FiveWest Software Engineering Applicant Assessment

Applicant: Victor Bantchovski

Question 1 – Theory

1.1 Concurrency and Parallelism

1.1.1. *Please explain the differences between the distinct concepts of parallelism and concurrency in computer science.*

Both concurrency and parallelism describe non-sequential computational processes. Concurrency refers to the *overlapping*, yet not necessarily simultaneous, execution of two or more tasks. It is possible for concurrent processes to occur on a single processor. Parallelism deals with the *simultaneous* execution of two or more tasks or subtasks – this requires at least two processors.
We can have a program that is concurrent, parallel, both or neither.

1.1.2. *In what sort of contexts would you want to use parallelism, and in which contexts is concurrency sufficient?*

Concurrency alone is suitable in situations where the tasks undertaken are relatively light and a single processor can seamlessly handle their overlapping executions. In situations where the task or tasks are more computationally intensive, parallelism is preferable as the computations can be distributed over multiple processors and executed simultaneously – resulting in a net reduction of time for the task to complete.

1.1.3. *What is Python's GIL (Global Interpreter Lock)? How can one achieve parallelism in Python, given the GIL?*

The Python Global Interpreter Lock (GIL) is a mutual exclusion object which acts as a lock allowing only one thread at a time to have control of the Python interpreter. This results in thread-safe memory management and, since only one lock is in use, sidesteps the issue of deadlocks. Though GIL improves the performance of non-parallel programs (since only one lock is required) its consequence is that one is unable to execute multiple threads of code simultaneously. This issue is resolved by the implementation of multi-processing. Multiple processes are created, each with its own interpreter, memory, threads and GIL.

1.1.4. *Async is a popular concurrent paradigm. When would it be appropriate to use it, and how does it work? (for example, in Python's async.io library or in Node.js)*

Async IO is best used when there are IO-bound tasks contained in the program. This takes advantage of the potentially wasted time the program is idle for and repurposes for the execution of other tasks. An implementation of this paradigm is through the execution of a coroutine in the program, which upon reaching its IO bottleneck sends a message to the

event loop to run some other code until the IO phase completes.

1.1.5. *In concurrent and parallel programs, how do concurrently executing tasks communicate? Contrast the paradigms.*

In single-processor concurrent programs, all threads share the same address space – hence communication between tasks is trivial. However, in multi-processor parallel programs inter-process task communication is not so straightforward as each process resides in its own address space.
One way that processes can communicate is via shared memory, whereby the processes involved carry a shared memory region of address space. Another way is through message passing – communicating via the exchange of messages using the operating system as a middle-man. Both of these methods have their own downsides and upsides.
The differences in inter-task communication between the concurrent and parallel paradigms come down to the fact that parallel processes are truly simultaneous and separate – finding a point of connection is not as trivial and many complexities arise.

1.2. Databases

1.2.1. *What is a database index and how are they implemented?*

A database index is a data structure which allows for faster database lookup operations. It is essentially a copy of selected columns from a table whose rows are then organised in a way which can facilitate faster lookup – each row carrying a reference back to the original table records. Typical data structure implementations of indexes utilise trees and hash tables.

1.2.2. *Why do databases support more than one type of index implementation (reference examples)?*

The different types of index implementations each carry their own benefits and pitfalls. An example is the clustered index, which sets an order to the way the data is physically stored on the disk. This allows for fast lookups when a specific range is being considered, especially if it all happens to lie in the same disk partition. However, overheads can occur when updating the original table since that affects the physical arrangement of the data on the disk. A non-clustered index can be implemented to avoid this issue, with the cost of it generally being slower for data retrieval. Indexes are usually optimised for searching in some ascending order, so the reverse index was introduced as a similar option which is optimised for searching in descending order.

1.3. Memory-Safety

1.3.1. *What does it mean for a language (or code) to be memory-safe? Why would you want a language to have the potential to not be memory-safe?*

A memory-safe language or piece of code is one which guarantees the protection of the integrity of the memory which a program interacts with from possible access errors or memory leaks.

Unsafe languages give more control to the programmer. Raw pointers and raw memory can be managed directly and result in more optimised processes. Also, static objects can be altered.

1.4. Microservices & Authentication

1.4.1. *What is microservice architecture? What are some pros and cons of this system design choice?*

Microservice architecture is an architectural style used in the development of applications which separates the application into a collection of loosely coupled and independently testable services.
Pros: Microservices can be developed and deployed independently of other microservices, which allows for faster and easier software development and scaling. Furthermore, the relative independence of microservices reduces the communication requirements between teams working on different microservices.
Cons: The decoupling between microservices can be difficult to maintain and the decoupling itself has its own drawbacks like forming information barriers, resulting in an increase in inter-service message processing time and making testing more complicated. Additionally, the architectural infrastructure may result in increased complexity in programs that would otherwise be simpler and makes refactoring responsibilities of microservices more difficult.

1.4.2. *What are a few ways different microservices can communicate with each other?*

Several forms of inter-microservice communication are detailed below.
HTTP Communication: synchronous HTTP calls between two services are simpler to implement but result in a greater coupling. Asynchronous calls fix this issue, however, are more complicated to implement.
Message Passing: entirely sidesteps the issues of direct service-to-service communication and instead services communicate with a central platform. Complexities of this method involve finding ways to keep services updated on the status of other services on which they depend. Furthermore, a coupling is introduced between services which wish to communicate via the requirement for an agreed upon structure and content of the message.
Event-driven Communication: similar to message passing, but the service only listens for the occurrence of an event rather than the details of the event and its corresponding service. This keeps the services even more loosely coupled.

1.4.3. *What are some different ways to authenticate yourself against a secure service endpoint?*

Several ways to authenticate oneself against a secure service endpoint are given below.
Basic authentication: Requires a username and password to authenticate the user. Generally considered the least secure method.
Token authentication: A private token is used to grant authentication. Tokens are issued by an authentication service and are stateless and self-contained.
Open authorisation (OAuth) authentication: An access token is granted by the user to a third party which enables it to access information from another third party temporarily.

1.5. Application Design

1.5.1. *What is a design pattern? Can you explain the Model-View-Controller (MVC) design pattern? What sort of frameworks mostly make use of this pattern?*

Design patterns are reusable solutions to general problems that occur in software development. The Model-View-Controller design pattern separates the presentation of information to the user (the view) from the actual data and logic of the application (the model). The controller is then what manages the interface between the view and the model. Frameworks such as angular.js and backbone.js use this design pattern.

1.5.2. *What is the Document Object Model (DOM)? Why use a virtual DOM?*

The Document Object Model is a programming interface for HTML and XML documents. It allows one to create, access and manipulate web documents. A virtual DOM has the benefits of being light and faster to update than an actual DOM.

1.5.3. *What is server-side rendering? What are its advantages and disadvantages?*

Server-side rendering is when an application is converted into an HTML page on the server which is then sent through to the client.
Advantages: Pages load faster and there are fewer compatibility issues on the browser side.
Disadvantages: Can be resource expensive as the server takes the load of rendering the pages and rendering non-static or large and complex pages can cause slow-down.

1.5.4. *What is JSX? How is it interpreted by browser JavaScript engines?*

JSX is a JavaScript Syntax Extension in React, which allows one to write React components in HTML-like code. Browsers are unable to directly read JSX, instead the JSX is compiled as JavaScript code which is then interpreted by browsers.

Question 2 – Problem Solving/Algorithms

2.1 Capital Reallocation

The algorithm was written using Python 3.9, with the only libraries being Pandas (for dataframe support) and Sys (for args input support).

The Capital_Reallocation.py file can be run from the terminal. For the default example from the assignment sheet, the argument `'example'` (with apostrophes) can be given. Otherwise, the first argument should be the old allocation of accounts csv file and the second argument should be the new allocation csv file. These csv files should be present in the directory of the program itself and will be turned into dataframes, so similar formatting as in the assignment sheet should be used. Particularly, there should be an 'Account_name' column which will be used as the index.

The repository has with it included .csv files constructed from the assignment sheet example inputs. An example of terminal input follows below:

```
python Capital_Reallocation.py old_df.csv new_df.csv
```

2.2 Index Rebalancing

The algorithm was written using Python 3.9, with the only libraries being Pandas (for dataframe support) and Sys (for args input support).

The Index_Rebalancing.py file can be run from the terminal. For the default examples from the assignment sheet, the argument `'example'` followed by options `'1'` or `'2'` (with apostrophes) can be given. Otherwise, the first argument should be a .csv with a 'Ticker' column (to be used as an index), a market cap 'MCAP' column and a 'Price' column; the second argument should be the float asset_cap of the index and the third argument the total capital available. The .csv file should be present in the directory of the program itself and will be read as a dataframe, so similar formatting as in the assignment sheet should be used.

The repository has with it included a .csv file constructed from the assignment sheet example dataframe input. An example of terminal input follows below:

```
python Index_Rebalancing.py in_df.csv 5 10000
```

Question 3 – Systems Design

The digital parking payment system will be developed using a microservice architecture. The systems diagram below can be used for reference. The service consists of two applications – driver oriented and manager oriented – both of which have a mobile app and web app form. The registration process (drivers providing user, vehicle and payment details and managers providing manager, parking lot, banking and rate details) is not indicated in the diagram in order to declutter but will interlay with the *authorisation* and *driver/manager accounts* services. Thereafter, whenever a user wishes to engage with the app, the app service will first engage with the *authorisation service* before establishing its communications with the respective *accounts service*.

The *camera system services* are linked to the cameras themselves and automatically detect the number plates of vehicles entering the parking lot, the service then updates the respective *driver account* with the necessary details, and the driver receives a notification and can view the amount required to be paid in real time on the app. When the driver leaves the parking lot, his vehicle's number plate is detected by the *camera system* which updates the *driver account* that an exit is occurring, the *payment service* is then engaged and payment is made via the user's designated payment method. Success/failure of payment then updates the *driver account*, which in turn notifies the user.

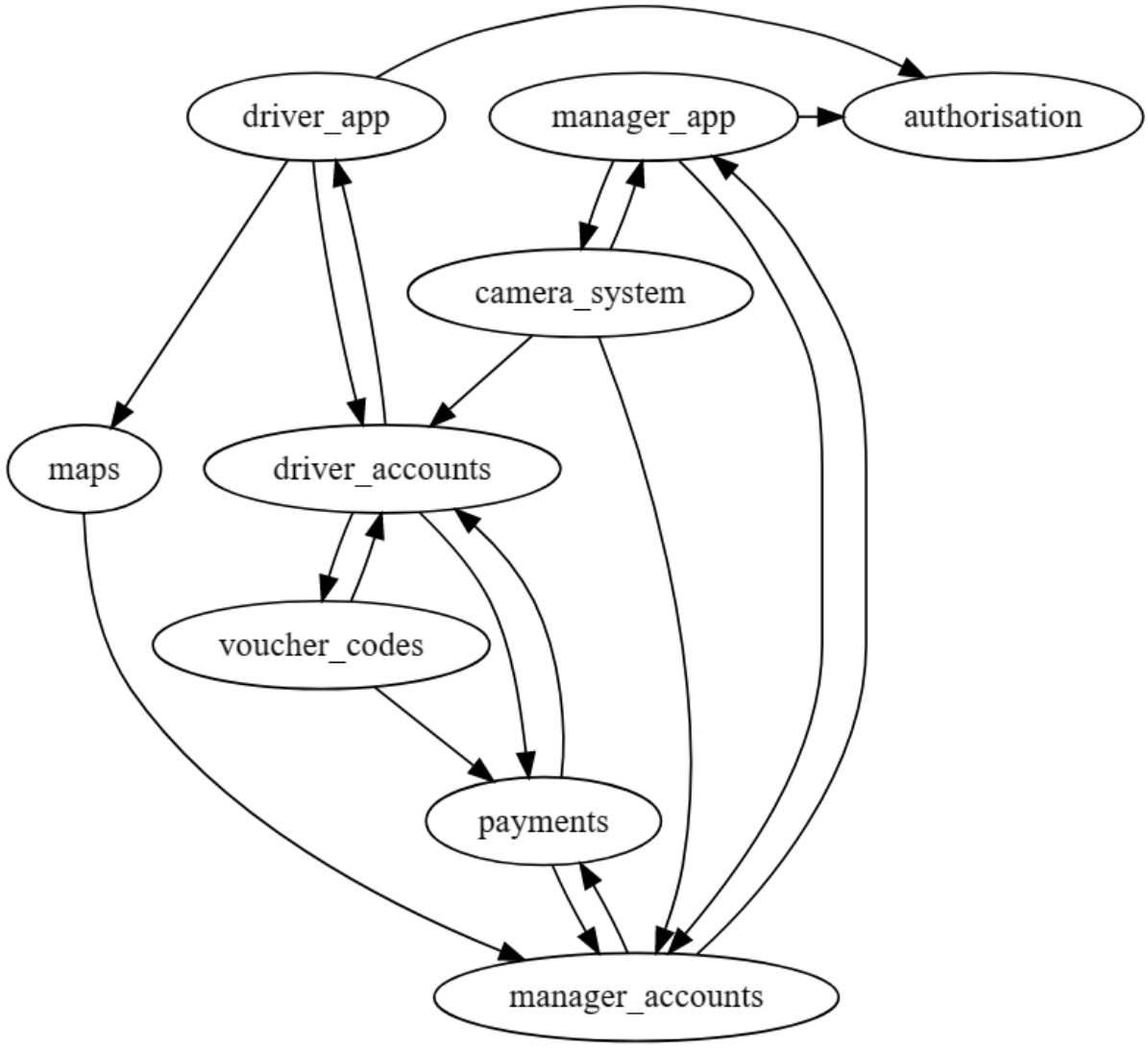
The parking lot managers login through their own app, which interfaces with the *manager accounts service* and the *camera systems service*. Via the *camera systems service* live and historical video and images can be accessed through the app. Access to the *manager accounts services* allows a manager to update the billing rates/change to a demand-based billing and view usage analytics. The *manager*

accounts service communicates with the *payments service* so that the manager-user can keep a live update and record of payments, costs and profit.

The *driver app* possesses a map feature which obtains its data from the *maps service*. The *maps service* maintains communication with the *manager accounts* to be able to keep accurate details and locations of the active parking lots. Drivers can also purchase vouchers through the app, which communicates the request with the *voucher code service* via the *driver accounts service*. Additionally, vouchers can be purchased at automated pay stations at the parking lots. To use a voucher, one simply redeems it in the app, and the communications are propagated to the *payments service*. Whenever a voucher code is created by the *voucher code service*, the *payments service* is also updated with corresponding voucher code data so that it can acknowledge the code when it is finally redeemed.

Systems Diagram

Below is the systems diagram of the inter-service communications for the digital parking lot system:



Database design:

To maintain the loosely coupled design of the microservices architecture, each service should independently possess its own relevant database. A synchronisation mechanism can be put in place to make sure no inconsistencies occur.

The apps can store the relevant user/historical data on the systems on which they are run.

The *driver accounts service* will have a database containing the details, parking and transaction history of each driver-user (and voucher details if a voucher has been bought through a device).

The *payments service* will contain only certain key user details, their payment information, and all other third-party details necessary to facilitate payments as well as voucher details.

The *camera system service* will contain a database of all vehicle stay information – this will also be stored in the *manager accounts database*, which also will have manager and parking lot data.

The *maps service* database will contain primarily geographic information corresponding with user details, and all the addresses of the parking lots and their respective details.

Lastly, the *authorisation service* will have a database of the user and third-party information necessary to be able to authenticate and authorise users.

Question 4 – System Engineering

The 'Drink Link' app was created to act as an alcohol consumption tracker for patrons visiting a bar. The app would be used by bar staff to ensure no patrons are served if their blood alcohol concentration (BAC or bac) surpasses a certain threshold. Staff have the option of adding and removing patrons, adding and removing each patrons drinks, as well as viewing a patrons blood alcohol status in real time. If the threshold is passed by a patron, there details change colour from black to red.

Running the App

The app uses python 3.8 and can be run in a docker container (preferably Linux systems). First the docker image must be created. Open a terminal in the "Q4 Alcohol Tracker App" directory and run,

```
sudo docker build -t compose-flask .
```

to create the cocker image. The docker container can then be starting by running,

```
sudo docker compose up
```

after which you can open 'http://127.0.0.1:5000' in a browser to view the app (this link will usually be shown in the terminal).

Alternatively, the app can be run from the "Q4 Alcohol Tracker App" directory terminal. First ensuring all required modules from requirements.txt are present within the environment, then simply running,

```
python3 app.py
```

where this time 'http://127.0.0.1:5000' will automatically open in the default browser.

App Structure

The frontend is written in javascript and html, with typical cdn imports for Bootstrap styling and javascript. A cdn import was made to use the typeahead.js and Bloodhound libraries which enable a dynamic typeahead when searching for drinks or patrons for an improved user experience. The backend logic was written in python 3.8. The app uses the python Flask library to create an API with which facilitates front-to-back end communication. All python code is in the same file. A SQLite database was used for simplicity, implemented using the flask_sqlalchemy library.

The SQLite database has five tables:

- Patron: ID, name, sex, bodyweight. Records of all patrons that have been in the bar ever.
- Current: timeIn, name, bloodAlc, ID. Keeps track of the patrons present in the bar.
- Order: orderID, patronID, drinkName, dateTime. Stores details of every order.
- Ingredient: name, ABV. Stores drink ingredient details from thecocktaildb.com for quick look up.
- Drink: name, alcohol_content. Stores drink alcohol content data from thecocktaildb.com for quick look up.

The tables are accessed either directly from the app.py file or using javascript ajax requests. The app uses a combination of SSR and CSR.

App Logic

The blood alcohol content of patrons was calculated using the Widmark formula[1],

$$BAC = \frac{alc}{r \times M} \times 100\% - mr \times t,$$

where alc is the grams of alcohol in the drink, r is the ratio of water-weight to bodyweight of the patron (gender averages: men – 68%, women – 55%), M is the mass of the patron, mr is the patron's rate of metabolising alcohol (human average[1]: $mr = 0.015\%$) and t is the time since the alcohol was consumed. The BAC threshold was chosen to be 0.15%[1].

When a patron orders their first drink, their BAC is set with,

```
t = now() - time_of_last_order
```

Then at regular intervals (chosen to be 5 seconds) the app updates the Widmark formula, where the value of `t` gradually begins to increase, hence resulting in the reduction of the patron's BAC. When a new order is added,

```
BAC = BAC_order1 + BAC_order2
```

where `BAC_order1` account for the time decay thus far. Upon update, `time_of_last_order` is updated to the most recent order. In practice, `time_of_last_order` is looked up by matching the patron ID to the Order table and pulling the last result's dateTime attribute.

Similarly, if an order is removed then the patron's BAC is accordingly updated. The effect of the drink on the patron's BAC (time decay considered) are subtracted from the patrons net BAC. So removing drinks that have been fully metabolised will have no effect on BAC.

Furthermore, if a patron has left the bar but then returns later, their drinks are retrieved and BAC is recalculated.

The alcohol decay updates do not update the patron BAC fields in the Current table. Instead the result is retrieved by the script.js file and painted on the screen where required. This was done to simplify the concurrent flow of the application and avoid bad interleavings It also more easily allows for an existing patron to return to the bar and have their BAC calculated or have the app be launched after being closed for a period with accurately depicted BAC levels for the current patrons. The Current.bloodAlc field is updated only when drinks are added or removed. Otherwise, the backend processes the bloodAlc before the ajax request receives it. A consequence of this design decision is a less natural flow of logic.

The app further tries to optimise search time for drinks. It does this by saving the ingredients and their ABV of each drink looked up on thecocktailsdb.com. It then also saves the drink itself and its alcohol content. When a drink is looked up to be added to the patron, the app first checks if the drink exists in the database, if not it calculates the drink alcohol content by checking each ingredient in first in the database, else in thecocktaildb.com.

Functionality

Upon launching the webapp, a user can perform several tasks:

- **Add new patron**: Submit patron details to add a new patron to the Patron and Current tables. Browser will notify if the form requirements are not met: all fields in form must be filled, ID and bodyweight must be numbers, ID must be unique.
- **Add existing patron**: Use ID to search for existing patron in database to add to Current table. Browser will notify if the form requirements are not met: field must be filled, patron must not exist in Current, ID must be number.
- **View patron details**: Once a patron has been added, their tile will appear on the UI. Click the tile for a modal with patron details and order history to appear.
- **Add a drink**: In the patron modal, search for a drink from thecocktaildb.com and add it. BAC will be updated and a drink will be added to order history in format: drinkID, drinkName, orderDateTime.
- **Remove a drink**: Drink can be remove from patron modal, and BAC is updated.
- **Remove current patron**: Patron tile can be removed. Patron is deleted in Current, but remains in Patron table.
- **Reset all patron data**: Button at the bottom of the window which resets Patron, Current and Order tables.

Furthermore, in the app.py file, global variables RATE and FACTOR can be used to adjust the metabolic rate and an acceleration factor to the rate. Used for testing purposes to see BAC decay happen quickly. Default RATE = 0.015 and FACTOR = 1.

In script.js the global variables LIMIT and UPDATE_RATE set the BAC threshold and the rate (milliseconds) at which the BAC decay is updated on screen respectively. Default LIMIT = 0.15 and UPDATE_RATE = 5000

Possible Improvements

Code readability can be improved by renaming certain functions and variables. Naming convention and consistency can be further enforced too. Some repetitive code areas in the app.py and control.js

files can be made into their own functions. The backend logic or at least certain functions can be separated from the app.py file into a utils.py file.