `save` method tries to save the provided data to a specified file path, also handling errors and terminating the program if necessary.

~~~
### `test_the_build_script_and_ensure_that_it_works_as_expected_with_diffe

```python
```python
import unittest
from pathlib import Path
from your_build_script import BuildScript

class TestBuildScript(unittest.TestCase):
    def setUp(self):
        self.test_dir = Path(__file__).parent / 'tests'
        self.bs = BuildScript(self.test_dir)

    def test_build_with_default_config(self):
        output = self.bs.run()
        # Adjust this assertion according to the expected behavior of your
        self.assertTrue(output, msg="Build failed with default config")

    def test_build_with_custom_config(self):
        custom_config_path = self.test_dir / 'custom_config.yaml'
        output = self.bs.run_with_config(custom_config_path)
~~~

~~~
        # Adjust this assertion according to the expected behavior of your
        self.assertTrue(output, msg="Build failed with custom config")

if __name__ == '__main__':
    unittest.main()
~~~

This code provides a basic test structure for testing your build script using Python's built-in `unittest` module. It assumes that you have created a YAML configuration file (custom_config.yaml) in the 'tests' folder to test with custom configurations. Adjust the tests and assertions according to the expected behavior of your build script.

~~~
### `prepare_release_notes_detailing_changes_made_in_the_new_version_of_th

```python
```python
class ReleaseNotes:
    def __init__(self, version, changes):
        self.version = version
        self.changes = changes

    def to_markdown(self):
        notes = "\n# Release Notes v{}".format(self.version)
        notes += "\n\n## Changes\n"
        for change in self.changes:
            notes += "- {}".format(change)
        return notes

def update_release_notes(old_version, new_version, changes):
    if old_version and new_version:
        release_notes = ReleaseNotes(new_version, changes)
        previous_release_notes = ReleaseNotes(old_version, ['Previous rele
        return [previous_release_notes.to_markdown(), release_notes.to_mar
    elif old_version:
        return [ReleaseNotes(old_version, ['Previous release notes can be
    else:
        return [ReleaseNotes(new_version, changes).to_markdown()]


### `write_code_to_connect_to_the_database_if_using_a_relational_database_

```python
```python
import psycopg2

def connect_db():
    try:
        conn = psycopg2.connect(
            dbname="your_database_name",
            user="your_username",
~~~

```
            password="your_password",
            host="your_host",
            port="your_port"
        )
        return conn
    except Exception as e:
        print(f"Error: {e}")
        return None
```

### `test_the_deployed_application_in_the_staging_environment_and_fix_any_

~~~python
```python
import requests
import json
from time import sleep

STAGING_API_URL = "https://your-staging-app-url.com"
ISSUE_REPORT_URL = STAGING_API_URL + "/report_issue"

def check_api():
    response = requests.get(STAGING_API_URL)
    if response.status_code != 200:
        report_issue("API returned non-200 status code")

def run_test_case(test_case):
    response = requests.post(STAGING_API_URL + "/run", json=test_case)
    if response.status_code != 200:
        report_issue(f"Test case {test_case['name']} failed")

def report_issue(message):
    data = {"message": message}
    requests.post(ISSUE_REPORT_URL, json=data)

def main():
    test_cases = [
        {"name": "Test case 1", "parameters": {}},
        # Add more test cases here...
    ]

    for test_case in test_cases:
        run_test_case(test_case)

    check_api()

if __name__ == "__main__":
    main()

~~~

🧠 Combined Effort Estimation (SBERT + Code Analysis)

🔍 Top Similar GitHub Repositories (Filtered)

Repository	Estimated Effort (hours)	Description
huataihuang/cloud-atlas-draft	209.44	Cloud Compute Technology Guide
NewcastleComputingScience/csc8110	22.00	Cloud Computing Coursework
abhishekakumar/GatorSquad	117.68	Cloud Computing Course Project
l-wang/15619-Cloud-Computing	300.32	Cloud Computing
wenh/wenh-cloud2	4.72	cloud computing

Average Effort (without outliers): 130.83 hours

💻 Estimated Effort for Your Generated Code

- **owner**: local
- **repo**: 0b24e922-a408-4de2-bbaf-6794e7778e5a
- **C_comp**: 12.623
- **complexity_mode**: power
- **Optimistic**: 9.45
- **Most_Likely**: 12.62
- **Pessimistic**: 32.53
- **Effort (days)**: 16.67
- **estimated_effort_hours**: 133.36
- **effort_details**: {'C_comp': 12.623, 'Optimistic': 9.45, 'Most Likely': 12.62, 'Pessimistic': 32.53, 'Effort (days)': 16.67, 'hours': 146.73}

📜 Policy Analysis

Default Policy

Ensure user input is validated. Avoid hardcoding credentials. Use secure file handling.