Notes for JMeter and the entire process:

**Recording data**

*Method:*

- One option was to use the HTTPS, test script recorder. Initial thought is to prefer this method if running my own applications locally, thereby we can hook it up to the docker port and listen to the requests being sent.
- Opted to use the BlazeMeter chrome extension because the site needed to be performance tested is already deployed

*Process:*

- Poor experience using the chrome extension for me. Captured all web calls fine, but didn't capture any API calls.
- Firefox saved my bacon with their HAR option, which I downloaded and converted via the BlazeMeter converter. This came at jus the right time, especially since I was struggling to hit the API endpoints using postman (for some reason).

*Note:*

If doing this again, I would have made it more apparent which calls are API calls and which calls are web ones, the quick checklist is:

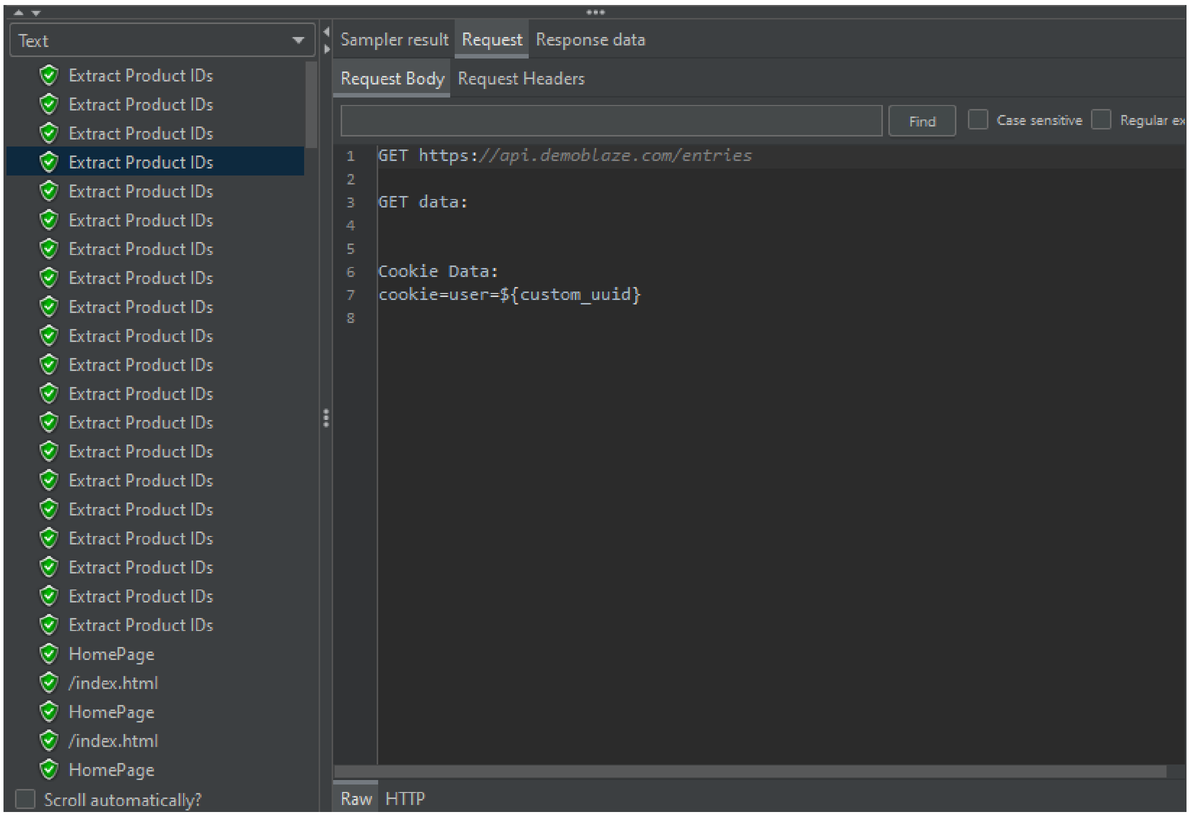
**Learnings**

- Most config elements should be placed at the thread level (https://jmeter.apache.org/usermanual/build-web-test-plan.html#adding_cookie_support). This was surprising because BlazeMeter automatically put them at the Test(?) level.
- Assertions should probably be based on one thing only and be just enough to get the test going (I don't think the intention is to write a Cypress-level test when it comes to attention to detail), to reduce resource requirements.
- A thread group represents a group of users. If we want the same users to perform a different set of actions, we contain it within that group. I think it would be interesting to also learn when to use different thread groups, perhaps users with different roles, or use-cases?

**Execution plan**

*Interesting tidbits:*

Struggled to generate a random cookie for each iteration. Somewhat fixed this by using a post-processor groovy script, but the initial “Extract Product IDs” sampler won’t have the correct cookie:



Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
Extract Product IDs
HomePage
/index.html
HomePage
/index.html
HomePage

```
GET https://api.demoblaze.com/entries

GET data:


Cookie Data:
cookie=user=${custom_uuid}

```

Also had to generate a random “id” for the “/viewcart” endpoint along with randomizing a productid from the initial “/entries” endpoint to simulate a user randomly picking an item.

Did very basic assertions (html for web pages, response content for api calls). I also extracted the other fields of the product (category, description, image etc.), with hopes of asserting them later down the line. This never happened, but I left it in.

Used a gaussian random timer for all web requests to simulate the randomness of a user “thinking” while on the webpage.

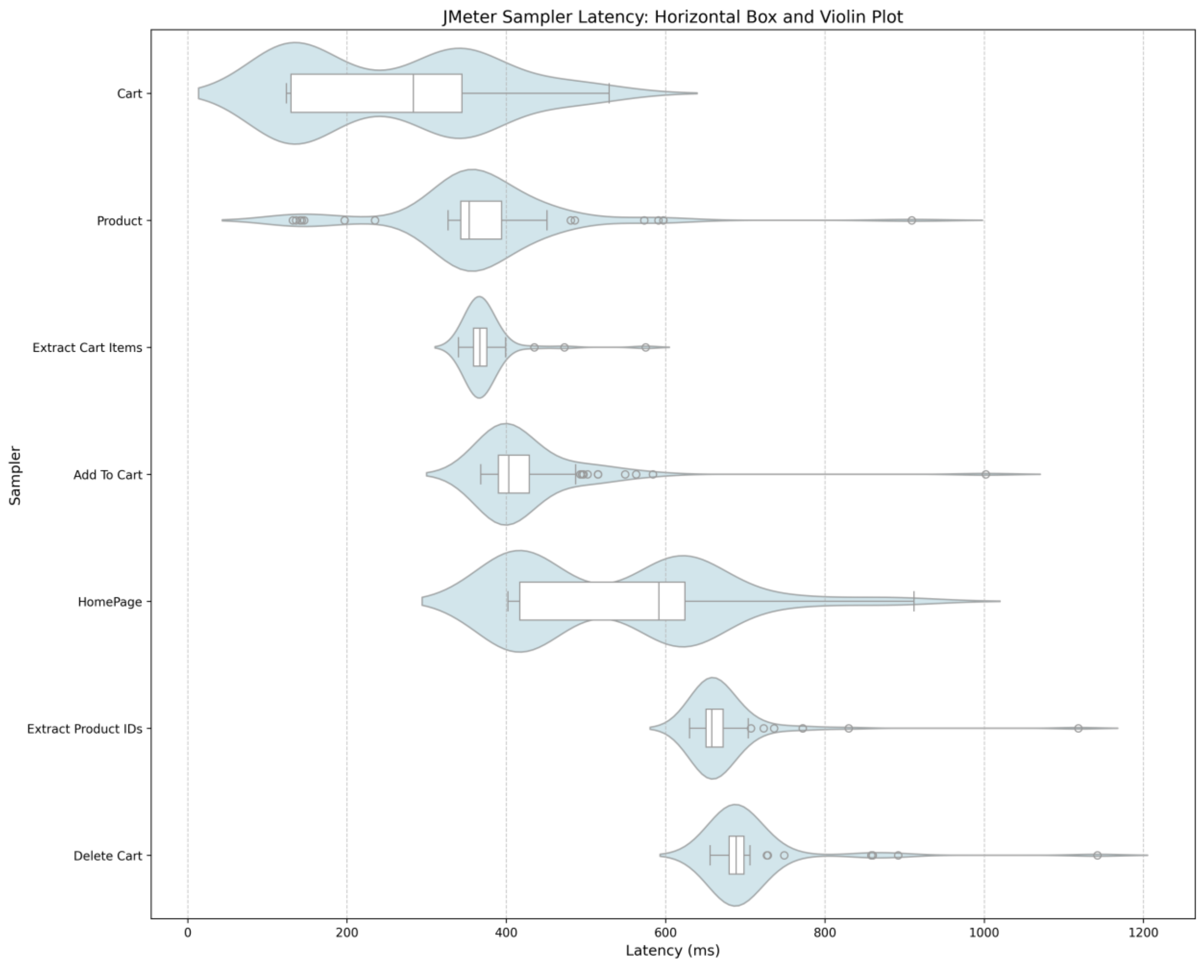
*Debugging:*

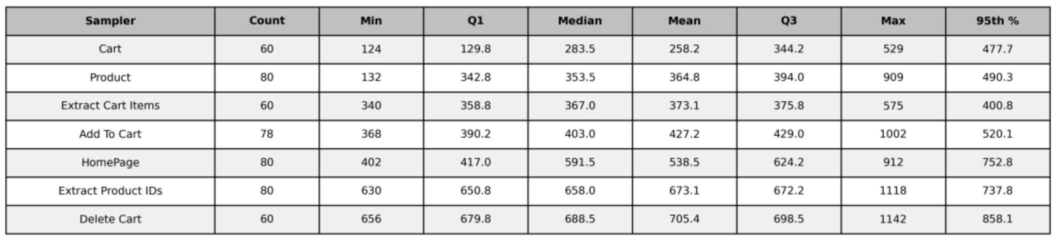
- Debug sampler is good
- Postman is also good for API calls

## Post analysis

I was in the process of doing this before reading the updated assignment, so it’s quite munged and a bit of me doing random things here and there. I pulled out my old stats textbooks, gave them a quick skim and did some analysis on Python. **All the code is done using Claude, which means there is a likelihood that it’s rubbish**. The reason being that I wanted to focus on generating and analyzing the results rather than worrying about syntax for the scope of this assignment.

I ran the JMX file for 5 minutes and used the output for analysis.

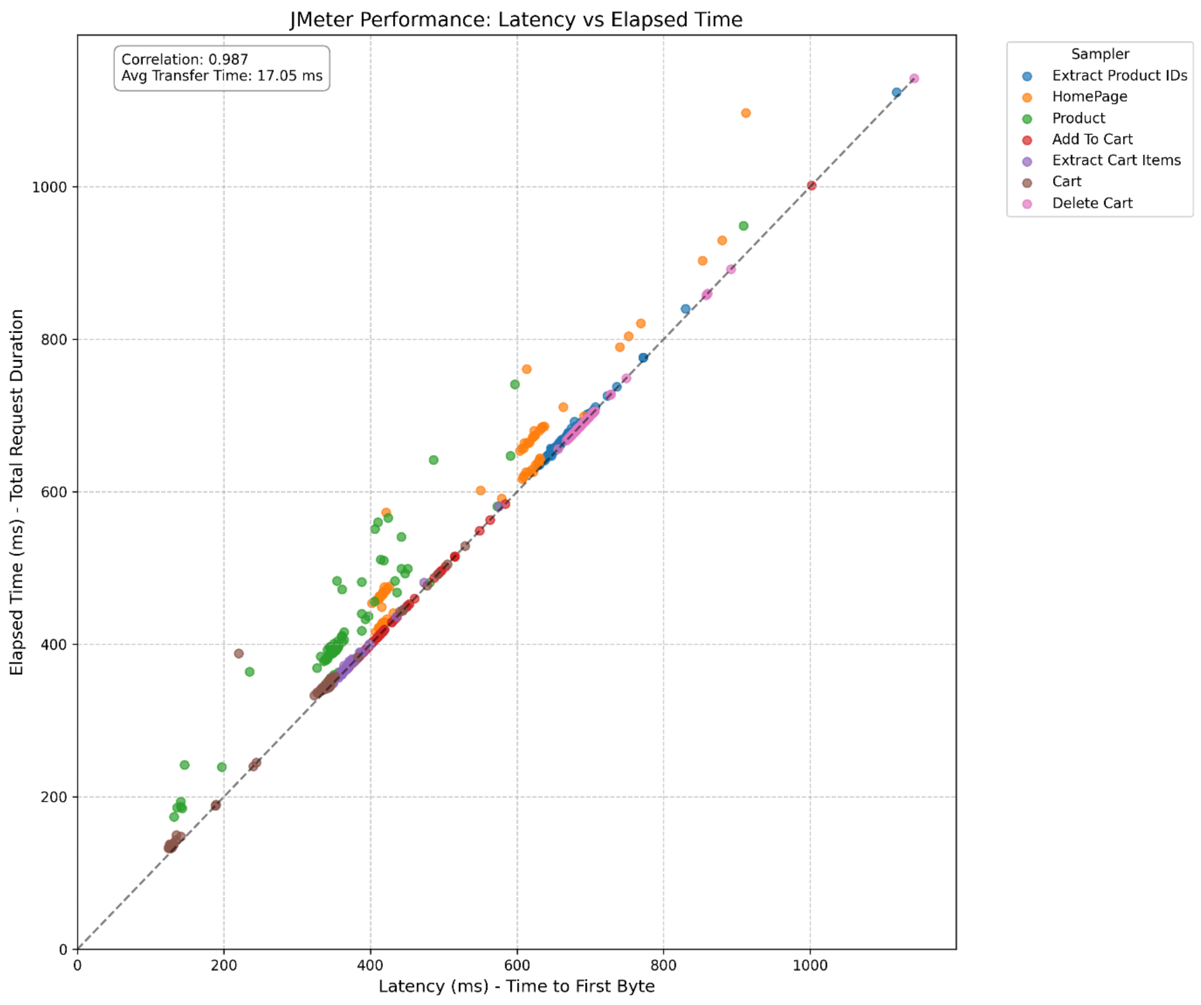




| Sampler | Count | Min | Q1 | Median | Mean | Q3 | Max | 95th % |
|---|---|---|---|---|---|---|---|---|
| Cart | 60 | 124 | 129.8 | 283.5 | 258.2 | 344.2 | 529 | 477.7 |
| Product | 80 | 132 | 342.8 | 353.5 | 364.8 | 394.0 | 909 | 490.3 |
| Extract Cart Items | 60 | 340 | 358.8 | 367.0 | 373.1 | 375.8 | 575 | 400.8 |
| Add To Cart | 78 | 368 | 390.2 | 403.0 | 427.2 | 429.0 | 1002 | 520.1 |
| HomePage | 80 | 402 | 417.0 | 591.5 | 538.5 | 624.2 | 912 | 752.8 |
| Extract Product IDs | 80 | 630 | 650.8 | 658.0 | 673.1 | 672.2 | 1118 | 737.8 |
| Delete Cart | 60 | 656 | 679.8 | 688.5 | 705.4 | 698.5 | 1142 | 858.1 |

Quick points:

- “Cart” and “HomePage” have a bimodal data structure, both being WebAPI calls. Could be due to lack of samples. It would be interesting to investigate why since this may suggest distinct groups (e.g. a cached response group vs a non-cached response group).
- My (uninformed) hunch is to reduce the IQR first for performance improvements because it might signify a lack of consistency somewhere.
- Second would be to investigate some outliers for the API queries and determine whether they are real and expected.



Quick Points:

- Graph compares Latency and Elapsed time.
- My understanding is that latency is the duration between when the request gets sent and the first response is received, while elapsed time is when the request is sent until the complete response is received.
- Since some of the responses are passing back the entire HTML elements, I would investigate those that are inconsistent. For example, the “Cart” (brown) call is usually highly correlated with the Latency, hence that one outlier would be interesting to investigate.