Universität St.Gallen

Integrative Master's Project (IMP) 2022

Infrastructure for and Processing of Behavioral and Physiological Data Streams

Jonas Hermann, Kay Erik Jenß

January 14, 2023

1 Introduction

Software development can be a challenging task. It includes many different aspects such as writing, reviewing and adapting code from other developers. This requires an extensive set of cognitive skills such as knowing, comprehending, deciding or problem solving [12], making programming a highly demanding and complex task. It is therefore not surprising that most errors in software can be directly related to a failure in the cognitive process of the developer [13]. The probability for such errors heavily depends on the current mental state of the programmer. While the same person might work without any relevant errors when he is fresh, the very same one probably tends to make more mistakes when he is overworked. Therefore, systems have been developed in that can detect these critical cognitive states. In the context of software development, they have been used in the past to assess the quality of code reviews [12]. Building upon the work of Sorg et al. [23] our vision for this project was the following: We want to develop an architecture for a near real-time analysis of cognitive load during code comprehension. To be as little invasive as possible, we only want to use and eye tracker, since other devices such as an EEG revealed usability problems in the past [21]. The goal is to provide an in detail understanding of the difficult parts in a code while it is read by the user and provide adequate support. For this, we use biometric measures that can be inferred from eye tracking data. There are many applications from such a system, ranging from online assistance during code reviews to on-demand help in the development process.

2 Related Work

This section introduces the theoretical background of this project. This includes related work regarding biofeedback systems in the context of software development and other industries. Besides that, the theoretical foundation of metrics for cognitive load is provided.

A broad range of biometric measurements has been investigated in the past to determine a person's cognitive state. Some of them have been applied to infer a programmer's mental load while working on code snippets. Radevski et al. [21] proposed a system that uses an EEG to infer developers productivity in real time. The goal is to help managers to assign appropriate tasks according to every developer's knowledge. Moreover, the system should aid the programmer to take sufficient breaks and manage stress. The authors decided to use EEG, as it provides rich data that can be used to infer a subject's current cognitive state. They concluded that EEG is a good proxy for deriving stress but wearing the device introduced comfortability problems with the participants of the study.

Parnin [20] investigated the potential of electromyography(EMG) to decode subvocal utterances during programming to determine the cognitive load. Sub-vocal utterances are electrical signals that are sent to tongue, lips or vocal cord during the performance of complex tasks. The author found that subvocalizing correlates with the comprehension of more complex code. Moreover, he found that participants were especially subvocalizing during the editing of important code parts. The results suggest that this technique might be a suitable measure for identifying complex parts of a code.

Nakagawa et al. [17] used Near Infrared Spectroscopy (NIRS) to measure programmers cerebral blood flow during code comprehension tasks. The authors conducted an experiment with an easy and a hard piece of code. They found that 8 of 10 subjects of the experiment exposed an increased amount of oxygenated haemoglobin during the comprehension of highly obfuscated code, suggesting that this could be an indicator for menatal load.

A paper by Fritz [16] suggests biometric measurements such as heart rate variability to automatically detect difficulties a programmer experiences while working on a part of the code. These difficulties are seen as a proxy for potential errors. The intention of the system is to replace commonly used code reviews which are more labour intensive. The author concludes that biometrics are a suitable to determine the perceived difficulty of code elements and identify parts that are more error prone.

While the approaches used to determine cognitive effort during programming tasks differ, the majority leverages eye tracking data to tackle this. Bednarik et al. [3] conducted an experiment to understand the development of program comprehension strategies. The authors showed Java code alongside a visualization of the execution flow to novice and experienced developers. The participants were assigned to comprehend the given program first and write a short summary of what the program does after a limited amount of time. Among others, an eye tracker measured the amount of time the subjects looked at various areas of interest. The authors concluded that eye tracking can help to understand the

behaviour and cognitive processes of programmers.
Similarly, Crosby et al. [6] conducted an experiment where subjects were presented with algorithms written in Pascal. The experiment was designed to examine the influence of programming experience on how subjects comprehend a complex piece of code. To answer this research question, an eye tracker was used to scan patterns of eye movements for two differently skilled subject groups.
Rodghero et al. [22] proposed a system leveraging eye tracking to provide code summaries. The authors aim at identifying the relevant keywords programmers view as important when summarizing source code. Therefore, they exposed 10 professional Java programmers to code snippets and asked them to write a short summary of the code afterwards. By tracking their eyes, the relevant parts for the subjects could be identified. The results are used to build and evaluate a novel summarization tool.
Fritz et al. [9] evaluated several psycho physiological sensors(eye tracker, electrodermal activity sensor and electroencephalography) to predict the perceived difficulty of a programming task. Using an Naive Bayes Classifier, they could predict task difficulty with 64,99% precision for a new developer and 84,38% for a new task. They found that the classifier could be improved by solely using data from the eye tracker or by leveraging a sliding window.
Hijazi et al. [12] introduce iReview, a system that uses eye movement dynamics and Heart Rate Variability to assess the reviewer's comprehension of the code he is currently reviewing. iReview evaluates the quality of each review and helps to identify the parts of the code that have not been reviewd well. The intention is to counteract the disadvantages of modern software reviewing processes, where often only one reviewer is in place to be more cost efficient. However, this person is a single point of failure if his review is not of high quality due to factors like stress or distraction. iReview was able to predict a low quality review with an accuracy between 75% and 87% for medium to complex programs.
Our work presented in this report directly builds upon Sorg et al. [23]. The authors used eye fixation features to infer cognitive load during code comprehension. They aim at a fine grained analysis of code to be able to precisely pinpoint the critical party of the code that lead to high mental load during comprehension. The overarching goal is to be able to provide targetted support to developers to prevent them from committing errors.

In order to derive cognitive load, Duchowski et al. proposed the Low/High Index of Pupillary Activity (LHIPA) [7]. This metric is calculate based on wavelet decomposition of the pupil diameter in order to derive a measure. This has been proven to be an effective way, more so than other approaches, such as deriving the cognitive load from the blink rate [4]. LHIPA as a ration of the low to high frequency results in the reverse effect of interpretation, where a smaller LHIPA value corresponds to a larger cognitive load. However, LHIPA was constructed and testes as an off-line metric, meaning that the data was analyzed after the experiment had been run. This has the benefit, that LHIPA can utilize a longer signal duration for analysis, resulting in more bands available for analysis, but

making it not suitable for real-time use [7, 14]. Jayawardena et al. proposed the a metric called Real-time IPA (RIPA), utilizing a Savitzky-Golay filter to achieve a low- and high-frequency analysis of the signal directly and in parallel [14]. This approach seems promising, but features tuning parameters, such as the buffer length and filter threshold, which affect the resulting metric. However, once configured and tuned, it might be a viable consideration for usage in the real-time setting, this project is aiming at.

3 Requirements & Problem Definition

Upon the start of the project, three main requirements for the implementation of a neuroadaptive coding assistance tool were defined.

Sorg et al. [23] suggest a system that analyses the cognitive load after the code comprehension. In our project the challenge is to perform this analysis of incoming gaze data as close to real-time as possible. This is needed as the goal is to provide useful assistance during the process of dealing with code. If this is delayed by too much, the user potentially moved away from the relevant part of the code already. Therefore, the feedback should be provided as soon as possible, but a latency of a few seconds is acceptable in this scenario.

The second requirement of the system was to leverage metrics that are able to measure cognitive load based on the raw data from an eye tracker. These metrics should be computed as efficiently and quickly as possible to handle the large amount of data produced by the eye-tracker and to satisfy the real time requirement described above. For this, existing metrics such as LHIPA and RHIPA presented above can be used [7, 14] .

Third, the software architecture must be designed in a way that allows for the easy addition of new metrics and modalities over time. This allows for usage of the basic infrastructure with ongoing research, but also allows to increase the validity of the output by enriching the decision with multiple indicators for cognitive load. Based on the identified requirements, we are providing a solution, which comes as close to solving all of these points, as possible.

4 Process

During this project, we used an iterative approach following the Build-Measure-Learn cycle. This allowed us to quickly adapt to upcoming challenges and learn quickly. In this chapter, our three iterations and their outcomes are discussed.

4.1 Iteration 1

Starting out with the project an initial implementation was present. This solution can be found in the nassy repository[1], including instructions on how to get an instance running.

Build As the project's starting point was provided, the initial build phase was concerned with running the existing project. After discovering, that there are issues with the Apache NiFi docker image, so that they are not able to run on ARM-architecture M1 chips from Apple, the project setup was done on a Linux machine, where the images ran without issues.
However, the NiFi instance, created from the provided template, did not run. This was discovered after loading the template and attempting to send data to the corresponding endpoint. As the endpoint was always returning a status code of 200, this took quite some time to debug but was supported by the visual interface NiFi offers with a visualization of queues between the activities. Coarse-grained the application consisted of two parts, which were split up after converting the data into CSV format and writing it into an SQLite database. From there, the first execution thread was using DCAP and the included model in order to calculate one metric of cognitive load. The second thread of execution continuously reads from the database in irregular, but very short intervals and always retrieves the data from the last minute. This simulated the sliding window, which is useful to create a continuous value of LHIPA. Based on the fact, that based on the queues, DCAP seemed to cause the issues, accompanied by the fact, that initially the focus was not supposed to be on the ML model, the first thread was removed in order to simplify the application and ensure that it is running.
As NiFi does not allow for windowed aggregations, we were required to initially keep the workaround using SQLite as an intermediary. From there the data would now still be read and LHIPA calculated. This result would then be published via a WebSocket as before. Additionally, we introduced a counting metric, which took the data, of which line of code was gazed upon, and counted a general sum of gazes aggregated by line. This metric was introduced based on the eye-mind hypothesis, stating that there exists a strong correlation between where one is looking and what one is thinking about [15]. This metric was also published on the same WebSocket.
In addition to the changes made to the processing architecture, a new custom front-end was created, which reacted to the incoming metrics on the WebSocket

[1] https://github.com/ics-unisg/nassy

and displayed it to the user. It was needed, as the existing front-end was not responding to incoming data on the WebSocket. This new interface was created using JavaScript, React.js and Chart.js and was largely aimed to imitate the existing functionality in displaying pupil dilation. It further showed a bar graph comparing the newly introduced line count metric with the option to hide line 0. This was important, as any gazes, which could not be mapped to a line because the software was not able to map it, were assigned to line 0. Filtering this out, showed only the gazes that were assigned to a specific line in relation.

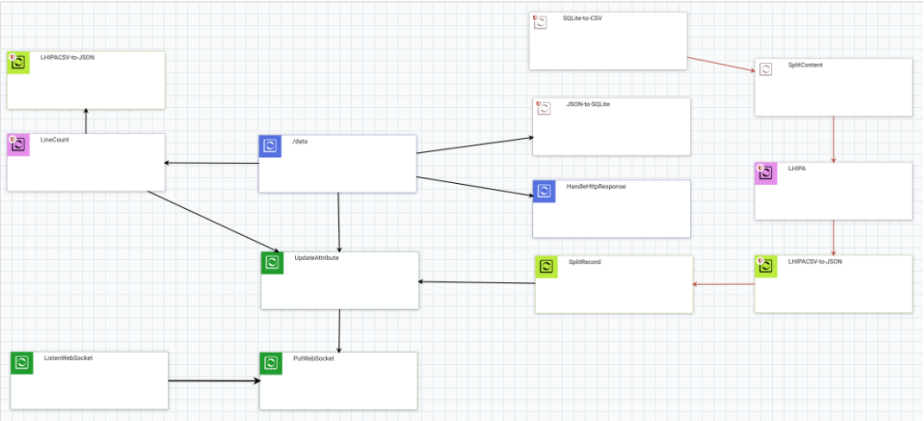

Fig. 1. Screenshot of the NiFi Application after applying the described changes

Measure As we wanted to test the performance of the system under the aspect of real-time performance, we tested a setup, where data was sent by a script in the same interval, that real data would be delivered. We then measured the difference in time between the last of the 4000 records being sent and the last measure of the line count being published to the WebSocket and then how fast the front-end displayed the new information. While the front-end stayed consistent at around 200 ms difference from when the event was published until it was displayed, the NiFi processing took significantly longer with 300 seconds. In order to identify the reason for the latency, we ran additional tests, which only mirrored the records received by NiFi to the WebSocket output after appending a timestamp attribute to them, an operation assumed to run in constant time for every record. But even in this setting, for the amount of 4000 records sent, there was a significant difference between when the last record was sent by the script and when NiFi output the last record of 180 seconds. This made it obvious, that the issue must lie somewhere in NiFi itself, as this simple setup with few steps already had issues to stay in sync, which was only worsened by the additional complexity of calculating LHIPA and counting the lines.

Learn After getting the project set up, we were able to discover several drawbacks of the existing system. We addressed some of them, but very heavily weighted arguments remain. One significant issue is latency. We observed Nifi introduce a very high latency in the processing of the raw data. This can be a major issue when analyzing data, preventing real-time analysis when working with large volumes of data. As NiFi uses standard in- and output for communication between activities, this introduces additional latency, causing the queues to grow. Further, NiFi is keeping so-called flow-files, which are written to the file system. On the scale that NiFi is supposed to operate, this introduces a non-negligible amount of operations and lag therefore. Another problem with using Nifi is that the data may not be kept in the correct order within the Nifi workflow. We measured this by sending 10 numbered data elements, which had different orders in the queues and the output. This is a significant issue, as the order of the data is important for the analysis being performed. Further, NiFi does not support windowed aggregations. This is another important learning which leads us to change the underlying architecture, as the current implementation does not guarantee the exact amount of outgoing results as events go in, as the activity within NiFi just reads data on a schedule. This schedule might be in sync with the rate at which data is received, but the irregular processing times might skew this, resulting in the windowing workaround missing single data points. Finally, we observed consumer leakage, where data is lost due to the lack of processing speed and queues overflowing. After a single queue amassed more than 10000 elements in it, data is just disregarded, resulting in missing data.
Overall, while Nifi might be powerful software architecture for data processing, it is important to consider these drawbacks. Especially the data loss and no guarantee about the order of items are the biggest learnings we discovered, leading us to look for other alternatives.

4.2 Iteration 2

As discussed in the previous section, we found that the existing system architecture is not suitable for fulfilling the specified requirements, especially the near real time performance. Therefore, we decided to do a complete reengineering of the system. The figure below provides an overview over the architecture we sugggest. The prerecorded gaze data is replayed to a Kafka topic via a Python script. To mimic the frequency of the eye tracker, this is done every 8 ms. Spark is reading from that topic and performs the counting of the code lines like in iteration one. LHIPA is not computed in this iteration. The reasons for this are extensively discussed in the subsequent chapter. The result is again outputed into another Kafka topic. This stream is read by a Python script, which writes the data into a PostgreSQL database. This database is queried by Grafana and the data is visualized. In this section, the reasoning behind the individual components is given. Moreover, its performance is compared to the previous architecture and the learnings of this cycle are discussed in further detail.

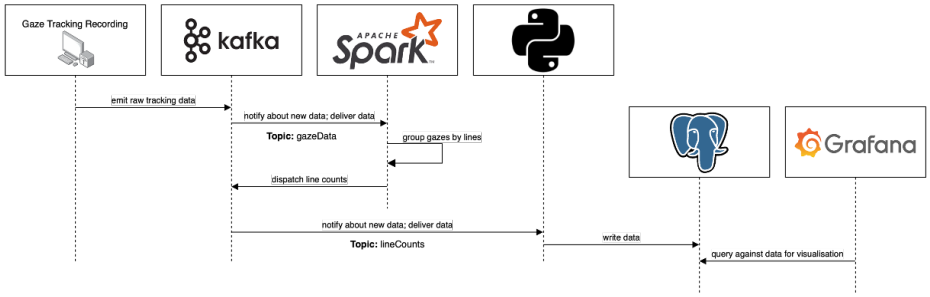

Fig. 2. Architecture overview

Build This section provides the reasoning behind the decisions for every component of the architecture, starting with *Spark*, as the central component. Spark is a platform to process data on computer clusters or single-node machines [1]. It is maintained by the Apache foundation.In this report, we always refer to Spark Structured Streaming when the term Spark is mentioned. Structured Streaming is a new way to handle data streams introduced in Spark 2.x [25]. Among others, the novelty in Structured Streaming is the introduction of micro batching. There are three main reasons why Spark was chosen among other options.
The first one was its popularity and usability. Spark is around since 2014 and is still maintained. It is adopted by a number of big companies, such as Amazon, Yahoo! and Baidu [2]. As a consequence of its popularity, Spark has "[...] the biggest and thus the most active open supply communities out there nowadays" [8]. This increases the usability of the platform, as there is a lot of guidance available online. Moreover, there was existing expertise available in the chair we could benefit from. The second benefit promised by Spark is a relatively easy extensibility. There are multiple libraries available that can enhance Spark out of the box. Examples are GraphX for dealing with graph computations and MLlib [24] for integrating machine learning models into the processing flow. Especially the latter was relevant for us, as a potential future extension of the system is to add a model for cognitive load recognition. Other than that, Spark provides a Python API. This was not a must-have criterion in the selection process but is a very handy add on if a ML-model is integrated in the future, as Python is the most commonly used language for data science tasks. As an alternative, it would also be possible to provide the model via a separate service. However, this entails additional latency to distribute the data and the result.
The last criterion for choosing Spark was its good tradeoff between providing low latencies and high throughput. There are multiple paper such as [10] and [5], which conclude that Spark comes with higher latencies than its main alternatives Storm and Flink (both maintained by Apache) but provides for higher throughput. The reason for this is the Micro-Batching approach taken by Spark. The system splits the continuous stream of input data into smaller groups and processes them once every time interval. This leads to more efficient computa-

tions and therefore higher throughput. However, it increases the latency a bit as the computation is not executed for every incoming datapoint [10].
Considering the requirements of the system, only near real time performance needs to be achieved. Therefore, having delays in the range of a couple of seconds is acceptable. At the same time, the eye-tracker currently in use samples every 8ms, therefore provding a large stream of data which needs to be processed. This becomes even more important as the sampling frequency of the eye tracker increases. To deal with that amount of data, a high throughput of the processing system is required. Therefore, the latency-throughput trade-off in Spark fits our application scenario well.
The second most important decision was to replace the channel for data distribution. The initial system used an HTTP endpoint to send data into the system and a websocket to provide the computation results to the frontend. In our system, we substituted both channels with *Kafka*. Apache Kafka is a distributed streaming platform that is used for building real-time data pipelines and streaming applications. The main reason for this decision was that all relevant streaming platforms (such as Spark, Siddhi, etc.) support Kafka as data source. It was therefore easier to test multiple systems as we did not need to adapt the input modality. Besides that Kafka has some other advantages over HTTP. It allows for decoupling of the producer (in our case the script replaying gaze data) and the consumer (Spark), as the data can be processed asynchronously. HTTP introduces synchronous communication. Among that, Kafka provides for more fault tolerance as it stores the data for a configurable amount of time (retention period). This is not provided by HTTP as it is a stateless protocol. Finally, Kafka is explicitly designed to handle high throughput low latency data streams. HTTP is a request-response protocol that is not built for such scenarios.
With respect to the frontend, the Vue dashboard of the initial solution was replaced by *Grafana*. This decision ties back to the requirement for extensibility. If a new metric is added to the system, a visualization needs to be added programmatically to the custom frontend. This slows down the deployment cycle as it is relatively time intensive. Grafana provides a graphical interface to create and adapt these visualizations with comparably little effort. This decision entails a certain loss of flexibility, as the configuration options in Grafana are not as extensive as for "handmade" graphs. However, since the frontend was not the main focus of our project that was acceptable.
Despite the big number of plug-ins for Grafana, there is no stable solution to integrate the tool with Kafka. There is a plugin [2] that provides this functionality, which did not work to install properly despite many trials. Moreover, little documentation and guidance is available. Therefore, we chose to introduce a *PostgreSQL* database to work around this problem. Grafana provides an out of the box connector which is working stable. A Python script was used to pipe the data from the Kafka stream into the database. We are aware that this workaround introduces some additional latency. However, this was not a significant issue as discussed later on. This solution has the handy side effect that

[2] https://grafana.com/grafana/plugins/hamedkarbasi93-kafka-datasource/

the visualizations in Grafana can be easily adapted by modifying the underlying database query.

Measure The most important measure for the line count metric is the processing latency of the system. To be able to compare our implementation to the initial system, we measured this in exactly the same way. This is the time difference between the last record replayed into the system and the last output of the line counter. In our case, the output time is the time when the result is published to the corresponding Kafka topic. This measure was conducted for different numbers of gazepoints to evaluate the system under rising load. As discussed above, a write operation to the database and a database request from Grafana is necessary to display the results in the frontend. The latency introduced in this part is measured separately for two reasons. Firstly, we wanted to gain insights into the "core" of the system (Spark and Kafka), as the frontend is in principal interchangeable. Moreover, this allows for evaluating how much latency the database workaround introduces and if it is significant. The table below shows the results of the measurements.

Table 1. Latencies of Nifi compared to Spark for different amounts of gaze points

#Records	Latency NiFi (in seconds)	Latency Spark(in seconds)
100	25	4
1.000	240	3
5.000	1.300	3
10.000	2.400	3
30.000	Not measured	4
40.000	Not measured	6

As apparent from the table above, we did not measure the latencies for NiFi with 30 and 40 thousand records. This was because the trend of exponential growth became clear already in the first four measurements. Moreover, the numbers were significantly higher to the Spark latencies, suggesting that no further evaluation is needed for the comparison. Finally, the expected time to perform the measurements is very long (more than 1 hour), which is not feasible considering there are no further insights expected. The most significant insight from the measurements is that the latency of NiFi is growing exponentially with an increasing number of records inputted into the system while it almost stays constant for Spark. This is clearly visible in the figure below. Moreover, the absolute latency was significantly smaller for Spark. To process 10.000 entries, NiFi took 2.400 seconds (about 40 minutes) whereas Spark only took 3 seconds. This is an improvement of 80000%. As described above, the latency of the database workaround was measured separately. For all input sizes it was constantly lower than 0.1 seconds and therefore negligible. With respect to the frontend, a very

small latency between the computation of the line count and the display is added by the refresh interval of Grafana. This is set to 200 ms.

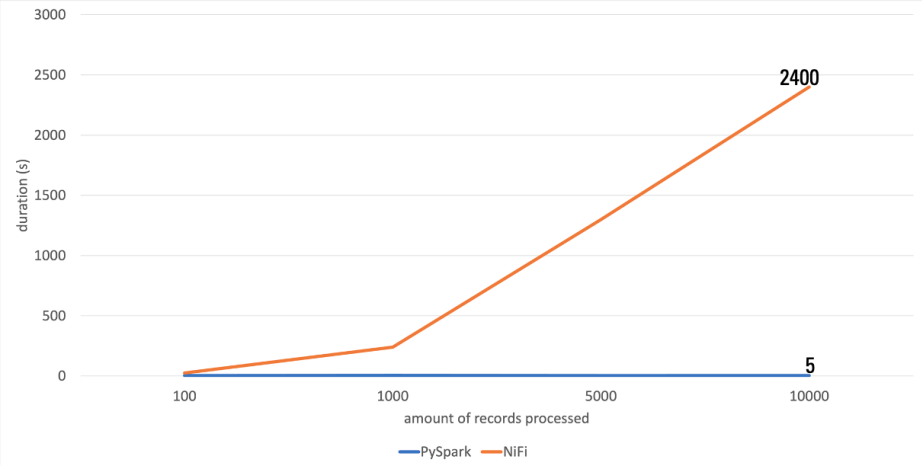

Fig. 3. Latency comparison for the line count metric.

Learn The key insight from this iteration was that it is possible to build a system that satisfies the near real-time requirement with respect to the line count metric. This is in line with reports from similar use cases that deal with counting. One of the most commonly cited ones is the so-called "wordcount benchmark". In this benchmark, the time to count the appearance of every distinct word in a given text is measured. For this task, Spark constantly outperforms other systems [18] [11]. This suggests, that for the line count metric, Spark is a good solution. Moreover, considering the similarity to the wordcount benchmark, Spark might be the best solution. Other than that, it was shown that the initial system using NiFi can be outperformed significantly by introducing a different architecture.

4.3 Iteration 3

Having found from the previous iteration that Spark works very well for the line count metric, we wanted to extend the system by implementing the LHIPA metric in Spark. As the metric is based on pupil dilation, it needs multiple gaze-points from the eyetracker to be computed. This is because the change in the pupil diameter can not be inferred from a single measurement. Consequently, a windowed approach is needed to compute LHIPA. In this section, the different windowing approaches and their limitations are discussed. Afterwards, the limitations we encountered are presented. Finally, the alternative solutions we tried are described.

Build The first approach to implement LHIPA was based on a tumbling window. As discussed earlier, Spark Streaming operates on microbatches. Consequently, only the datapoints within the respective microbatch are available at processing time and therefore for the tumbling window. Since LHIPA needs roughly 45 seconds of replayed data to be computed, this means that every microbatch needs to include at least this amount of datapoints to be able to create a window with enough data. This is visualized in the image below.

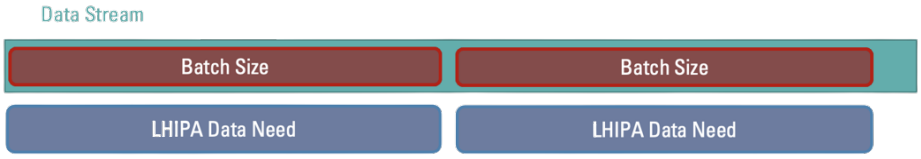

Fig. 4. Visualization of the relationship between batch and window size.

To achieve this, Spark provides options to specify the minimum and maximum amount of records that are read from Kafka during every processing interval. This minimum was set to the minimum amount required to compute LHIPA. While it would have been possible to set the minimum to a higher number to create multiple tumbling windows within every batch, we decided not to do so. This was because we would have introduced more latency by having bigger microbatch sizes, as the computation is only executed once a batch is "completed". Therefore, a microbatch is equivalent to a window in this approach.
As aforementioned above, the LHIPA metric relies on a relatively large amount of datapoints. Implementing the tumbling window approach described above consequently introduces a relatively high latency, as the computation can only be executed once the data is provided in a new microbatch. To mitigate this issue, sliding windows can be used. They have the advantage that only a very small amount of new data is required to compute a new LHIPA score, since data from the previous window is reused. After an initial ramp-up time to fill the first window, this leads to a considerably lower latency. This is visualized in the figure below.

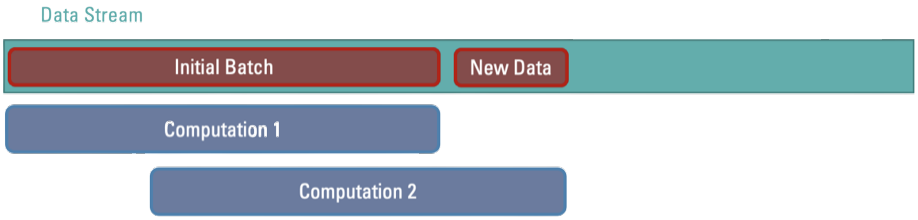

Fig. 5. Visualization of the advantage of a sliding window with respect to latency

Spark natively supports sliding windows. Since it works on microbatches, the case that the relevant data for one window is spread across batches occurrs frequently. This means that some data of this window arrives later, when the next batch becomes available. This is not always an issue. Generally speaking, all computations that can have an interim result can deal with this. Examples are averaging and counting. With these types of operations, the result is updated once the missing data in the window becomes available. However, this becomes a problem once the whole data is needed for a computation. In these cases, the information from the second window is missing at the time of the processing (Processing Time #2 in the image below).

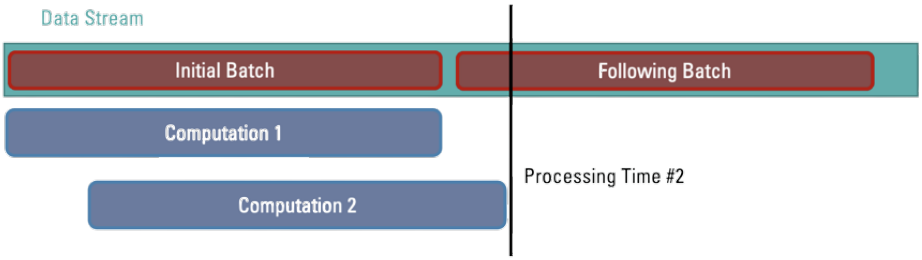

Fig. 6. Visualization of the problem using sliding windows together with batch processing

This tampered the LHIPA implementation with sliding windows, as the whole data is needed. We tried to workaround this issue by concatenating the data from the batches and performing the sliding window on this concatenated dataset, which was not succesfull.

Learn The main outcome of this iteration was a better understanding of the implications of micro-batch processing. While they proved to be advantageous for the line count metric implemented in the previous iteration, they prevented the successfull implementation of the LHIPA metric. This limitation was also recognized by other reports such as [11] and [19]. To overcome this issue, we tried alternatives to Spark not leveraging micro batches. This is subsequently reported on.

Try other solutions While refining the architecture using Spark, different approaches were also considered. These consisted of multiple quick iterations and build phases under the larger umbrella of the third iteration phase. In this phase, we tested the Faust stream processing library for python[3], Siddhi[4], and a self-implemented approach, which attempted to process the incoming data.
Faust is a stream-processing library created at Robinhood. It promised to build data processing agents, which could handle huge amounts of traffic, stating their ability to "process billions of events every day"[5]. The main benefit of this library would have been the ability to use python libraries as usual, allowing for easy extension with a machine learning model or other data-related work, where python is able to show its strengths. However, we were not able to create a valid instance of an agent, that uses a sliding window, bringing us back to the issues we had with Spark in the first place. Even trying a custom implementation of a sliding window using the *HoppingWindow* class failed. This was due to issues with the access to given windows and not being able to determine which window contained which information in a timely manner.
The self-implemented version was very similar to the Faust library in the sense that it followed the same idea: to allow the usage of all common python libraries. However because it is a custom implementation, it was highly specialized and stored only the needed amount of records before discarding the rest. This used simple python data structures such as dictionaries and lists but faced the issue that it was not able to achieve a constant runtime complexity, as seen in Table 2. Based on this linear runtime, we attempted to introduce synchronicity into the implementation, so that after a window had the required size, its LHIPA metric would be calculated asynchronously. This faced common issues of python dealing with asynchronous threads and would not guarantee the order of elements, as well as still maintain a linear runtime, however with a way smaller factor. This meant that this solution was not viable anymore, as the time only beat the very first implementation, but showed clear drawbacks in time when compared to the solution we achieved in Spark.

[3] https://github.com/robinhood/faust
[4] https://siddhi.io
[5] https://faust.readthedocs.io/en/latest/

Table 2. Latencies of the custom implementation for different amounts of gaze points

#Records	Latency custom implementation (in seconds)
100	3
1.000	26
5.000	132
10.000	297

While Siddhi might be a promising tool, configuring it to the specific use case of this project was not the easiest. Again, here we had issues working with connecting the data streams for data coming in and out of the system with Kafka, as well as HTTP. However, going forward and with Siddhi very recently being integrated with the WSO2 Enterprise Integrator[6], this might be a feasible consideration to reconsider going forward.

5 Future Work

Building on the software architecture created, it is possible to extend the system in a variety of ways in the future.

iTrace Online Delivery While during all the experimentation the data was replayed from previously recorded experiments, it should be further tested with the live setup containing an end-to-end test. This includes the raw data coming from the eye tracker, as well as the enriched data providing the line to which the gaze can be mapped.
It further needs confirmation, that the enriched data even can be provided in real-time, which is likely but has not been proven to be true.

More complex measurements Not only could the present architecture be tested with the RIPA metric [14], but also other modalities might be included in the future to enhance the certainty about the perceived stress of the user.
One way to achieve this would be to have different processing entities, which receive the data at the same time through the incoming Kafka Stream. In the present setup, these could then output their individual results into the Grafana visualization, as seen in Fig. 7. Including multiple measurements for a single feedback would however require the timestamps to be aligned, so that for each step in time, there is a definitive answer on how much stress was perceived. Along with this comes the issue of the sampling frequency, which will then need to be considered as well for all involved modalities. This might introduce additional latency into the consideration, as a result, should only be published after all respective modalities have provided data for a given timestamp.

[6] https://ei.docs.wso2.com/en/latest/streaming-integrator/overview/overview/

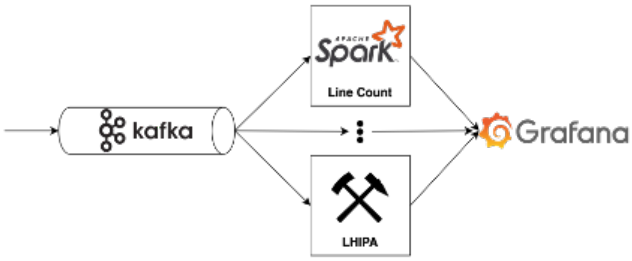

Fig. 7. Exemplified processing of different modalities in parallel.

ML Model While the solution present at the beginning of this project already included a model, which aimed at predicting the stress based on the provided raw data, this model was left out of the new architecture. This decision was made due to the over-simplification of the model and based on the knowledge, that research is being done on better models at the supervising chair. Therefore it might be a well thought consideration to include a machine learning model back into the architecture acting as another modality. This would likely again need to operate on a given span of historical data, as well as follow the restrictions lined out by the previous paragraph regarding time alignment.

Intervention as Result While the intervention was always listed as a big aim of this setup, it has been largely disregarded due to the reworked architecture in the background. However, it still plays a big role in providing the user with the help needed based on the data.
Therefore, additional work can be done to utilize the feedback resulting from the presented architecture. In order to best decouple the components and based on the architecture utilizing Kafka, it might be reasonable to utilize this tool in the Intervention as well in a way that a separate topic receives events on the lines, where a high level of the cognitive load has been detected. The actual intervention is then decoupled and could be adapted based on what shows to be an effective result or based on the personal preference of the individual users.
Examples however could be setting a break-point in relation to the line of the event, this could be before, at, or after the line based on the context and detected complexity. Another idea, which could be researched is to utilize tools such as OpenAI GPT-3[7] or Codex[8] to provide the user with an explanation of the line in the context given.

[7] https://beta.openai.com/docs/models/gpt-3
[8] https://beta.openai.com/docs/models/codex

6 Conclusion

The goal was the creation of a software architecture to be able to process incoming raw data consisting of gazes and code lines in real-time and provide feedback to the user based on a measure of cognitive load. In addition to accurately and efficiently processing incoming data and providing real-time feedback, the software architecture aimed to be extensible, allowing for the easy addition of new metrics and modalities. This was achieved to a certain degree. The basic foundation has been laid out to add further metrics or modalities to the system, as the incoming Kafka Stream can be consumed by multiple entities at the same time and be processed in parallel. However, combining these different elements back together and synchronizing them was not yet achieved and will need to be resolved in a future iteration. We achieved to provide near real-time feedback by using a measure in the count of gazes per line. LHIPA and RIPA might be metrics, which, based on the architecture could be integrated in the future. As could a machine learning model. Therefore, we have proven the basic functionality of providing feedback as close to the event as possible, as well as integrated a simple measure of cognitive load.
Overall, the presented architecture has been thoughtfully constructed in order to achieve best performance and openness of the system. It improves on the previous existing solution and presents an interface, which is not dependent on the state of research regarding eye-tracking data, as further modalities can be added or exchanged when needed.

References

[1] Apache. “Spark”. In: *Spark Website* (2023). [Last visited 13.01.2023]. URL: https://spark.apache.org/.

[2] Apache. “Spark”. In: *Spark Website* (2023). [Last visited 13.01.2023]. URL: https://spark.apache.org/powered-by.html.

[3] Roman Bednarik and Markku Tukiainen. “An Eye-Tracking Methodology for Characterizing Program Comprehension Processes”. In: *Proceedings of the 2006 Symposium on Eye Tracking Research amp; Applications*. ETRA ’06. San Diego, California: Association for Computing Machinery, 2006, pp. 125–132. ISBN: 1595933050. DOI: 10.1145/1117309.1117356. URL: https://doi.org/10.1145/1117309.1117356.

[4] Siyuan Chen and Julien Epps. “Using Task-Induced Pupil Diameter and Blink Rate to Infer Cognitive Load”. In: *Human–Computer Interaction* 29.4 (2014), pp. 390–413. DOI: 10.1080/07370024.2014.892428. eprint: https://doi.org/10.1080/07370024.2014.892428. URL: https://doi.org/10.1080/07370024.2014.892428.

[5] Sanket Chintapalli et al. “Benchmarking streaming computation engines: Storm, flink and spark streaming”. In: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 2016, pp. 1789–1792.

[6] M.E. Crosby and J. Stelovsky. "How do we read algorithms? A case study". In: *Computer* 23.1 (1990), pp. 25–35. DOI: 10.1109/2.48797.

[7] Andrew T Duchowski et al. "The low/high index of pupillary activity". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–12.

[8] Data Flair. "Apache Storm vs Spark Streaming – Feature wise Comparison". In: *Data Flair Blog* (2022). [Last visited 13.01.2023]. URL: https://data-flair.training/blogs/apache-storm-vs-spark-streaming/.

[9] Thomas Fritz et al. "Using Psycho-Physiological Measures to Assess Task Difficulty in Software Development". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 402–413. ISBN: 9781450327565. DOI: 10.1145/2568225.2568266. URL: https://doi.org/10.1145/2568225.2568266.

[10] Darshankumar Vinubhai Gorasiya. "Comparison of open-source data stream processing engines: spark streaming, flink and storm". In: (2019).

[11] Guavus. "5 REASONS WHY SPARK STREAMING'S BATCH PROCESSING OF DATA STREAMS IS NOT STREAM PROCESSING". In: *SQLStream* (2021). [Last visited 13.01.2023]. URL: https://sqlstream.com/5-reasons-why-spark-streamings-batch-processing-of-data-streams-is-not-stream-processing/.

[12] Haytham Hijazi et al. "iReview: an Intelligent Code Review Evaluation Tool using Biofeedback". In: Oct. 2021. DOI: 10.1109/ISSRE52982.2021.00056.

[13] Fuqun Huang et al. "The impact of software process consistency on residual defects". In: *Journal of Software: Evolution and Process* 27.9 (2015), pp. 625–646.

[14] Gavindya Jayawardena et al. "Toward a Real-Time Index of Pupillary Activity as an Indicator of Cognitive Load". In: *Procedia Computer Science* 207 (2022), pp. 1331–1340.

[15] Marcel Adam Just and Patricia A Carpenter. "Using eye fixations to study reading comprehension". In: *New methods in reading comprehension research*. Routledge, 2018, pp. 151–182.

[16] Sebastian Müller and Thomas Fritz. "Using (bio)metrics to predict code quality online". In: May 2016, pp. 452–463. DOI: 10.1145/2884781.2884803.

[17] Takao Nakagawa et al. "Quantifying Programmers' Mental Workload during Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 448–451. ISBN: 9781450327688. DOI: 10.1145/2591062.2591098. URL: https://doi.org/10.1145/2591062.2591098.

[18] Nicolas. "Mini-Cluster Part IV : Word Count Benchmark". In: *Nico's Blog* (2015). [Last visited 13.01.2023]. URL: http://blog.ditullio.fr/2015/10/31/mini-cluster-part-iv-word-count-benchmark/.

[19] John Papiewski. "Advantages Disadvantages of Batch Processing". In: *Techwalla* (2021). [Last visited 13.01.2023]. URL: https://www.techwalla.com/articles/how-does-a-computer-process-data.

[20] Chris Parnin. "Subvocalization - Toward Hearing the Inner Thoughts of Developers". In: *2011 IEEE 19th International Conference on Program Comprehension*. 2011, pp. 197–200. DOI: 10.1109/ICPC.2011.49.

[21] Stevche Radevski, Hideaki Hata, and Kenichi Matsumoto. "Real-Time Monitoring of Neural State in Assessing and Improving Software Developers' Productivity". In: *2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*. 2015, pp. 93–96. DOI: 10.1109/CHASE.2015.28.

[22] Paige Rodeghero et al. "Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 390–401. ISBN: 9781450327565. DOI: 10.1145/2568225.2568247. URL: https://doi.org/10.1145/2568225.2568247.

[23] Thierry Sorg, Amine Abbad-Andaloussi, and Barbara Weber. "Towards a Fine-grained Analysis of Cognitive Load During Program Comprehension". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 748–752. DOI: 10.1109/SANER53432.2022.00092.

[24] Spark. "Spark MLLib". In: *Spark Website* (2023). [Last visited 13.01.2023]. URL: https://spark.apache.org/mllib/.

[25] Prag Tyagi. "Spark Streaming vs Structured Streaming." In: *Medium* (2021). [Last visited 13.01.2023]. URL: https://medium.com/towardsdataanalytics/spark-streaming-vs-structured-streaming-ef6863d5b60.