Technical Report

Project Title: Moore's Law Web Service

1. **abstract**

We want to create a web service application using flask to communicate with a database and send http responses. The requests are sent by a client library which includes various methods a user might use, for example: get, update, delete, etc.

2. **Introduction**

Moore's Law is the observation that the number of transistors in an integrated circuit doubles roughly every two years, while the cost of producing those circuits remains the same or decreases. It's a historical trend and an extrapolation, not a physical law, that has been a driving force in the semiconductor industry for decades. This law helped shape and define our current understanding of the semiconductor industry and is usually taught in every introduction to computer architecture course globally.

Even though this Law is not a physical law, we can still provide the data and the methods any data scientist who wants to prove this law, experiment with relevant data and verify whether this law applies to other computer components. This is exactly what our project provides to our clientele, a fully functioning webservice with a well documented and easy-to-use client-side library to allow them to fully explore freely without the worry of large data management.

Over the course of this report, the process and creation of this service will be documented and explored over what has been provided, how, and the current limitations of this tool.

3. **Data Gathering and Initialisation**

```
+---initialize
ª   ª   add_data_to_mongodb.py
ª   ª   download_data.py
```

To get started we need our yet to be created web service to get the data from somewhere. Hence we need to create a MongoDB database instance on the local network of the device. To do that, the project directory folder has an initialize folder. This folder contains two scripts; “download_data.py” allows us to gather the data from online and download it to our local machine, if we don't already have it, and “add_data_to_mongodb.py” helps us to migrate our local csvs to a mongodb instance.

The first script “download_data.py” uses the requests dependency to read data from 3 different urls; cpu, gpu and ram data in csv format, given the request has a status code of 200.

To save the data locally, the script creates a subdirectory (if not already present) where it adds our files.

The script “add_data_to_mongo.py” instantiates a MongoDB client. One of the functions (add_data) takes data from our previously downloaded files. It extracts all the files and splits them with a predetermined regex pattern (since the names of the components include commas). Additionally, since all the data is in string format, the script makes a singular string instance into the appropriate data type unless it is a “NA”, in that case it keeps it as a string. The second function clears the database and removes all the collections previously created by the first function.

Running these two files, first “download_data.py” then “add_data_to_mongo.py” helps us get the data we need and migrate it to MongoDB. This will greatly help us with the rest of the project.

4. **Web Service Project**

```
+---web_service_project
   ª   Pipfile
   ª   Pipfile.lock
   ª   run_web_service.py
   ª   __init__.py
   ª
   +---common_route_methods
   ª   ª   parse.py
   ª   ª   __init__.py
   ª   ª
   ª
   +---routes
   ª   ª   cpu_web_service.py
   ª   ª   gpu_web_service.py
   ª   ª   ram_web_service.py
   ª   ª   __init__.py
```

The web service project includes separate folders all serving for different functions of the actual web service. “run_web_service.py” invokes all the different routes which have been created and can be easily expanded to include other components to the service.
The script “run_web_service.py” depends on the flask and mongo client libraries.
Previously, we created a MongoDB client, now we proceed to create a flask application which can receive requests to query the database.
Each component we have (cpu, gpu and ram) can have its own methods to request information, thus we use blueprints. Each component provides a route where the actual web service, for that particular component, is made. In this case we have very similar methods for all the components, but if in a future update a component may need its own unique method, this can easily be done without too much of a hussle. After registering each route to the web service application, the “run_web_service.py” script proceeds to include some universal methods for the

web service itself, for example: the home method which provides an informative message for the user, get_all_components, and get_component_names which output all the data or common useful information.
Finally, under the main, we create and start the web service in port 5050.

Going back to the routes, each component gets its own unique “mini” web service allowing it to output data. Each component has its own get, add, update and delete functions routed to certain http requests. The methods output a dictionary with data or a useful message about the request. If an error arises from a request, the message will include something like: “{“error”: …}” with an error code. Otherwise, if the request tries to get data, the output will be a list of dictionaries or simply an empty one if nothing was found. Finally, to modify, add or delete data, a message will be outputted looking something like this: “{“message”: …}” with a success code.

The **GET** method expects a query, this query is inside the arguments of the http request. To gather the right information to output, a query parser is invoked to convert the query dictionary of the http into a mongo readable one.

The **POST** method (aka. add data) expects the http request to come with data in the body of the request. given that the data has the same structure as the one in the MongoDB for that particular collection, the method inserts the json as a dictionary in the corresponding collection.

The **PUT** method (aka. update) expects both a query and the specific part of the component it has to update. This means it gets a query from the http argument, it parses it, it checks if the component exists and not more than one does, then finally takes the data from the body of the request and converts the correct component part to the given value.

The **DELETE** method expects a query or an empty query to know what to delete. After parsing the query and matching it to the data in the database, it then removes the found components from the collection. If the query is empty, it deletes the whole collection.

The typical Mongo query looks very similar to a normal dictionary. The “parse.py” script includes a useful method to convert a dictionary of request values into a MongoDB readable query. It does so by converting values which are supposed to be integers into integers, so later we can compare them. Then it allows us to get values less than (<), more than (>), less than or equal (<=) or more than or equal (>=) to a given value in the query by adding *__gt*, *__lt*, *__gte* or *__lte* to the end of a feature (component part). This usually gathers a whole list of components which fit the criterion. For the rest of the query, the dictionary is insightful enough that we can use it directly in the mongo database thus we add each parameter to the final outputted mongo query.

Since the web service directory is a package, each folder has its own “__init__.py” script. This is not inherently necessary, but it is good practice to include, especially if an older version of python is run.

In summary the web_service_project folder allows for initialization of the web service. “run_web_service.py” needs to be running for the client library to work. This is because the flask application creates a local server allowing a user to receive data from their requests.

5. **Client library**

```
+---client_package
a    a    Pipfile
a    a    Pipfile.lock
a    a    setup.py
a    a    __init__.py
a    a
a    +---build
a    +---client_package_library.egg-info
a    +---dist

a    a
a    +---web_client
a    a    a    component.py
a    a    a    cpu.py
a    a    a    gpu.py
a    a    a    main.py
a    a    a    ram.py
a    a    a    turn_dict_to_obj.py
a    a    a    __init__.py
```

The client library is essentially another folder which has been used to create a wheel package allowing it to be used all throughout the machine.
Inside the library we have a “main.py” script which has many methods a client might want to use to send requests to the web service instance (which should be running in contemporary). The “turn_dict_to_obj.py” script is another tool inside the library, this allows to convert the dictionaries into custom made objects with various use cases.

To make the client package into a sort of library we need the setuptools dependency, this dependency allows us to convert the whole folder into a wheel file. Wheel files can be used as libraries throughout the machine. The use case of this is to utilize the wheel file as a classic library for a user to have (this will be used in the demo application). The wheel package is found under the /dist directory.

The main part of the web client is found under the /web_client directory. Here the “main.py” script includes all the various methods a user can use to communicate through the web_service to the database.
Which include:

- home_message……..…….gets the home message as a dict
- get_component_names……..gets the names of all the collections/components
- get_component_features……specifying a component, you gets its features

- get_all_collections……get all of the data
- get_a_collection……get a collection of data
- check_if_component_exists……check if a component exists
- get_specific_component_s…….get specific components based on a query
- add_data……adds a singular component
- add_many……..adds multiple components
- update_a_component……..updates a component given query and updated data
- delete_component_s……deletes components given the query
- delete_collection……deletes a whole collection
- plot_components……plots a given components by a given feature

To include objects, each component can be converted into a custom python object. “turn_dict_to_obj.py” is a script allowing a user to input a dictionary, or a list of dictionaries depending on the method, and obtain an object corresponding to that component. Once again, this is expandable, meaning that in the future we can add more component classes which might have their own unique functionality.
As objects, each component can still access the functionalities of the main methods, it is versatile in that sense.

The “cpu.py”, “gpu.py” and “ram.py” scripts include the cpu, gpu and ram classes. The current objects have some similarities, thus they are children to the component class. If in the future a new components need to be added, they can inherit from the component class to save some common functionalities, which include:

- describing the component features
- returning the object as a dictionary

All the current classes differ just by their features, but each one has similar functionalities; everytime a new object is created, it automatically is added in the database, unless it's already present, everytime a setter method is called (after initialization) for that particular component object feature, it is replicated by an update in the database.
Finally, if ever any feature value is not valid, a value error is raised.

6. Demo project

```
+---demo_project
a       demo.py
a       Pipfile
a       Pipfile.lock
a       __init__.py
```

The demo project is meant to represent and show off all the different functionalities of the client library. Additionally “demo.py” is dependent solely from the client library.

Here we find a few more important methods the client library offers. Mainly it is showing how the methods from "main.py" were designed to be used, everytime you create a new object it is automatically added and updated into the dataset, if you like to deal with only dictionaries you can also do so.
You can get feature information, update, delete and plot all the components or a list of them.

7. **testing**

```
+---testing
a    a    cpu_test.py
a    a    main_test.py
a    a    Pipfile
a    a    Pipfile.lock
```

The testing module includes unit tests for all the component classes and the "main.py" methods of the client library. The unit tests are done with pytest, hence the dependencies are found under dev in the pipfile. Additionally, testing uses the client library, thus it is also found in the pipfile.

Whenever a parameter for a method is bad, the method raises a ValueError.
Whenever a component is added, updated or deleted, a message is printed. If the component is already present or doesn't exist, the message will say so.

The component class testing is done by:
checking if the component is added whenever the object has been created.
checking if the component feature has been updated once the setter is called for that feature.
checking if the get_dict method returns the correct dictionary corresponding to the component

To test the "turn_dict_to_obj.py" methods we just simply create a component dictionary or a list of them. Then we make them into objects and test if all the keys and values still correspond.

8. **Docker**

Docker containerizes our entire software environment, ensuring that the application and all its dependencies run consistently across different machines. For this project, it encapsulates the **web service** and the **MongoDB database**, into isolated containers, each defined in a `docker-compose.yml` configuration. This eliminates setup discrepancies between development, and deployment environments.