Author: Marco Agatensi

Student ID: 7172615

email: [marco.agatensi@edu.unifi.it](mailto:marco.agatensi@edu.unifi.it)

**PROJECT REPORT**

**Objective of the Project**

The final goal of this project is to develop an Anomaly Detector to be used at run time to detect ongoing anomalies in my laptop.

In short, this project can be divided in three phases:

1. System Monitoring for data collection
2. Training of a binary classifier
3. Integration of the monitor with the classifier to develop the final anomaly detector

**0 – Folder Structure**

*DCML-CPS\_Project*/

|

|---*log*/

|---*output\_folder*/

|

|---*saved\_models*/

| |--- best\_model\_stacking.pkl

| |--- scaler.pkl

|

|---*src*/

| |---*monitoring*/

| | |--- AnomalyDetector.py

| | |--- InjectionManager.py

| | |--- LoadInjector.py

| | |--- SystemMonitor.py

|

| |---*utils*/

| | |--- SeverityLevel.py

| | |--- SystemState.py

| | |--- utilities.py

|

|--- main\_anomaly\_detector.py

|--- main.py

**1 – System Monitoring**

A monitor is a process that reads some values of the system to be monitored. First of all, to build a monitor, it is then necessary to define what resources has to be monitored: I chose to monitor the CPU and the RAM using the python library psutil. In detail the monitor reads data about:

* CPU → For each logical core (8 cores) are monitored the times (percentage), frequencies and percentages of usage. For the entire CPU is monitored the frequency, while the percentage of usage is obtained by averaging the utilization percentage of each individual core to avoid inconsistencies between the utilization percentages of individual cores and that of the entire CPU (which would be obtained through measurements at different instants of time). For each physical core (4 cores) the temperature is monitored.
* RAM → all information that can be read about the RAM using psutil.virtual\_memory() function are monitored (memory available, free, used, inactive, buffers, cached, shared, ecc).

To make the code as reusable as possible, I wrote the code so that it was organized into classes and each of them has some parameters that allow for some customization in the creation of the objects.

The monitor is represented by the SystemMonitor class (file SystemMonitor.py). The constructor of this class has some parameters that allow the user to build different types of monitor. The monitor also saves the current state of the system using the enum SystemState (SystemState.NORMAL or SystemState.UNDER\_INJECTION).

To gather system data, the class SystemMonitor has the method monitor() that returns a dictionary which represents a datapoint (set of monitored parameters at a certain instant of time and in additions the name of the current fault injection, that has value “None” if there is no ongoing fault injection). This class also provide the method start\_injection(str), that has to be called before starting a fault injection to set the current state of the system to SystemState.UNDER\_INJECTION and to save the type of the current injection. When an ongoing injection stops, the other method end\_injection() has to be called to set the state of the system to SystemState.NORMAL and to set the current injection to “None”.

The injection of anomalies is done using the InjectionManager class which makes use of the LoadInjector class to load the injections. The user, to inject anomalies into the system has to create an instance of the InjectionManager class passing as arguments to the constructor at least the path to the json file that contains the list of injectors and the estimated duration of each injection. The InjectionManager class has the method read\_injectors(bool) that reads the json file and save the injectors into a list. If the object of this class has been created passing the parameter inj\_number with a value greater than the number of injectors in the json file, then additional injectors, randomly chosen from those already present, are added to the list until that value is reached. Each injector in the list is an instance of the LoadInjector class. The user, after calling the read\_injector() method, can start an injection by calling the method inject\_fault() that take the first element of the list of injectors and call on it the method inject() of the class LoadInjector that starts the real injection. When the user want to stop the current injection, it has to call the method stop\_injection() of the InjectionManager class that using the force\_close() method of the class LoadInjector, stops the current injection. The LoadInjector class, is an abstract class with two implementation: CPUStressInjection, that performs the injection on some single logical core or on the entire CPU, and MemoryStressInjection that performs the injection on the RAM. To overload the entire CPU or its logical core are used the functions load\_all\_core () and load\_single\_core of the python library cpu\_load\_generator. The load on the target single core is dynamically determined based on the current load of the other cores. For this purpose, there is an instance of the SystemMonitor class that monitors only the CPU current usage. The RAM is filled up until a certain threshold, 90%, of the virtual memory usage is reached. When this threshold is reached, the RAM is discharged by a random amount to avoid a system crash. For this purpose, there is an instance of the SystemMonitor class that monitors only the VM current usage.

In the file main.py there is the main method in which the monitor and the injection manager are created. Then, there is a loop in which a phase of normal behavior is alternated with a phase of injection, until all the injectors have been injected. Datapoints that are monitored are written in a csv file using the function write\_dict\_to\_csv() in the file utilities.py. For each normal phase 120 datapoints are written to the csv, and for each injection 80 datapoints are written. The number of injections performed is 60 and then at the end of monitoring, that lasts around 2 hours and half, the final csv dataset contains 12.000 datapoints. After each monitoring step, only a sleep of 0.1 second is done as the monitoring system, to collect more accurate cpu data, already takes 0.6 seconds. The output csv file of the dataset is saved in the folder output\_folder.