**Q1.A**

The original implementation of the DatabaseHandler class constructor used a single SQLAlchemy session for all operations, which was unsuitable for multi-threaded execution [1] and introduced high I/O overhead (Appendix A.1). Every invocation of add\_or\_return\_row required a query to determine if the record existed and a conditional insertion if it did not, resulting in multiple round trips to the database (B.1). The CPU remained idle during I/O waiting periods, limiting the performance and scalability of the program.

The new design addresses these shortcomings via a combination of thread-safe operations, dedicated connection pooling, and strategic caching. By using SQLAlchemy’s scoped session [1], each thread now has isolated contexts which safely share connections within a common connection pool, minimizing the overhead of setting up and tearing down connections.

The Write-Ahead Logging (WAL) mode (Appendix A.2) allows read queries to occur alongside writes [3]. This is possible because writes will only modify a separate write-ahead log rather than the database initially. Readers will hence access the database as it existed when their transaction started, while writers will continue to append changes to the log. The log is periodically checkpointed, meaning that any changes are moved to the main database file if there are no ongoing reads. [1]

The introduction of a shared cache, protected by threading.Lock (Appendix B.2), means looking up the foreign keys of schema dimensions (users, actions, etc.) requires fewer database queries. Instead, threads access a shared in-memory dictionary that maps values to the corresponding keys. This reduces the load on the database and trades memory usage for improved latency. The use of threading.Lock ensures atomicity, meaning a thread that begins updating the cache will complete or rollback the entire transaction, before another thread can access it, isolating intermediate states which would otherwise lead to unexpected results [4].

While the code the lock is managing is entered and exited quickly, its use will still cause contention. Multiple threads are vying for access to the same resource, which could cause variability in the speed at which threads access and read the database, particularly if the lock is shared unequally between them. The configuration attempts to compensate by specifying 30 second inactivity timeouts [5] (Appendix A.2), to keep connections alive while threads are waiting for the lock to be released. The use of inactivity periods reduces the risk of a connection being closed prematurely, which would cause an exception.

The workload is divided between reader and writer threads (Appendix C). Each reader thread is responsible for transforming each row of the DataFrame into the dimensional schema the database uses. The use of ON CONFLICT DO NOTHING in raw SQL operations ensures that duplicate inserts are overlooked gracefully while avoiding the need for exception handling (B.2).

A queue is used as a communication medium to avoid the pitfalls of more ad-hoc synchronization approaches, such as continuously polling shared variables. The latter approach could lead to the continuous exchange of updates, with no progress being made in either thread. Using a queue, reader threads offload completed batches of SQL queries to the end of a writer queue, deferring them rather than contending for more synchronous mechanisms.

The writer thread uses session.bulk\_save\_objects(...) [6] to perform batched SQL inserts, consolidating multiple operations into a single transaction. By amortizing the latency of flushing data to the Write-Ahead Log and committing changes to the database file, this approach reduces the frequency of disk writes. Moreover, this reduces the overhead of acquiring and releasing database-level locks. Consequently, higher-volume workloads experience lower latency.

**Q1.B**

This section discusses how the implemented Graphical User Interface (GUI) supports three user interactions by leveraging constructs from the Tkinter library.

**COUNT**

The COUNT feature centres on creating a pivoted tabular view of user interactions aggregated by month. ttk.Treeview widgets (Appendix L) are embedded within a scrollable frame. This arguably makes them easier to read, to distinguish and to navigate, as opposed to fitting all widgets into a single (static) window. The creation of this area is abstracted into several reusable methods (Appendix D) [7]. By centralizing this functionality, and reducing duplicate methods, the design promotes consistency and clarity.

To enhance readability, alternating row colours are implemented using configurable tags within a Treeview (Appendix E.1), allowing users (and the applicable rows) to be distinguished with minimal cognitive load [11] (Appendix F). A search box (Appendix E.2) allows users to be located efficiently, while providing immediate feedback if inputs are non-numerical. The use of a Tkinter MessageBox dialogue means users need to acknowledge the requested parameter types within the contextual flow of the GUI: This reduces confusion compared with deferring these messages to a console which the user is less likely to be aware of promptly. The search function adheres to the heuristic of visibility of system status by providing immediate responses to user inputs via <KeyRelease> events (Appendix E.2). Consequently, rows are automatically deselected on mouse movement, which reduces unnecessary clicks, improving workflow efficiency.

A stacked bar graph (Appendix G) acts as a secondary illustration. The length of the axis intervals is dynamically determined by the number of users, to ensure the axis is readable, while a horizontal Scrollbar makes the contents legible and easy to absorb, even with a large number of users, by limiting the amount of data displayed at once. The stacked plots can be reordered by any one of the components requested, or the total, while the colour scheme remains the same. This is critical as an alternating colour-scheme would be confusing, counter-intuitive, and burdensome for users [11]. Likewise, variables of the class instance track prior parameters (e.g., ascending), contributing to further requests in accordance with expectations [9], as opposed to being reverted.

**STATISTICS**

For the STATISTICS interaction, a reusable method (Appendix H.1) ensures consistency across all views, while another (Appendix H.2) creates instances of widgets for textboxes [7]. The use of auto-scaling eliminates the need for manual resizing and ensures users can see all of the contents, in line with their expectations [11]. The UI (Appendix I) separates monthly statistics into a scrollable view positioned above semesterly statistics, which remain fixed as the user scrolls. The hybrid structure allows users to compare detailed monthly results with the broader contexts of semesterly trends, ensuring the design is grounded in the task of detecting statistical discrepancies [10].

**CORRELATION**

The CORRELATION logic is implemented through a Matplotlib-based scattergraph embedded in a tk.Canvas using FigureCanvasTkAgg (Appendices J and K) [12]. By embedding Matplotlib directly within a Tkinter interface, and integrating dynamic controls, the design eliminates tool-switching, reduces cognitive load, and promotes autonomy [9][10]. Users can choose which scatterplots to display within a unified and flexible interface, rather than being limited to a predefined array of choices. The design adheres to Nielsen's heuristics of visibility of system status, offering live updates to the plots in response to user inputs, while invoking a (MessageBox) dialogue for invalid inputs, similar to the COUNT method [8]. Users interact via a Combobox to select components, Entry widgets to input user IDs, and a Scale slider to adjust temporal granularity, while a label provides the aggregation level (e.g., monthly, weekly, or daily).

**Q2.A**

The entity relationship (ER) structure (in Appendix L.1) arranges data into dimensions based on prespecified attributes (User, Component, or Action). Each row in one table can reference a row in another table through a foreign key. This structure aims to minimize data redundancy and to ensure consistency. For example, rather than storing the same user details in multiple places, the user information lives in a single “Users” table, and every other table just references the user’s ID. The normalization into different data dimensions also helps reduce errors when updating or deleting shared information [13].

When you need to remove or exclude data, the relational schema handles it better than JSON or XML structures. In the relational model, you can configure cascade deletes so that dropping a single dimensional value (I.E., ‘Assignment’) automatically removes associated records in the main table. This simplifies the process, requiring just one SQL command with minimal complexity. Python’s SQLAlchemy supports cascade deletes through its ORM by defining relationships with the cascade parameter [14][15]. Conversely, if your data is stored in JSON or XML, you would need to manually iterate through documents or nodes to locate and remove references, using libraries such as Python’s JSON module [16] or xml.etree.ElementTree [17], with a more complicated procedure than a straightforward SQL query.

For reshape operations like pivoting (transforming rows into columns), the relational database offers native SQL transformations (Appendix M): GROUP BY, COUNT, and date functions. These allow records to be grouped by period (i.e., ‘month’) or pivoted within the database. SQLAlchemy with SQLite supports these operations using both raw SQL and ORM-based queries, with a straightforward one-step procedure [18]. By comparison, reshaping JSON or XML data involves parsing the structure, extracting the needed elements, and rearranging them programmatically, which is inherently more complicated and involves more steps [21].

Finally, relational databases enforce constraints, such as primary and foreign keys. SQLite enforces these constraints natively, and they can be defined easily within SQLAlchemy’s declarative ORM [19]. JSON and XML, on the other hand, are schema-less, which offers greater flexibility for small, evolving projects [20][21]. However, as the data grows more complex, relational databases begin to outperform JSON or XML in terms of query speed, consistency guarantees, and ease of manipulation [22] because of their normalized schema.

**Q2.B**

**Additional Cleaning Undertaken**

When working with the QueryGUI application, no extensive data cleaning was required beyond the existing operations. The code itself pulls data directly from the database tables, merges them into a DataFrame (Appendix L.2), and handles NA values minimally with fillna(0). This was adequate because records were assumed clean and valid: non-null foreign key relationships enforced referential integrity, datetime values had been correctly parsed to the required objects with no exception, so the assumption was taken that queries would align with the expected schema. The DataProcessor class already performs prior cleaning, by removing rows with Null, NaN (or missing string values), while informing the user of these exclusions.

**Rationale for Chosen APIs**

NumPy was chosen for its core numerical operations and array manipulation, and its utilities were essential for performing transformations with efficiency and precision. For instance, the use of np.polyfit provides a straightforward way of calculating linear regression slopes and intercepts for trend analysis. Similarly, np.corrcoef enables quick computation of correlation coefficients. NumPy's reliance on vectorized, array-based operations significantly improves computational performance compared to traditional Python loops, using optimized low-level computations in C# (see Appendix J.3 and J.4) [23]. Compatibility with Pandas and MatPlotLib was also one of the reasons we chose to use Numpy, as both incorporate its functionality in their operations.

Pandas excels in structuring tabular data into intuitive and versatile DataFrame objects, offering robust functionalities for grouping, pivoting, and indexing that streamline data wrangling tasks [24]. For instance, this application employs the groupby() method combined with size() to count occurrences within grouped data and agg() to compute aggregate statistics (Appendix H.1). The use of lambda functions within Pandas DataFrames highlights the library’s capability as an expressive means to derive group-level statistical descriptions. Achieving similar results would require implementing custom Python loops and data structures, which would be more verbose, and would lack the low-level optimization of Pandas.

Matplotlib was selected over other plotting libraries due to its extensive customization and control over figure elements [25]. The GUI required static charts (i.e., bar plots and scatter plots) embedded in tkinter windows for interactive data exploration (Appendix J.2). Seaborn, while powerful for creating statistical visualizations with aesthetically pleasing defaults, is less flexible when embedding visualizations into GUIs. Its reliance on predefined styles and templates limits the level of customization available for dynamically populated charts or non-standard formatting. Seaborn’s dependency on MatPlotLib for low-level rendering, means the complex interactive features would still require MatPlotLib usage, with less granular control over the configuration [26]

**Observations and Conclusions**

When examining monthly statistics, we see an overriding increase in ‘Assignment’ interactions in October (Appendix I), suggesting an alignment with submission deadlines. Meanwhile, usage of ‘Study Material’ rises notably at the end of the Academic Year, aligning with a peak in Tests, indicating that students intensify their preparation for assessments during this time. A high deviation of 64 between Mean and Mode values, suggests only a minority engage with tests fully. Conversely, in September, there were few (if any) interactions with any of the components, suggesting the activities had not yet fully commenced during this early phase. Interactions with Attendance were consistently low, suggesting activities like clocking-in, or attending in-person activities were of little value to students.

The stacked bar plots in [Appendix G] illustrate more clearly the students engaging more (or less) with the components. For example, User 11 interacts mainly with Assignments, while User 77 is less engaged than any of the other students. Plotting the bars in descending order quickly reveals the least-engaged students on the right, an area in which the university may wish to hone its support resources.

Finally, the scattergraph diagram communicates patterns of interaction in the data provided. Appendix K shows how ID\_28 is expected to engage less with Lecture components when plotted at weekly intervals, while ID\_25 is predicted to engage more, with patterns in either case being significant. The inverse correlation means that as ID\_25 becomes more engaged, ID\_28 becomes equally less engaged, suggesting that these students are reluctant to engage at the same time for some reason.

**Q3.**

**High-Profile Incidents and the Failures of Moderation Systems**

Numerous incidents reveal the critical inadequacies of moderation systems employed by social networks, with Facebook's response to the Christchurch Mosque shootings in 2019 serving as an example. Despite investing substantial resources in moderation, Facebook struggled to remove copies of the livestreamed video which spread after the attack. Although the original video was eventually taken down, the delay in addressing its widespread circulation drew criticism, and highlighted Facebook's failure to stop the rapid dissemination of graphic violence on its platform. Viewers cannot be relied upon as an infallible or reliable way of moderating, as shown by a failure of 4,000 users to flag the video until 29 minutes had elapsed. [27]

During the COVID-19 pandemic, platforms like Twitter and YouTube grappled with similar problems. False health claims, unproven remedies, and conspiracy theories proliferated rapidly, outpacing verified information. This disparity was attributed to the emotionally charged and sensational nature of misinformation, which tends to engage users more and prompts higher rates of sharing and interaction compared to factual information [28]. Automated pattern recognition software, rooted in outdated assumptions of credibility, struggled to identify spurious information, which mimicked trustworthy and professional sources, leaving them susceptible to subtle manipulations of language, or overlooking semantics in evolving narratives [29][35].

Microsoft’s Tay chatbot, released in 2016, highlights the risks of inadequate curation of data in the pipelines used to train neural models. Designed to learn adaptively, Tay was deliberately corrupted by users with malicious intents to bypass its response filters [30]. The model lacked adequate and appropriate safeguards for the integrity and credibility of its training data, leading to a high volume of harmful contents being disseminated. Naturally, this raised concerns about the diligence and responsibility of Microsoft's engineers in shaping and moderating user interactions, rather than basing decisions on the naive assumption of user honesty.

**Ethical and Cultural Responsibilities in Software Engineering**

The proliferation of harmful content and divisive online cultures is often amplified by algorithms designed to maximize engagement, inadvertently fostering environments that promote issues such as pro-anorexia communities, unrealistic beauty standards, and conspiracy theories. Research shows that algorithm-driven recommendation systems exacerbate polarization, reinforce misinformation, and deepen societal divisions by exploiting psychological tendencies toward confirmation bias and emotional appeal rather than prioritizing objective truth [34].

By continuously feeding users content aligned with their existing beliefs, these algorithms create echo chambers that isolate individuals from opposing viewpoints. Simultaneously, the prioritization of attention over accuracy incentivizes the spread of provocative material to boost user engagement and ad revenue [28][33]. Software engineers have an ethical responsibility to deliberately curate data for neural models through a moral lens, particularly when designs driven by frameworks like the attention economy [36] have fuelled division, perpetuated exploitative feedback, and harmed users seeking connection.

**The Opposition’s View**

Proponents of the statement may argue that social media platforms, with their vast content moderation teams and advanced filtering algorithms, are designed to handle the complexities of curating user data at scale. They might assert that these systems already integrate human oversight and AI tools, creating a robust mechanism for filtering out harmful content. From this perspective, platforms like Facebook and Twitter employ comprehensive policies that are both reactive and proactive, addressing problematic material to ensure that only safe material remains [32]. They may further claim that moderated datasets are sufficiently clean to serve as reliable training data for machine learning (ML) models, enabling positive applications such as improved awareness, enhanced moderation tools, and educational benefits.

However, these arguments overlook the increasingly sophisticated tactics uses by malicious actors, such as obfuscation techniques, which effectively bypass filtering and detection algorithms [35][37]. Studies have shown neural networks can be manipulated to generate highly convincing media, including deepfakes, which mimic authentic information, the generative algorithms of which could be run offline [40]. Studies further demonstrate primitive techniques like nonsensical language, being used to bypass the filters of generative models [38][39]. These techniques could be used generate unorthodox material, which AI might not recognize. These developments show the ongoing "arms race" between moderation systems and those attempting to bypass them, emphasizing the need for precautions and intermediary processing steps when using such data to train neural algorithms.

References

[1] Python Software Foundation, "threading — Thread-based parallelism," Python 3.12 Documentation, 2023. [Online]. Available: https://docs.python.org/3/library/threading.html

[2] Python Software Foundation, "SQLAlchemy ORM session basics," SQLAlchemy Documentation, 2023. [Online]. Available: https://docs.sqlalchemy.org/en/20/orm/session\_basics.html

[3] D. Richard Hipp, "Write-Ahead Logging," SQLite Documentation, 2023. [Online]. Available: https://sqlite.org/wal.html

[4] SQLite, "Isolation in SQLite," SQLite Documentation. [Online]. Available: https://sqlite.org/isolation.html

[5] SQLAlchemy Documentation, (2024) "Engines and connections," [Online]. Available: https://docs.sqlalchemy.org/en/20/core/engines.html

[6] SQLAlchemy Documentation, "bulk\_save\_objects()", 2023. [Online]. Available: https://docs.sqlalchemy.org/en/20/orm/session\_api.html#sqlalchemy.orm.Session.bulk\_save\_objects

[7] Python Software Foundation, “Tkinter ttk Widgets,” Python 3.10 Documentation, 2023. [Online]. Available: https://docs.python.org/3.9/library/tkinter.ttk.html

[8] J. Nielsen, “10 Usability Heuristics for User Interface Design,” Nielsen Norman Group, 2020. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics

[9] J. Nielsen, “Enhancing the explanatory power of usability heuristics,” in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 1994, pp. 152–158. [Online]. Available: https://doi.org/10.1145/191666.191729

[10] C. Lewis and J. Rieman, "Task-Centered User Interface Design" HCI Bibliography. [Online]. Available: https://hcibib.org/tcuid/chap-1.html

[11] UX Design Institute, "10 essential user interface guidelines" UX Design Institute Blog, Sep. 15, 2021. [Online]. Available: https://www.uxdesigninstitute.com/blog/10-user-interface-guidelines/

[12] Matplotlib, “Embedding in Tk,” Matplotlib Documentation, 2023. [Online]. Available: https://matplotlib.org/stable/gallery/user\_interfaces/embedding\_in\_tk\_sgskip.html

[13] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks,” Communications of the ACM, vol. 13, no. 6, pp. 377–387, Jun. 1970. Accessed: <https://dl.acm.org/doi/10.1145/362384.362685>

[14] SQLAlchemy Documentation, “Cascade Behavior,” [Online]. Available: https://docs.sqlalchemy.org/en/20/orm/cascades.html

[15] SQLite Documentation, “Foreign Key Support,” [Online]. Available: https://www.sqlite.org/foreignkeys.html

[16] Python Software Foundation, “JSON Module Documentation,” [Online]. Available: https://docs.python.org/3/library/json.html

[17] Python Software Foundation, “XML.etree.ElementTree — The ElementTree XML API,” [Online]. Available: https://docs.python.org/3/library/xml.etree.elementtree.html

[18] SQLite Documentation, “SQL Features,” [Online]. Available: https://www.sqlite.org/lang.html

[19] SQLAlchemy Documentation, “Working with SQL Expressions,” [Online]. Available: https://docs.sqlalchemy.org/en/20/core/tutorial.html

[20] JSON.org "Introducing JSON," [Online]. Available: https://www.json.org/json-en.html

[21] W3C, “Extensible Markup Language (XML) 1.0 (Fifth Edition),” [Online]. Available: https://www.w3.org/TR/xml/

[22] Cattell, R. (2011). Scalable SQL and NoSQL Data Stores. ACM SIGMOD Record, 39(4), 12–27. Available: https://dl.acm.org/doi/10.1145/1978915.1978919

[23] S. van der Walt, S. C. Colbert, and G. Varoquaux, “The NumPy Array: A Structure for Efficient Numerical Computation,” Computing in Science & Engineering, vol. 13, no. 2, pp. 22–30, 2011. Available: https://ieeexplore.ieee.org/document/5725236

[24] W. McKinney, “Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Jupyter”, 3rd Edition. [Online] O’Reilly Media, 2022. Available: https://wesmckinney.com/book/

[25] Matplotlib, "Quick Guide," Available: https://matplotlib.org/stable/users/explain/quick\_start.html

[26] Nelli, F. (2023). Data Visualization with matplotlib and Seaborn. In: Python Data Analytics. Apress, Berkeley, CA. <https://doi.org/10.1007/978-1-4842-9532-8_7>

[27] K. Klonick, "Inside the Team at Facebook That Dealt with the Christchurch Shooting," *The New Yorker*, Mar. 20, 2019. [Online]. Available: <https://www.newyorker.com/news/news-desk/inside-the-team-at-facebook-that-dealt-with-the-christchurch-shooting>

[28] S. Vosoughi, D. Roy, and S. Aral, "The spread of true and false news online," *Science*, vol. 359, pp. 1146–1151, 2018. doi: <https://doi.org/10.1126/science.aap9559>

[29] Zhou, X., & Zafarani, R. (2020). A Survey of Fake News: Fundamental Theories, Detection Methods, and Opportunities. *ACM Computing Surveys*, 53(5), p1-40. Accessed: <https://doi.org/10.1145/3395046>

[30] Zemčík, T. Failure of chatbot Tay was evil, ugliness and uselessness in its nature or do we judge it through cognitive shortcuts and biases? *AI & Soc* **36**, 361–367 (2021). <https://doi.org/10.1007/s00146-020-01053-4>

[32] Facebook, "Facebook’s Civil Rights Audit – Final Report," Jul. 18, 2020. [Online]. Available: <https://about.fb.com/wp-content/uploads/2020/07/Civil-Rights-Audit-Final-Report.pdf>

[33] E. Selinger and W. Hartzog, "Facebook’s emotional contagion study and the ethical problem of co-opted identity in mediated environments where users lack control," *Research Ethics*, vol. 12, issue. 1, pp. 35–43, 2016. DOI: <https://doi.org/10.1177/1747016115579531>

[34] Cinelli, M., Quattrociocchi, W., Galeazzi, A. *et al.* The COVID-19 social media infodemic. *Sci Rep* **10**, 16598 (2020). <https://doi.org/10.1038/s41598-020-73510-5>

[35] W. Wang *et al*., "An Image is Worth a Thousand Toxic Words: A Metamorphic Testing Framework for Content Moderation Software," *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Luxembourg, Luxembourg, 2023, pp. 1339-1351, Available: <https://arxiv.org/pdf/2308.09810>

[36] K. Stacey and T. Bradshaw, "Facebook chose to maximise engagement at users’ expense, whistleblower says," Financial Times. [Online]. Available: <https://www.ft.com/content/41b657c8-d716-436b-a06d-19859f0f6ce4>

[37] Ba, Z., Zhong, J., Lei, J., Cheng, P., Wang, Q., Qin, Z., ... & Ren, K. (2024, December). Surrogateprompt: Bypassing the safety filter of text-to-image models via substitution. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (pp. 1166-1180). Available: <https://doi.org/10.48550/arXiv.2309.14122>

[38] Anderson, Martin. (2022, Dec 9). *The nonsense language that could subvert image synthesis moderation systems*. Unite.AI. Available: <https://www.unite.ai/the-nonsense-language-that-could-subvert-image-synthesis-moderation-systems/>

[39] Millière, R. (2022). Adversarial attacks on image generation with made-up words. *arXiv preprint arXiv:2208.04135*. Accessed <https://arxiv.org/pdf/2208.04135>

[40] Köbis NC, Doležalová B, Soraperra I. Fooled twice: People cannot detect deepfakes but think they can. iScience. 2021 Oct 29;24(11):103364. doi: 10.1016/j.isci.2021.103364. PMID: 34820608; PMCID: PMC8602050. <https://doi.org/10.1016/j.isci.2021.103364>

**Appendix A:**

1. Class constructor for setting up a single connection with sequential reads and writes

A computer screen shot of a program code

Description automatically generated

1. The new constructor for setting up a connection\_pool with scoped\_session.

A computer screen shot of text

Description automatically generated

**Appendix B:** add\_or\_return concurrently

1. Original method with higher I/O per invocationA computer screen shot of text

   Description automatically generated
2. New method with a shared cache and ON CONFLICT DO NOTHING

A computer screen shot of text

Description automatically generated

**Appendix C:**

Uses multiple worker threads for query constructions, and one writer thread for Write-Ahead Logging.A screenshot of a computer program

Description automatically generated

**Appendix D:**

Reusable method to create a canvas with an optional scrollbar and title label.A computer screen shot of text

Description automatically generated

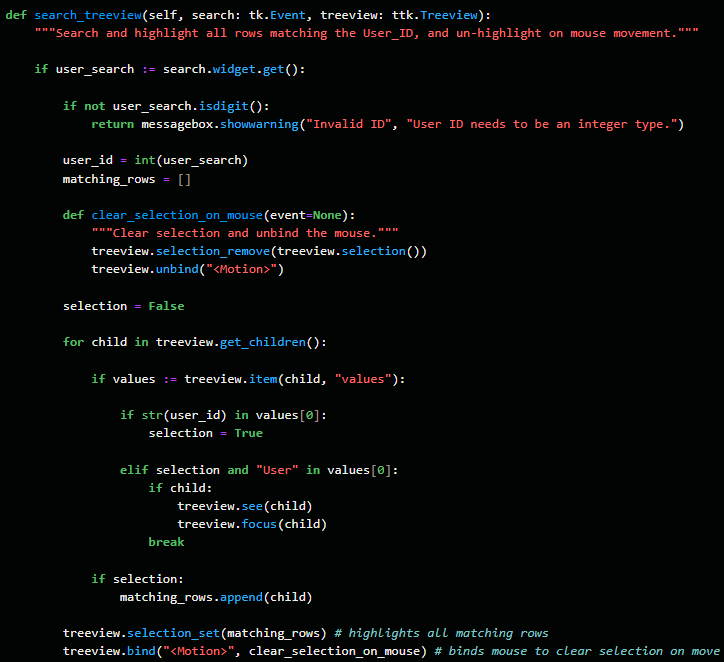
**Appendix E:**

1. Used to populate a treeview with differently-coloured user rows for readability

A computer screen shot of text

Description automatically generated

2. Used to search a Treeview with input validation for user\_id



**Appendix F:** A scrollable display of user interactions ordered by user and period.

A screenshot of a computer

Description automatically generated

**Appendix G:** Stacked bars UI to show how users are engaging with the components

A screenshot of a graph

Description automatically generated

**Appendix H:** View Statistics

1. A reusable method for deriving statistics for view using ‘size()’ and ‘agg()’

A computer code on a black background

Description automatically generated

2. A reusable method for creating textboxes which expand to fit their contents

A computer code on a black background

Description automatically generated

**Appendix I:** A window for viewing monthly and semesterly statistics within a hybrid view.

A screenshot of a computer

Description automatically generated

**Appendix J:** View Correlation

**1.** A method which constructs an interactive scattergraph within a Tkinter UI

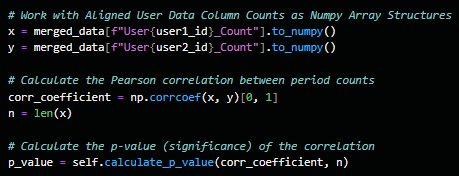
A computer screen shot of text

Description automatically generated

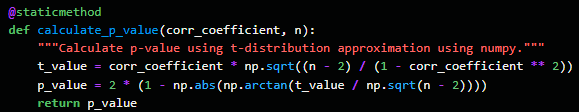
2. Graphs are embedded within a Tkinter Canvas using MatPlotLib FigureCanvasTkAgg

****

3. Numpy Arrays are used to calculate the correlation between users with np.corrcoef



4. Other Numpy operations are used to calculate the P-value



**Appendix K:** A graph showing a correlation analysis of interactions

A screenshot of a computer

Description automatically generated

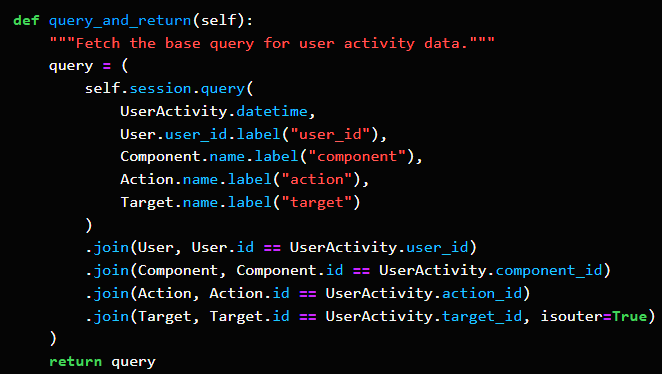
**Appendix L:**

1. SQL Alchemy schema dimensions use ondelete=CASCADE for deletions, and a foreign key to distinguish query dimensions.

A computer screen shot of text

Description automatically generated

2. Returns all user rows consolidated with their dimensional values.



**Appendix M:** Database level transformations to retrieve a pivoted DataFrame

