IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 17 January 2023, accepted 7 February 2023, date of publication 14 February 2023, date of current version 21 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3245043

RESEARCH ARTICLE

Web Performance Evaluation of High Volume Streaming Data Visualization

SAIFUL KHAN, (Member, IEEE), ERIK RYDOW, SHAHRIAR ETEMADITAJBAKHSH, KAREL ADAMEK, AND WES ARMOUR

Oxford e-Research Centre, Department of Engineering Science, University of Oxford, OX1 3PJ Oxford, U.K.

Corresponding author: Saiful Khan (saiful.khan@stfc.ac.uk)

This work was supported by the Science and Technology Facilities Council (STFC), U.K., under Grant ST/W001969/1.

ABSTRACT Many software and hardware applications generate an increasing volume of data and logs in real-time. Visual analytics is essential to support system monitoring and analysis of such data. For example, the world's largest radio telescope, the Square Kilometer Array (SKA), is expected to generate an estimated 160 TB a second of raw data captured from different sources. Transporting large amounts of data from distributed sources to a web browser for visualization is time-consuming due to data transport latencies. In addition, visualizing real-time data in the browser is challenging and limited by the data rates which a web browser can handle. We propose a novel low latency data streaming architecture, which uses a messaging system for real-time data transport to the web browser. Based on this architecture, we propose techniques and provide a tool for analyzing the performance of serialization protocols and the web-visualization rendering pipeline. We empirically evaluate the performance of our architecture using three visualizations use cases relevant to the SKA. Our system proved extremely useful in streaming high-volume data in real-time with low latency and greatly enhanced the web-visualization performance by enabling streaming an optimal number of data points to different visualizations.

INDEX TERMS Visualization system, real-time systems, streaming and messaging system, web services, performance evaluation, and applications.

I. INTRODUCTION

Real-time data visualization is widely utilized to monitor the status and operation of hardware and software systems. Areas which benefit from visualizing large amounts of data in near real-time include, large scientific experiments [1], [2], network monitoring [3], mass spectroscopy [4], [5], [6], [7], fraud detection [8], [9], game analytics [10], [11], radio communications, energy research, atmospheric science [2] and others. As the scale of the monitored system increases the amount of data to be visualized also increases. Large data rates give rise to architectural and visualization bottlenecks which are not present for smaller visualization systems [12].

The Square Kilometer Array (SKA) telescope, will be a large hardware and software system in need of real-time monitoring. The SKA will be the largest radio telescope constructed to date and is expected to capture data at a rate of 160 TB/s [13], and will consist of roughly 100,000 antennas in Australia and South Africa [14]. Owing to its scale and complexity the SKA requires a robust system to monitor its status and operation. To allow operators to monitor the health, and current status of the different parts of the telescope the SKA requires a real-time quality assessment visualization system. Visual monitoring systems are a feature of all current large radio telescopes, such as the Atacama Large Millimeter Array (ALMA) [15], and the MeerKAT [16] radio telescope. More generally real-time monitoring visualizations are used by large and data intensive scientific experiments [17], such as CERN [1], [18], and LIGO [19]. The SKA real-time visual monitoring system serves as an early concrete example of a system which must be able to handle the data rates future large-scale visual monitoring solutions, in a variety of fields, will experience.

Despite the ubiquity of real-time data visualization, little work has been done to address (a) the scalability of such

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

systems and (b) how increased data rates impact the performance when displaying real-time visualizations in a web browser. The architectures used by existing real-time data visualization systems are pull-based, where the browser periodically polls (also called pulls) data from an intermediate source, e.g., a database and updates the display with new data. While this approach is used widely, we observe two key limitations with this approach. Firstly, it does not scale favorably when compared to the SKA's latency requirements for high volume and high frequency data. Secondly, the performance of web browsers limits how much data can be processed, displayed, and updated within a given time frame. The browser may crash or the visualization lag if the rate at which is supplied to a web-based visualization exceeds what the browser is capable of.

To address these limitations, we start by conducting an empirical study to determine the latencies due to the different stages of an architecture which relies on polling. Based on this investigation, we propose an alternative, push-based, data streaming architecture. Motivated by the success of message brokers [20] for message pushing applications, we find that they are able to support our latency requirements [21] and stream the various types of data required for different visualizations in parallel. We thus adopt a message broker, designed for streaming data, to implement the push-based architecture for real-time data visualization applications

Since the proposed push-based streaming architecture was able to supply data for real-time visualizations at rates greater than a browser may be able to cope with, the introduction of the push-based architecture necessitated measuring the browser performance when displaying various visualizations. These performance measurements allow us to optimize the streaming data rates for a browser, to minimize rendering lag when displaying real-time visualizations. The browser visualization performance measurement technique, necessitated by the increased throughput of the push-based architecture, is another novel contribution of this work. Three real world use cases are implemented to evaluate their web visualization performance as well as to evaluate the streaming data architecture. The outcome of our work is as follows:

- We reviewed technologies used for radio astronomy data visualization and real-time data visualization in general. We discussed their limitations in high-frequency and high-volume real-time data visualization contexts, such as for the SKA.
- We proposed a scalable architecture for high-frequency and high-volume data streaming and visualization in the web browser. This architecture is transferable to different data intensive domains.
- We identified key performance indicators that contribute to latency in real-time web data visualization. We implemented three visualizations as use cases and analyzed their performances in terms of the identified key performance indicators.
- We developed a tool for supporting rapid development and evaluation of web visualizations and open-sourced the tool. Finally, we discuss the limitations of our study and propose recommendations for high-volume real-time data visualization in the web browser.

In Sec. II of this paper we discuss the current standard practices of real-time and dynamic data visualization system development. In Sec. III, we discuss the limitations of a traditional, pull-based, architecture in terms of the SKA's requirements. To address these limitations, we propose a scalable data streaming architecture in Sec. IV. In Sec. V, we present the methodologies for analyzing the performance of key performance indicators (V-A), three visualization use cases and their data models (V-B), and brief implementation details (V-C). Followed by Sec. VI, where we present serialization, deserialization, data streaming or transmission, and rendering performance of the three use cases. In Sec. VII, we discuss the general application of our work, limitations of the study and recommendations for overcoming these limitations.

II. RELATED WORK

A. RADIO ASTRONOMY AND SCIENCE DATA VISUALIZATION

In radio astronomy projects, such as the SKA, data visualization is commonly used to monitor instruments and visualize the data captured by the telescope. In the case of the Atacama Large Millimeter Array (ALMA) radio telescope [15], [22] data is saved in a database. The database facilitates on-demand retrieval as well as near real-time polling of the data for visualizations. ALMA's dashboards are implemented using a combination of technologies, including d3.js for the interactive data driven visualizations and SVG for rendering. MeerKAT [23], another radio telescope, uses a bespoke system for monitoring the antennas. MeerKAT's system uses a Redis Pub/Sub messaging system to transport data to the user interfaces. MeerKAT also uses Graphana, an off-the-shelf solution, for creating dashboards. A large science project outside radio astronomy, CERN [1], [24] uses the messaging system, e.g., Kafka, for transporting data from different sources to central databases, e.g., search engine (Elasticsearch), time series database (InfluxDB), Apache Hadoop and so on. They developed user interfaces and visualizations using off-the-shelf tools, e.g., Grapha, InfluxDB, Prometheus, Kibana, etc., for monitoring different systems [18]. The LIGO Scientific Collaboration developed web-based real-time science data visualization [19] but did not provide sufficient technical details of their visualization software and its architecture.

All the aforementioned works are motivated by the need for large-scale data visualization for real-time monitoring. However, these works lack technical contributions and offer limited insight into more general streaming data visualization system design and development. Our work addresses this gap with a more technical contribution to high-volume and low-latency streaming data visualization system architecture and development.

B. REAL-TIME DATA VISUALIZATION

Real-time and dynamic data visualization systems are of course also developed for a broader range of applications, including uses both within the wider sciences and industry. Examples include: network monitoring [3], analysis of vibrations during drilling operations [25], monitoring of sensors, manufacturing processes, control and hardware [26], urban air quality monitoring [27], etc. All these systems use databases to store the data received from the source and use the database polling technique for near real-time access and data visualization.

Visualization of real-time three-dimensional (3D) data, e.g., road environment data from light detection and ranging (LiDAR) is presented in [28], and 3D GIS data is presented in [29]. We anticipate that the amount of data to be visualized in this project is large. However, there are limited technical details on how the data is streamed or transferred to the user interface in real-time.

Protopsaltis et al. [30] presented a survey of visualization methods, tools, and techniques for the IoT. They discuss different visualization tools, techniques, and challenges in developing visualizations for various IoT domains. However, there are no technical details on streaming data visualization systems.

Mass spectroscopy is another field where large amounts of data is commonly visualized using the browser. Examples include the seaMass [7] software. This work involves streaming and then visualizing large amounts of scientific data, by sending a data set iteratively, based on the cognitive importance of the different features, thus quickly presenting the most important information to a user. However, this application differs from a monitoring application in that a selected data set is streamed to the browser in steps, as opposed to using streaming to visualize data which changes in real-time. In this work polling is used to stream the data to the client.

In the field of game analytics, visualization is used to investigate the player behavior in online games [10], [11], for example in the CMX educational MMORPG [31]. This can involve visualizing real-time player data, as is done by the yoGURT framework [32], a framework which relies on polling to stream data.

The popular off-the-shelf visualization tools, e.g., Graphana, Tableau, and PowerBI do not implement real-time data streaming functionality. Recent work on a large-scale and dynamic data visualization system is presented in [33]. However, this work does not require real-time functionality.

Messaging systems (e.g., Kafka, RabitMQ, Redis Pub/Sub, etc.) have been widely used for real-time data streaming and transport in a variety of application domains, e.g., IoT, messaging apps, radio astronomy data, and science data. Especially Kafka delivers high volume data streaming with low latency [21]. However, the use of messaging systems for real-time streaming data visualization systems is rare in the literature.

C. WEB VISUALIZATION PERFORMANCE EVALUATION

Hoetzlein [34] analyzed graphics performance in rich internet applications using transparent 2D sprites. Their results show that the application performs better in the Google Chrome browser than in Internet Explorer and Firefox. They also implemented the application using WebGL-based GPU accelerated visualization, which outperformed the Flash and HTML5 Canvas [35] implementations. Kee et al. [36] suggested that interactive visualizations that require high performance should be implemented in WebGL and Canvas. The performance gain of visualizations implemented in WebGL is marginally higher than HTML5 Canvas implementation, whereas the development cost of a WebGL application is significantly greater.

Babovic et al. [37] performed web performance analysis for IoT applications and reported improved performance of HTML5 Canvas. Schwab et al. [38] proposed SSVG to translate SVG code to Canvas automatically. The development cost of SVG is the lowest compared to other methods, and it is popular and widely used for 2D data visualization. However, in SVG, rendering and animation are slow when the scale of data to be visualized is large. For that reason, HTML5 Canvas is preferred for large-scale data visualization and animation. Therefore, we use Canvas to implement our visualizations.

Lee et al. [39] surveyed and tested web-based data visualization tools and libraries, e.g., Google Charts, Flex, OFC, d3.js and, JfreeChart. They used 100,000 static data points for performance analysis.

None of these works address high volume, real-time and dynamic data visualization and its performance. In our work, we analyze the web visualization performance of large-scale and real-time streaming data.

III. REAL-TIME DATA AT SKA

This section will provide a broad overview of the SKA, which motivated this work and informed the real world visualization use cases. The SKA will consist of a large number of antennas which capture radio astronomical data, such captured data is called visibilities or visibility data. The first step of the data processing is that on-site "correlators" correlate the captured data. The correlated data is then sent to a distributed *Science Data Processor* (SDP) via a high-speed network. As shown in Fig. 1, the SDPs temporarily save the data in a very large *Shared Memory*. Different *Applications* access the data in real-time for, e.g., reconstruction of the sky image, astronomical discoveries, studying the astronomical signals, detecting anomalies, and so on. *Metric Generators* are distributed streaming data processing [40] applications that generate different metrics in real-time from the streamed visibility data, e.g., power spectrum, the phase information of baselines, polarization, etc. The metric generators also generate metrics from data received from other applications, e.g., real-time radio frequency interference detectors, different sensors on the antennas, other instruments, and so on.

The data processing pipeline of the SKA is designed to implement expensive operations, e.g., extraction, cleaning,

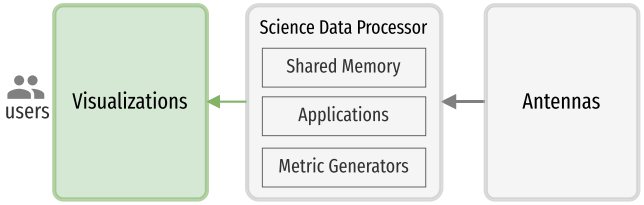

FIGURE 1. The high-level data flow at the SKA. The radio astronomical visibility data captured by the antennas are saved in the very large and distributed *Shared Memory* of *Science Data Processor*. Distributed instances of *Applications*, e.g., *Metric Generators* compute different metrics from the visibility data. Finally, the metrics are visualized in a web browser in real-time for the users, e.g., telescope operators and radio astronomers.

transformation, aggregation, etc., using low-level functions, e.g., metric generators, deployed in high-performance computing infrastructure. The metric generators also do additional pre-processing of data required to reduce consumers' or web browsers' need for further processing. While we will discuss the data structure of three metrics used for visualization use cases in this paper, the details about how the metric generators derive these metrics from the raw visibility data and the performance of the metric generators are not within the scope of this work. The essential functional role of our system is to transfer data generated by the metric generators to the browser and visualize different metrics using different plots and dashboards as in Fig. 1. This is to convey the current state of the telescope to operators and radio astronomers.

A. REAL-TIME DATA VISUALIZATION TECHNOLOGY OVERVIEW & CHALLENGES

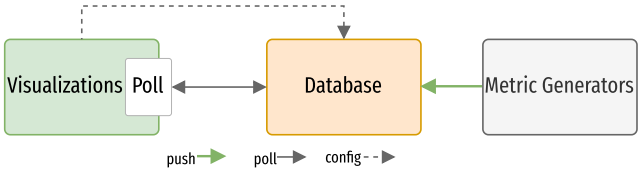

FIGURE 2. Off-the-shelf solutions use a database to store real-time data (e.g., timeseries), and visualization functions periodically polls the database and updates the plots and dashboards with new data in near real-time.

There are numerous off-the-shelf data visualization systems available for visualizing data in web browsers. In our first prototype, we adopted the polling data principle common in the literature. We used InfluxDB, a timeseries database to store the metrics generated by our metric generators and Grafana dashboard to continuously poll the data from the database and visualize. The high-level architecture of this prototype can be found in Fig. 2. The metric generators write new metrics to the timeseries database tables every second, hence the Grafana connector is configured to poll (illustrated in Fig. 3(a)) and retrieve the metrics from the database tables at every second. We used Grafana's built-in visualizations

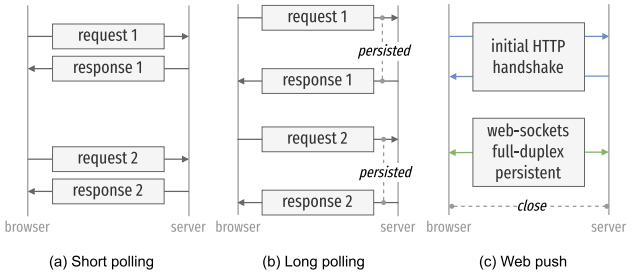

FIGURE 3. In (a) short-polling, a browser requests the server over a predefined interval for any new data. In (b) long-polling, the server receives a request but does not respond until new data is available from another request. In (c) Web push, e.g., WebSockets, a browser opens a two-way connection with the server, and the server keeps track of each client and pushes messages to the clients whenever available.

TABLE 1. Our test results show that using asynchronous driver and in-memory databases, e.g., Redis, we can reduce database I/O latency from millisecond (ms) to microsecond (μs) range.

Database	Driver	I/O operation time
InfluxDB	Synchronous	100–150 ms
InfluxDB	Asynchronous	3–5 ms
Redis Timeseries	Asynchronous	0.8–0.9 μs

Each payload consists of a 3-dimensional data in format [channel, baseline, polarization] × timesteps. In our test application, we used payload size [50000, 10, 4] × 1 to test read and write operation latency. Python's floating-point numbers are 64-bit. Therefore, a payload of dimension [50000, 10, 4] is ~15.25 MB. The results will vary from machine to machine.

(e.g., line chart, bar chart, and waterfall plot) to visualize the polled metrics.

While testing and evaluating the prototype we observed two major problems (a) a significant latency in transporting the data to web browser, and (b) the browser rendering performance.

For example, for a data of dimension [50000, 10, 4] × 67 ([channels, baselines, polarizations] × timesteps) (∼1022.34 MB) it takes more than ∼27 minutes to transport and render in Graphana chart, i.e., average ∼24 seconds for each data points. However, for a small data set the latency was low, e.g., data of dimension [1], [4], [10] × 133 (∼41.56 KB) takes only ∼24 seconds, i.e., on average ∼180 ms for each data point. The source code of our prototype can be found in [41] and additional test results can be found in the supplementary material provided with this paper. Please note that this test measures total latency: writing data to the database, polling the data from the database, and rendering the data into plots, e.g., line charts. When using an off-the-shelf tool, e.g., Grafana, one cannot easily measure the latency due to the individual steps of the visualization pipeline (e.g., transmission, deserialization, and rendering); doing so would require extensive customization of the tool.

In order to identify which steps of the process contribute the most to the total latency we performed end-to-end profiling of the prototype.

(1) Network latency – The prototype is designed as a service-oriented architecture [42] where the metric generators, databases, and web applications are developed and

composed [43], [44] as separate services running in different containers. It requires some time to transfer the data from one service to another, e.g., the metric generators to the database and the database to the web browser. Although network latency cannot be completely eliminated, it is known to depend on: payload serialization technique, size, proximity, the network connection speed between the services, etc. In Sec. VI, we will investigate serialization techniques which can potentially improve payload size, i.e., reducing network latency, without introducing significant deserialization cost at the browser.

(2) Database read and write operation latency – We noticed a significant latency in database operations. The database takes a significant amount of time to write the incoming metrics to an index and perform additional housekeeping operations, e.g., transaction, consistency or distributed transaction, linking records using primary and foreign keys, persistence, disk and in-memory data operations. We found that, on average, a synchronous InfluxDB read and/or write operation takes 100–150 milliseconds. Different databases perform different housekeeping operations based on their characteristics. For example, a non-transactional database can save a significant amount of transactional (e.g., atomicity, consistency, isolation, and durability (ACID) properties of database) processing time and an in-memory database can achieve very low latency and high throughput as data stays in primary memory. The selection of a database depends on the requirements and use case. To achieve the best performance, it is also essential to find the optimal ingestion workload parameters, e.g., the number of concurrent clients, batch size, the structure of ingestion data, the interval between two neighboring data points, and the data model to support queries and so on.

We refactored the prototype to find the latencies caused by database operations and reduce these further. We noticed that an asynchronous InfluxDB operation takes 3–5 milliseconds, which is a ∼30 times performance improvement compared to a synchronous operation. We further reduced the read and write operation latency to less than a microsecond by replacing the InfluxDB with an in-memory database, RedisTimeseries. The results are presented in Table 1.

(3) Polling latency – Polling is a technique in which the browser regularly asks the database for new data, as in Fig. 3(a). The web visualization functions make repeated requests to the database server for new metrics at a predefined interval (in our case, it is ∼250 milliseconds), as in Fig. 2. These repeated requests waste resources. For example, each new incoming connection must be established, the HTTP headers must be passed, a query for new data must be performed, and a response (usually with no new data to offer) must be generated and delivered. Finally, each connection must be closed, and any resources cleaned up. As discussed earlier, where we can improve the database I/O operation latency by the choice of database and varying the settings, there is no quick solution or workaround to improve the polling latency.

(4) Web visualization latency – In the web browser, ideally, each received data point should be processed and rendered into a visualization before the next data point arrives. If the metric generator generates and sends a metric to the browser at every T_{freq} second interval, the payload must be visualized in less than T_{freq} seconds. For the SKA, the value of T_{freq} is 1 second.

(i) Data processing latency – Web browsers receive data as serialized payloads. The serialized payloads need to be deserialized, transformed and mapped to a data structure compatible with visualization functions which renders the data into plots. We can minimize the transformation and mapping cost by prepossessing the data at the metric generator. However, the deserialization cost can only be reduced using an appropriate serialization and deserialization protocol. The traditional and commonly used protocol, JSON messages, are exchanged in text format. In contrast, the more advanced protocol ProtoBuf provides a set of rules, APIs, and data types to serialize and exchange messages more efficiently and cost-effectively. We investigate performance, e.g., serialization and deserialization cost of both the protocols in Sec. VI.

(ii) Rendering time – Rendering time depends on multiple factors, such as the amount of data to be processed, the processing functions, the visualization designs (plots) and their complexity, as well as the low-level rendering technologies used (e.g., SVG, Canvas). As discussed in the literature in Sec. II, Canvas graphics generally provides better performance than SVG. We therefore implemented a visualization use case, the spectrograms, using Canvas. However, different visualizations have different complexity and each needs to be evaluated separately. In Sec. VI, we investigate rendering performance for three different visualization use cases as examples.

IV. A SCALABLE ARCHITECTURE FOR STREAMING DATA VISUALIZATION

To address the inherent polling latency of the pull-based architecture, we adopted a messaging system and proposed an improved push-based architecture, seen in Fig. 4. The main objective of our proposed architecture is to stream data to a web browser using push-based communication protocols eliminating data transmission latency incurred due to periodic polling. There are four main components of the proposed architecture:

(1) Messaging System – The main component of our data streaming system is a messaging system. The messaging system sends messages between applications, processes, and servers, as in Fig. 5(a). The *broker* or message broker, e.g., Apache Kafka [47] is a publish-subscribe based messaging system. The broker handles all requests (e.g., produce, consume) and metadata from clients and keeps data replicated within the cluster, as in Fig. 5(b). There can be one or more brokers in a cluster.

Brokers use *topics*, which are categories or feed names to which records are stored and published. Our metric generators publish metrics on different topics. Applications,

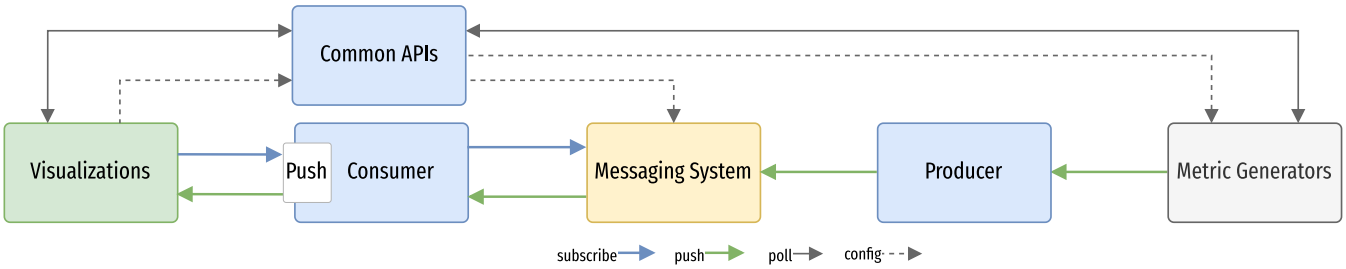

FIGURE 4. A generic architecture for high-volume data streaming and visualization. In general, the source and the data processor (here, Metric Generators) components will vary from application to application. Following this architecture, we developed a tool which is available in GitHub [45].

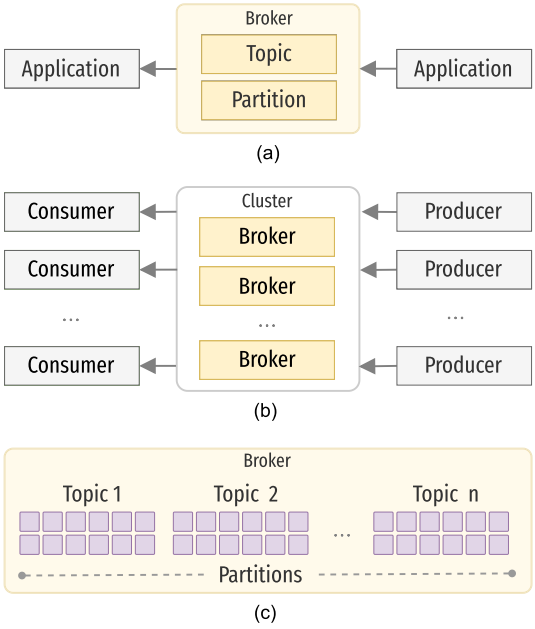

FIGURE 5. A detailed description of the *Messaging System* illustrated in Fig. 4. (a) The broker is implemented using the Apache Kafka messaging system which sends messages between applications. (b) The messaging system may consist of a cluster of brokers. (c) Data are organized into Kafka topics. This figure is adapted from [46] and [47].

e.g., web visualizations subscribe to the messaging system and consume the metrics published into the relevant topics. The metrics are pushed to the browser via WebSocket. Any application can connect to the system and process or reprocess metrics from a topic, as shown in Fig. 5(a) and (c).

Topics are divided into *partitions*, which contain records or metrics in an unchangeable sequence. Each record in a partition is assigned and identified by its unique offset. A topic can also have multiple partition logs. This allows multiple consumers (for instance, multiple visualizations opened in a browser window, or different user's machines) to read from a topic in parallel, as in Fig. 5(c). In addition, a database (e.g., Zookeeper) keeps the state of the cluster, e.g., brokers, topics, users, etc. The database also manages the brokers in the cluster.

Kafka achieves low latency message transport through sequential I/O and Zero Copy principles. A detailed description of Apache Kafka message broker can be found in [47] and a comprehensive benchmark of it in [21].

(2) Producer – Our metric generators publish different metrics to the message broker using the producer APIs. The producers are processes that push records into Kafka topics within the broker. The producers publish or write a stream of events to one or more Kafka topics. The metrics or data are stored for a specified retention period, which can be configured using the producer APIs.

(3) Consumer – While producer applications write data to topics the consumer applications read from topics. The *Consumer* subscribes to one or more topics and reads records on these topics. A consumer reads messages from partitions, in an ordered fashion. For example, if messages $m1, m2, m3, m4$ are inserted into a topic in this order, the consumer will read them in the same order. Since every message has an offset, every time a consumer reads a message it stores the offset value in Kafka, denoting that it is the last message that the consumer read. Additionally, if at any point in time a consumer needs to go back in time and read older messages, it can do so by resetting the offset position.

Web applications establish connections with consumers using WebSocket. The data read by the consumers are immediately transported to the web browser.

(4) Common APIs – These are a set of REST APIs implemented to perform various operations, e.g., configure different parameters of the message broker, sending commands to the metric generators (e.g., setting different averaging factors, selecting serialization protocols, configuring the streaming data processing engine, etc). We also implemented a set of APIs to pull data or metrics from the metric generators on-demand. The Kafka Confluent Control Center provides off-the-shelf APIs and a user interface for managing Kafka clusters.

V. EMPIRICAL STUDY DESIGN

A. METHODOLOGY

Following the development of the architecture described in the previous section, we used two high-level approaches to evaluate the key performance indicators of our proposed streaming data visualization system. In the first evaluation, we generate serialized payloads using different protocols on the server side and deserialize the payloads in the

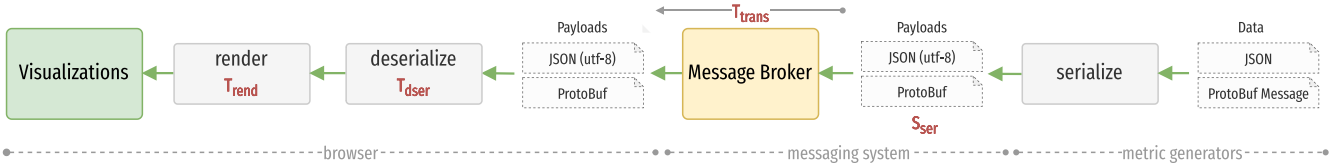

FIGURE 6. An end-to-end sequence diagram of dataflow with key performance indicators, e.g., payload size (S_{ser}), transmission time (T_{trans}), deserialization time (T_{dser}), and rendering time (T_{rend}) for measuring performance and latencies.

browser. This evaluation aims to measure the serialization and deserialization cost of the JSON and ProtoBuf protocols. That is, which serialization protocol generates smaller payloads and which is faster to deserialize.

For the second evaluation, we visualize the deserialized payloads in the web browser using appropriate plots implemented using JavaScript functions. This test aims to measure visualization latency, e.g., to determine how much time it takes to process and render payloads of different sizes into relevant plots.

We considered three visualizations as use cases, these were proposed by the domain scientists working on the SKA and radio astronomers. We deployed the system components (in Fig. 4) in separate containers on a host machine, emulating a distributed system. We used around 16 TB of visibility data captured by the MeerKat telescope to generate the metrics required by the three visualization use cases. The data is stored in a large network storage. Every second (T_{freq}), the metric generators compute metrics from the part of the visibility data captured on a particular timestamp and stream them through the message broker to the web browser.

Our application has several components and we measure latencies of key performance indicators, as shown in sequence diagram Fig. 6:

(1) Serialized payload size (S_{ser}) – Metric generators serialize data to create payloads. There are two key performance indicators, e.g., serialization time and payload size involved at this stage. Analysis of serialization time is not within the scope of this paper, as serialization will happen in a high-performance computing environment as a part of the data pre-processing. However, we want the size of the serialized payload to be small, so it can be reliably handled by the message broker and quickly transferred through the network. The payload size of serialized data depends on the data format and serialization protocol used.

(2) Transmission time (T_{trans}) – This measures the time spent sending metrics from metric generators to the browser. The components are deployed as distributed services, and the value of this parameter will depend on the proximity of the services, the network connection between the services, etc. In our test application, we deploy the services in multiple containers connected via the bridge network of a host machine.

(3) Deserialization time (T_{dser}) – The main latencies in the web browser are due to deserialization of payloads, data processing, and rendering cost. When the payloads are

```
message Spectrum {
  int64 timestamp = 1;
  float x_min = 2;
  float x_max = 3;
  float y_min =  4;
  float y_max = 5;
  repeated float power = 6 [packed=true];
  repeated float channels = 7 [packed=true];
  optional repeated float ci_l = 8 [packed=true];
  optional repeated float ci_h = 8 [packed=true];
}
```

```
class Spectrum (BaseModel):
  timestamp: int
  x_min: float
  x_max: float
  y_min: float
  y_max: float
  power: List [float]
  channels: List [float]
  ci_l: List [float]
  ci_l: List [float]
```

FIGURE 7. The spectrum plot data model in Protobuf (left) and Python (right).

received in the web browser, they are deserialized using Javascript functions. The deserialized payload is a JavaScript object. The deserialization time can contribute significantly to the latency.

(4) Rendering time (T_{rend}) – The deserialized JavaScript objects are visualized using corresponding visualization functions. Each visualization function executes a drawing function, which includes instructions on how the data will be plotted and rendered to the browser display. Internally, the browser executes Critical Rendering Path (CRP) [48] which involves JavaScript execution, rendering, and rasterization to create and display the visualization. With this metric we report the CRP time.

The metric generators generate and send metrics to the browser every second (T_{freq}), and each payload needs to be processed and rendered before the next payload arrives. We need to ensure that total latency in the browser, $T_{rend} + T_{dser} \leq T_{freq}$. As if this combined latency is greater than T_{freq}, one second in our case, the payload will accumulate, the browser heap size will increase, and the browser will become unresponsive.

B. USE CASES

To study the streaming architecture performance and the web visualization performance on the streamed data, we implement a simple use case, e.g., line plot, and two data-intensive use cases, e.g., spectrograms. These use cases were selected through discussions with SKA's domain scientists and radio astronomers.

1) POWER SPECTRUM PLOT

Multi-antenna radio telescopes rely on radio-interferometry. The mathematical details of interferometry are beyond the scope of this paper; however, the general idea is to combine the interference patterns generated by pairs of

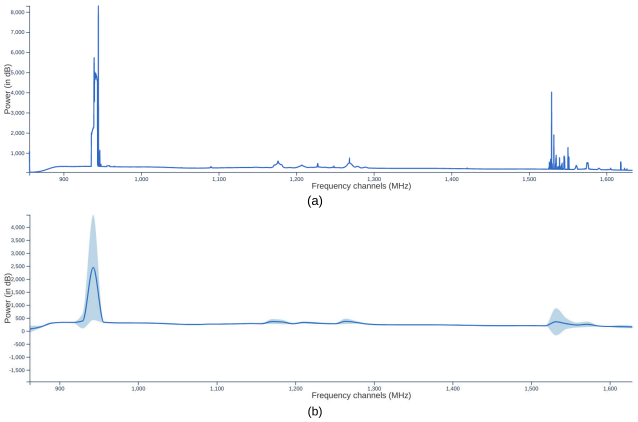

FIGURE 8. Example power spectrum plots computed from data from the Meerkat telescope. In this example, (a) plot all 4096 channels and their power values, and (b) averaged the number of channels to 64 ($\frac{4096}{64}$), where the confidence intervals are shown with a translucent color band. Channels are averaged together to reduce the number of data points.

```
message Spectrogram {
  int64 timestamp = 1;
  string baseline = 2;
  string polarisation = 3;
  repeated int32 phase = 4 [packed=true];
}
```

```
class Spectrogram (BaseModel):
  timestamp: int
  baseline: str
  polarisation: str
  phase: List [int]
```

FIGURE 9. Spectrogram data model in Protobuf (left) and Python (right).

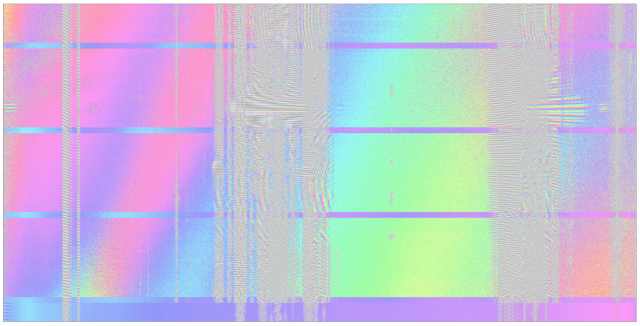

FIGURE 10. An example waterfall plot of a spectrogram representing a baseline and a polarization. The horizontal axis plots each frequency channel, and the color shows each channel's phase value (0° – 360°). The vertical axis is the time axis. The phase values are mapped to an HSL colormap.

antennas (called *baselines*), based on their physical location, to obtain an image of the observation field after a series of mathematical operations. More specifically, the complex voltage signal received by each antenna in each frequency channel is provided to a hardware component named a *correlator*. The correlator multiplies and averages over the voltages measured by all pairs of antennas. The value correlators produce are complex numbers called visibilities, these include cross-correlations (between two distinct antennas) as well as autocorrelations (between an antenna and itself). These values then go through various layers of processing to produce an image of the sky as it appears in the radio part of the electromagnetic spectrum. Autocorrelation values are proportional to the amount of power received by each antenna.

Fig. 7 shows the *Spectrum* data structure or data model used to encode the power spectrum data. The main attributes are the *channel* which is an array representing the channel values in MHz, and the *power* array containing the corresponding power values. Each channel and its corresponding power value are mapped to the x-axis and y-axis of a line plot. *x_min* and *x_max* are minimum and maximum channel values, which determine the range of the x-axis. Similarly, *y_min* and *y_max* are minimum and maximum power values, which determine the range of the y-axis.

Fig. 8 shows the distribution of the average autocorrelation values of all antennas across the frequency channels, i.e. the power spectral density (power spectrum) plot. Note that only the amplitude of the visibilities are used in this plot. Such a plot not only contains useful astronomical information but also helps with monitoring the functioning of the antennas. The SKA project, as the largest radio-telescope ever built, will operate on a maximum of 65,000 frequency channels.

The *ci_l* and *ci_h* are optional fields corresponding to confidence intervals, these fields are computed when we average multiple channels together, as in Fig. 8(b). The SKA can capture around 65,000 frequency channels. Averaging the number of channels together reduces the overall size of the payload to be transferred.

2) WATERFALL PLOT OF SPECTROGRAM

As discussed before, the correlator is constantly producing cross and autocorrelation values for all the baselines. Therefore, it is of interest to the operator to see these values for different frequency channels over time for each baseline as a moving signal.

Fig. 9 shows the *Spectrogram* data model. The key attributes are the *phase* which is an array representing the phase values between 0° – 360°, and the corresponding *baseline* and *polarisation*. Each phase value is mapped to the x-axis.

Fig. 10 shows an example snapshot of such a spectrogram [49] visualizing the phase values of the visibilities as a function of time and frequency (a similar plot can also be generated for the amplitudes). This plot is for a given baseline and polarization (the received signal by each antenna and the visibilities contain four different polarizations).

A spectrogram has x and y dimensions of frequency channels or phase and time. Data flows down, in a waterfall plot manner, as in Fig. 10. This effect is achieved by shifting the block of previously stacked spectra down the screen by one pixel or one line, then a new line of data is added at the top, with the oldest line of spectral data disappearing off the bottom of the display. Fig. 10 shows the 4096 phase data points plotted as a function of time. The current time and newest data are always at the top of the plot. The data rate is set at 1 line/sec, and the waterfall plot is set to hold 600 lines (height of the display). Therefore, the oldest line, written 600 sec ago, is at the bottom.

```
message Spectrograms {
  int64 timestamp = 1;
  repeated Spectrogram spectrogram = 2;
}
```

```
class Spectrograms(BaseModel):
  timestamp: int;
  spectrograms: List [Spectrogram]
```

FIGURE 11. Spectrograms data model in Protobuf (left) and Python (right).

Building the waterfall display is a two-step process. The plot is built off-screen by directly writing each pixel color into the memory buffer of an off-screen canvas. Each data point is represented as a rectangle of the height of one pixel, and the width equals to width of the display divided by the total number of data points. In Fig. 10, the width of each data point is ~ 0.5 pixel ($\frac{2000}{4096}$). Successive lines are written as adjacent rows of pixels. This method fixes the waterfall dimension to the length of the data vectors to be plotted and the number of data lines in the display. The second step is to use this off-screen canvas as a source for drawing an image of the Waterfall in the on-screen canvas using the *canvas.putImageData* method, thus allowing on-the-fly scaling of the data for presentation in any required size.

The phase values are between $0° - 360°$, and the color of each phase is computed by mapping to the hue, saturation, lightness (HSL) color map. Spectrograms are implemented in Canvas.

3) SPECTROGRAMS OF BASELINES AND POLARIZATIONS

Both from the astronomical and observation management perspectives (e.g. monitoring the operation of the telescope), it is important to display the spectrogram plots for a collection of baselines. However, it should be noted that for an array of antennas such as the SKA, the number of baselines (all possible pairs of antennas) will be gigantic (e.g., 65,000 channels and 130,000 baselines). In fact, even for smaller radio telescopes the number of baselines (which grows quadratically with the number of antennas) might be too large to be meaningfully visualized. Therefore, the display should allow the user to select subgroups of baselines depending on the requirements of the operator. For instance, the operator might be only interested in autocorrelations to monitor the status of individual antennas, or from the imaging perspective a certain group of baselines (e.g. longer baselines or shorter ones) might be of interest.

Fig. 11 shows the *Spectrograms* data model, which is a list containing multiple *Spectrogram* data. Fig. 12 shows an example snapshot of a series of spectrograms for different baselines and polarizations in a single display.

C. IMPLEMENTATION AND EXPERIMENTAL SETUP

We used the JavaScript library React [50] to implement a web user interface and HTML SVG and Canvas graphics to implement the visualizations. As described earlier, we used push-based protocol, WebSocket, as an underlying protocol for communication between the web browser and consumer. The server side components, e.g., consumer, producer and metric generators are implemented in Python. The production

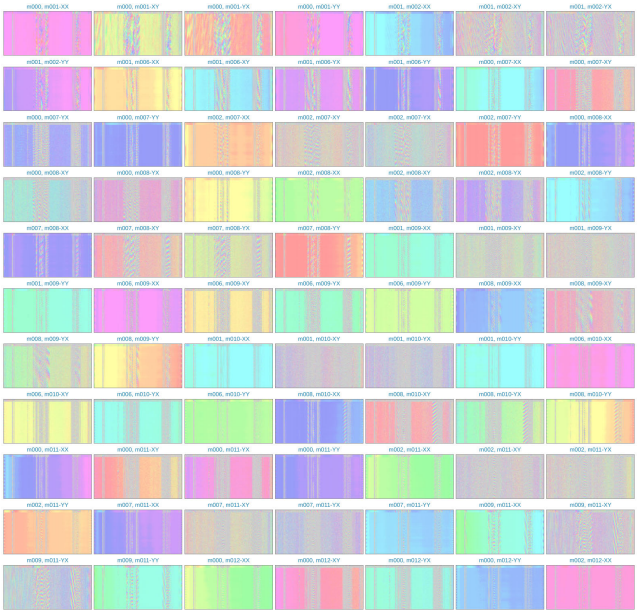

FIGURE 12. A table of 77 small spectrograms, each spectrogram representing a baseline and a polarization. Fig. 10 shows the high-resolution view of one of the spectrograms.

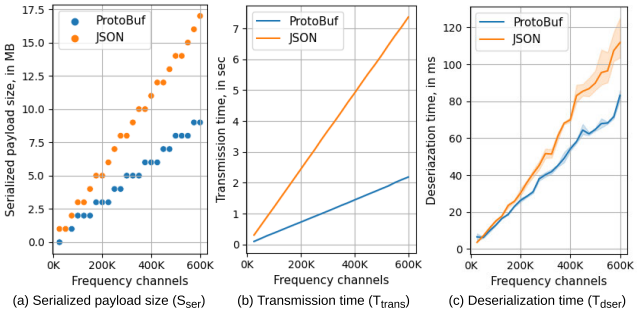

FIGURE 13. Serialization, transmission, and deserialization performance of Protobuf and JSON encoded *Spectrum* data. (a) Presents the serialized payload size (in megabyte), (b) presents the transmission time (in second) of the payloads, and (c) presents the deserialization time in the browser (in millisecond). The translucent color band shows confidence intervals in 95%.

version of the metric generators, data APIs, and the user interface are under development and are open sourced via GitLab [51], [52], [53]. The tool developed for rapidly prototyping and testing different data models and visualizations is also open source and available via GitHub [45].

We considered two widely used serialization protocols, JSON and ProtoBuf. In order to compare their efficiency, we generate different serialized payloads using both JSON and ProtoBuf protocols and we record the payload sizes. We then deserialize the payloads in web browser using JavaScript functions and we record the deserialization times. In order to compare the rendering performance of a visualization function, we feed data to the visualization function every second. For each received data point the drawing function is executed and the changes are animated. We used the Chrome browser runtime performance analyzer to collect

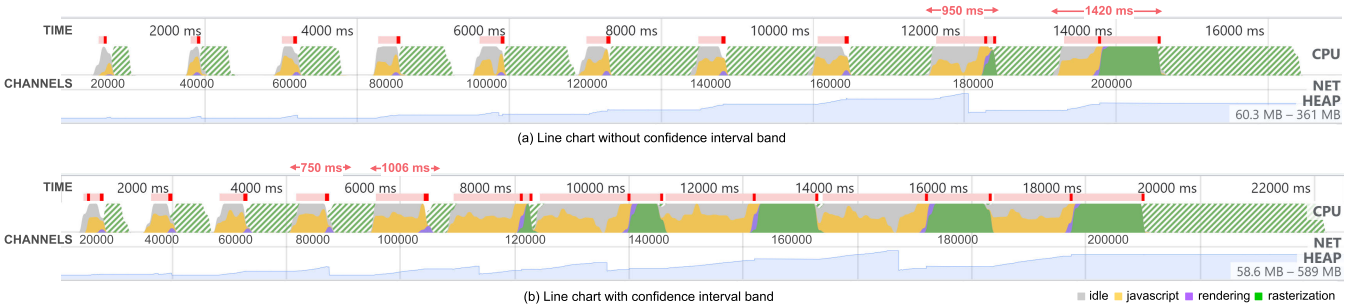

FIGURE 14. Rendering performance (T_{rend}) of the spectrum plots shown in Fig. 8. When there are 180,000 channels it takes ∼950 ms to render a *Spectrum* data into a line chart and for 200,000 channels it takes a ∼1420 ms (a). For (b) which includes drawing of an additional band along with the line chart to show the confidence intervals, the rendering pipeline takes an additional time- it takes ∼750 ms to render power values and confidence interval of 80,000 channels and ∼1006 ms to render 100,000 channels. The rendering latency increases with increase in number of channels, therefore the payloads queue in the browser waiting to be visualized which increases browser heap memory.

visualization critical rendering pipeline [48] performance data.

The performance evaluation was conducted on Ubuntu 22.04 LTS powered by an Intel i9-9900K (8-Core/16-Thread, 16MB Cache, 4.7GHz across all cores), 32GB DDR4 XMP (2933MHz) memory, M.2 PCIe SSD storage, and Gigabit Ethernet network. A Chrome browser and our front-end and back-end services are deployed on this machine. For a production deployment, the back-end services should be deployed on a cluster of virtual machines with auto scaling capability.

VI. RESULTS

In this section, we present the key performance indicators: (a) serialized payload size, (b) transmission latency, (c) deserialization time, and (d) web rendering time (as in Sec. V-A) for the three use cases (as in Sec. V-B).

A. POWER SPECTRUM PLOT

Fig. 13(a) shows the difference in serialization cost between ProtoBuf or JSON encoding of the *Spectrum* data model. It can be seen that ProtoBuf generated payloads are significantly smaller than their JSON counterparts. As the number of channels increases, ProtoBuf's space efficiency improves further compared to JSON. For instance, when serializing 600,000 channels, the size of the JSON payload is approximately twice that of the equivalent ProtoBuf payload. This indicates that ProtoBuf is more efficient than JSON at encoding large amounts of data.

Fig. 13(b) illustrates the time required for the push-based architecture to transmit payloads from the metric generators to the browser. Two principal observations can be made from this data: (i) As expected, the transport time for smaller payloads is smaller than that for larger payloads, and the latency due to transmitting the data increases with the size of the payload. (ii) Additionally, JSON encoded payloads take longer to transport than ProtoBuf encoded payloads. This suggests that ProtoBuf is more efficient at transmitting data compared to JSON, particularly for larger payloads.

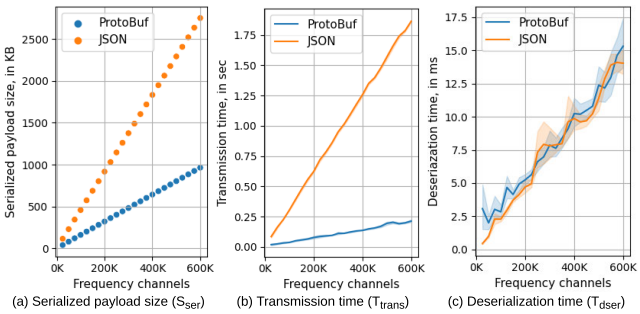

FIGURE 15. Serialization, transmission, and deserialization performance of Protobuf and JSON encoded *Spectrogram* data. (a) Presents the serialized payload size (in kilobytes), (b) presents transmission time (in second) of the payloads, and (c) presents the deserialization time in the browser (in millisecond). The translucent color band shows confidence intervals in 95%.

Fig. 13(c) demonstrates the deserialization time for ProtoBuf and JSON encoded payloads in the browser. It can be seen that the deserialization time for JSON payloads is higher than that for ProtoBuf payloads. This may be because the JSON payloads are larger than the corresponding ProtoBuf payloads, which may lead to longer deserialization times. This suggests that ProtoBuf is more efficient at deserializing data in the browser compared to JSON, particularly for larger payloads.

Fig. 14 presents the web visualization performance of a power spectrum plot with a size of 1200 × 600 pixels. The experiment begins by starting with 20,000 channels and increasing the number of channels by 20,000 every second. The results show that it is possible to render 180,000 data points in less than 1 second for a spectrum plot without confidence interval bands, and 80,000 data points in 750 milliseconds for a spectrum plot with confidence interval bands. This result demonstrates that the visualization is highly scalable and able to handle the approximately 65,000 frequency channels captured by SKA antennas. It is worth noting that the line chart in Fig. 8 is implemented using SVG, but a Canvas implementation of the line chart is expected to be able to handle even more data points.

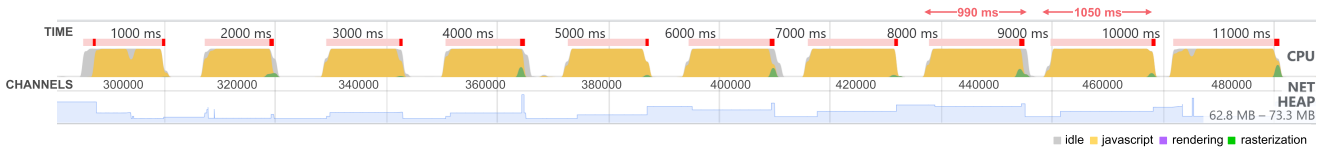

FIGURE 16. Rendering performance (T_{rend}) of a waterfall plot of spectrogram rendering 300000–480000 channels. When there are 440,000 channels it takes ∼990 ms to render a *Spectrogram* data into a waterfall plot and for 460,000 channels it takes a ∼1050 ms.

B. WATERFALL PLOT OF SPECTROGRAM

Fig. 15(a) illustrates the serialization cost when using ProtoBuf or JSON encoded payloads of the *Spectrogram* data model. The results show that the size of ProtoBuf payloads is marginally smaller than that of JSON payloads, only differing by 1∼2 MB at 600,000 channels, as opposed to around 8 MB in the Spectrum data model. This suggests that ProtoBuf and JSON are relatively similar in terms of their serialization efficiency for the Spectrogram data model. It is worth noting that the size difference between the two encoding methods may vary depending on the specific characteristics of the data being serialized.

Fig. 15(b) demonstrates that the streaming cost for JSON encoded payloads is significantly higher than for ProtoBuf encoded payloads. The data shows that it takes nearly 2 seconds to transmit a JSON payload consisting of 600,000 channels, while it takes only 250 milliseconds to transmit a ProtoBuf payload containing the same data. This indicates that ProtoBuf is much more efficient at streaming Spectrogram data than JSON, particularly for large payloads. This could be due to ProtoBuf's use of a binary encoding format, which is generally more efficient for transmission over a network compared to the text-based encoding of JSON.

Fig. 15(c) shows that the deserialization cost of the ProtoBuf and JSON encodings are almost the same. This suggests that there is little difference in the efficiency of the two encoding methods when it comes to deserialization. However, when compared to the results for the spectrum data model in Fig. 13(c), it is worth noting that the specific characteristics of the data being deserialized could affect the relative performance of ProtoBuf and JSON. Additionally, the performance of the deserialization process may be influenced by factors such as the hardware and software configurations of the system on which it is being performed.

Fig. 16 presents the web visualization performance of a spectrogram with a size of 1200 × 600 pixels. The experiment begins with 300,000 channels and increases the number of channels by 20,000 every second. The results show that the spectrogram plot is highly scalable, as it is able to render 440,000 data points with less than 1 second of latency. This demonstrates that the spectrogram visualization is able to handle large amounts of data efficiently. The spectrogram in Fig. 10 is implemented using Canvas graphics which is known for its high performance and is often used for rendering complex graphics like spectrogram.

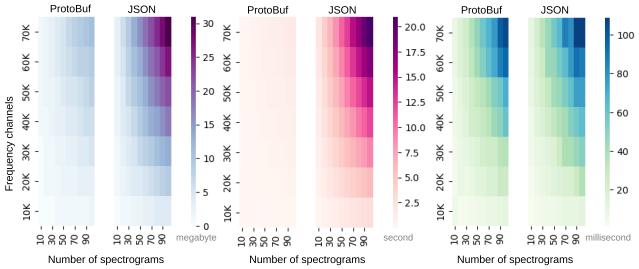

FIGURE 17. Serialization, transmission, and deserialization performance of Protobuf and JSON encoded *Spectrograms* data. (a) Presents the serialized payload size (in kilobytes), (b) presents the transmission time (in seconds) of the payloads, and (c) presents the deserialization time in the browser (in milliseconds). In these heatmaps, the size and time values are encoded to colormaps, the x-axis represents the number of spectrograms, and the y-axis represents the number of channels.

C. SPECTROGRAMS OF BASELINES AND POLARIZATIONS

Fig. 17(a) presents the serialization cost of ProtoBuf and JSON encoding of the *Spectrograms* data model. The results show that, as expected, the size of the ProtoBuf serialized data is marginally smaller than the size of the JSON serialized data. This indicates that ProtoBuf is slightly more efficient at serializing the Spectrograms data model compared to JSON. However, the difference in size between the two encoding methods may be relatively small, depending on the specific characteristics of the data being serialized.

Fig. 17(b) illustrates that the streaming cost for JSON encoded payloads is significantly higher than for ProtoBuf encoded payloads. The data shows that it takes nearly 21 seconds to transmit a JSON payload consisting of 100 spectrograms, each containing 70,000 channels, while it takes only 1.6 seconds to transmit a ProtoBuf payload containing the same data. This demonstrates that ProtoBuf is much more efficient at streaming data than JSON, particularly for large payloads. This could be due to ProtoBuf's use of a binary encoding format, which is generally more efficient for transmission over a network compared to the text-based encoding of JSON.

Fig. 17(c) shows that the deserialization performance of the ProtoBuf and JSON encodings are similar. As expected, similar performance was also observed for the Spectrogram data model.

Fig. 18 shows the web visualization performance for multiple waterfall plots containing spectrograms visualized in a web browser. Each spectrogram has dimension 200 × 120 pixels, and a total of 77 spectrograms are displayed in a tabular view on a single display. The data used to

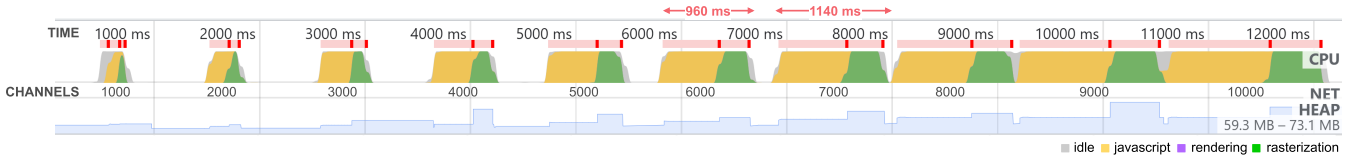

FIGURE 18. Rendering performance (T_{rend}) of a table of 77 spectrograms each rendering 1000 – 10000 channels. When there are 6,000 channels for each spectrogram to draw it takes ∼960 ms to render and for 7,000 channels it takes a ∼1140 ms.

generate these spectrograms consists of 1,000 channels, with the number of channels increasing by 1,000 every second. The rendering of 6,000 points into each spectrogram takes approximately 960 ms. The SKA antennas are expected to generate a total of 65,000 channels, 130,000 baselines, and 4 polarizations, which may require data averaging or subset selection in order to be visualized effectively in the web browser using a table of spectrograms. The spectrograms in Fig. 12 are implemented using the Canvas graphics system. Yup

VII. DISCUSSION

This work is motivated by the need for visual analytic systems capable of visualizing high volume streaming data in a browser in near real-time. A concrete system in need of this capability is the world's largest radio astronomy project, the SKA. As a large volume of data is continuously generated from different antennas and sensors, providing real-time monitoring support for the operators and radio astronomers at the SKA is essential. As discussed in Sec. III, we considered a number of solutions, such as using an off-the-shelf system of the sort used by current radio telescopes. We profiled such a system to identify the degree to which the different components contribute to the overall latency. Based on our profiling, we replaced the transactional database responsible for the latency with a low latency in-memory database. Although the in-memory database improved the I/O latency significantly, retrieval of data in real-time from a database requires continuous polling which contributes significantly to latency.

The need to transport high-volume data in real-time to a web browser led us to propose a push-based streaming data solution. The proposed streaming data system can transport a large volume of data every second to a web browser. We encounter limitations in processing, visualizing and animating the changes for the data rates our push-based streaming solution can supply. That led us to investigate high-performing serialization and deserialization protocols, rendering techniques, and a methodology for measuring rendering performance to find the optimal data rate at which our visualizations can render the supplied data.

A. WHAT DID WE ACHIEVE?

There are a number of benefits of our solution. We can (i) stream data of various types and sizes to the browser with a one-second latency, which conforms to the strict latency requirements of the SKA, (ii) streamline the system development process, for example, isolating metric generators, messaging system, RESTful APIs, visualizations, and user interface components, (iii) identify the key performance indicators and provide a methodology for measuring the run-time performances of these indicators, and (iv) developed an open-sourced tool, available through GitHub [45], that can help rapidly prototype streaming of new data types, and evaluate and optimize their web visualization performance.

B. GENERALIZING OUR APPROACH

While our approach has been motivated by the requirements of the SKA radio astronomy project, the proposed system and approaches are transferable to many other data-intensive settings where real-time visualization performance is crucial. Distributed microservices, IoT systems and sensors, data centers, machine learning models, etc., generate high volume data. Monitoring those systems and sensor logs in real-time is essential for continual operation, debugging, profiling, performance optimization, etc. The streaming data architecture can be adapted to streamline any low latency data processing and transporting pipeline. The methodologies discussed for web performance analysis can help develop and optimize general high-performing web-based visual analytics systems in a variety of domains, including, network monitoring, mass spectroscopy, game analytics, fraud detection, radio communications, energy research, and atmospheric science.

C. LIMITATIONS AND FUTURE DIRECTIONS

One may argue that the measured performance data does not correspond to ground truth. It is hard to generalize because (a) different browsers perform differently in different hardware, and (b) different visualization functions use different data structures, and the runtime performance depends on both. At SKA, the web visualizations will only be accessed from known hardware with a known browser. Measuring performance on the relevant hardware followed by streaming rate optimization a priori is therefore feasible for our system. To generalize the impact of the streaming visualization architecture implemented in this work, an approach where browsers automatically provide performance metrics from which the streaming data rates are adjusted would be suitable. This would allow visualizations using large amounts of real-time streaming data to be accessed from general hardware with optimal performance. A different line of work to pursue further would be to take human visual

perception into account when processing and aggregating metrics, to optimize performance not only with respect to throughput but to the maximal throughput which can be perceived by a user.

We are extending this system to improve the metric generators, which are expected to receive data from different distributed sources. Therefore, distributed data processing capability is essential to handle data from various sources, transform and aggregate such data, and scale the data processing operations. Because the producer and the messaging system, in Fig. 4, are designed to work in a distributed environment, both data processing and streaming system will integrate seamlessly.

VIII. CONCLUSION

In this work, we presented an architecture for streaming high-volume data in the web browser in real-time and proposed methodologies for web visualization performance analysis. Because of the need to visualize various matrices to ensure day-to-day SKA telescope operations, support quality assurance of the telescope instruments, and rapid observation of large volumes of radio astronomy data captured by the telescope, we adopted data streaming architecture. Because of the limitations of the web browser in how much and how frequently it can process and render data, we introduced methods to measure this quantitatively. The streaming and visualization architecture can be generalized to stream various data types from different application domains. The web visualization performance, e.g., deserialization time and rendering time, may vary for data type and visualization functions or charts. Therefore, the performance of each data type and its visualization should be measured separately.

ACKNOWLEDGMENT

The authors would like to acknowledge and thank Adam Campbell, Seth Hall, and Andrew Ensor from the Auckland University of Technology, and Rodrigo Tobar from the International Centre for Radio Astronomy Research, Australia, for developing the first version of the prototype involving Grafana and InfluxDB, measuring its latencies provided as supplementary material, and helping them familiarizing the prototype. They would also like to thank their SKAO colleagues Fred Dulwich, Benjamin Mort, Danielle Fenech, Mark Ashdown, and others for their domain inputs.

REFERENCES

[1] A. Aimar, A. A. Corman, P. Andrade, J. D. Fernandez, B. G. Bear, E. Karavakis, D. M. Kulikowski, and L. Magnoni, "MONIT: Monitoring the CERN data centres and the WLCG infrastructure," in *Proc. EPJ Web Conf.*, vol. 214, 2019, p. 08031.

[2] B. Matthews, "Metadata for information management in large-scale science," *Presentation at MPG EScience Seminar Metadata Infrastruct.*, Berlin, Germany, 2010.

[3] T. Zhang, X. Wang, Z. Li, F. Guo, Y. Ma, and W. Chen, "A survey of network anomaly visualization," *Sci. China Inf. Sci.*, vol. 60, no. 12, pp. 1–17, Dec. 2017.

[4] J. Henning and R. Smith, "A web-based system for creating, viewing, and editing precursor mass spectrometry ground truth data," *BMC Bioinf.*, vol. 21, no. 1, pp. 1–10, Dec. 2020.

[5] L. Kolbowski, C. Combe, and J. Rappsilber, "XiSPEC: Web-based visualization, analysis and sharing of proteomics data," *Nucleic Acids Res.*, vol. 46, no. 1, pp. 473–478, Jul. 2018.

[6] I. K. Choi, T. Jiang, S. R. Kankara, S. Wu, and X. Liu, "TopMSV: A web-based tool for top-down mass spectrometry data visualization," *J. Amer. Soc. Mass Spectrometry*, vol. 32, no. 6, pp. 1312–1318, Jun. 2021.

[7] Y. Zhang, R. Bhamber, I. Riba-Garcia, H. Liao, R. D. Unwin, and A. W. Dowsey, "Streaming visualisation of quantitative mass spectrometry data based on a novel raw signal decomposition method," *Proteomics*, vol. 15, no. 8, pp. 1419–1427, Apr. 2015.

[8] C. Maçãs, E. Polisciuc, and P. Machado, "ATOVis—A visualisation tool for the detection of financial fraud," *Inf. Visualizat.*, vol. 21, no. 4, pp. 371–392, Oct. 2022.

[9] M. Aschi, S. Bonura, N. Masi, D. Messina, and D. Profeta, "Cybersecurity and fraud detection in financial transactions," in *Big Data and Artificial Intelligence in Digital Finance*. Berlin, Germany: Springer, 2022, pp. 269–278.

[10] B. Medler and B. Magerko, "Analytics of play: Using information visualization and gameplay practices for visualizing video game data," *Parsons J. Inf. Mapping*, vol. 3, no. 1, pp. 1–12, 2011.

[11] Y. Su, P. Backlund, and H. Engström, "Comprehensive review and classification of game analytics," *Service Oriented Comput. Appl.*, vol. 15, no. 2, pp. 141–156, Jun. 2021.

[12] A. A. Goodman, "Principles of high-dimensional data visualization in astronomy," *Astronomische Nachrichten*, vol. 333, nos. 5–6, pp. 505–514, Jun. 2012.

[13] J. S. Farnes, B. Mort, F. Dulwich, K. Adamek, A. Brown, J. Novotny, S. Salvini, and W. Armour, "Building the world's largest radio telescope: The square Kilometre array science data processor," in *Proc. IEEE 14th Int. Conf. e-Sci.*, Oct. 2018, pp. 366–367.

[14] G. H. Tan, T. J. Cornwell, P. E. Dewdney, and M. Waterson, "The square Kilometre array baseline design V2.0," in *Proc. 1st URSI Atlantic Radio Sci. Conf. (URSI AT-RASC)*, May 2015, p. 1.

[15] E. Pietriga, G. Filippi, L. Véliz, F. D. Campo, and J. Ibsen, "A web-based dashboard for the high-level monitoring of ALMA," in *Proc. SPIE*, vol. 9152, pp. 461–472, Jul. 2014.

[16] C. Schollar, "RFI monitoring for the MeerKAT radio telescope," Univ. Cape Town, Cape Town, South Africa, Tech. Rep., 2015. [Online]. Available: http://pubs.cs.uct.ac.za/id/eprint/1042

[17] M. Lassnig, "Rucio beyond ATLAS: Experiences from Belle II, CMS, DUNE, EISCAT3D, LIGO/VIRGO, SKA, XENON," in *Proc. EPJ Web Conf.*, vol. 245, 2020, p. 11006.

[18] F. Locci, "CERN controls open source monitoring system," *Prometheus*, vol. 1, pp. 404–408, Jan. 2019.

[19] L. Blackburn, L. Cadonati, S. Caride, S. Caudill, S. Chatterji, N. Christensen, J. Dalrymple, S. Desai, A. D. Credico, G. Ely, and J. Garofoli, "The LSC glitch group: Monitoring noise transients during the fifth LIGO science run," *Classical Quantum Gravity*, vol. 25, no. 18, Sep. 2008, Art. no. 184004.

[20] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proc. 11th ACM Int. Conf. Distrib. Event-Based Syst.*, 2017, pp. 227–238.

[21] G. Hesse, C. Matthies, and M. Uflacker, "How fast can we insert? An empirical performance evaluation of apache kafka," in *Proc. IEEE 26th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2020, pp. 641–648.

[22] E. Rosolowsky, A. R. Taylor, and A. Goodman, "A visualization portal for ALMA data," Univ. British Columbia, Okanagan Campus, Kelowna, BC, Canada; Univ. Calgary, Calgary, AB, Canada, Harvard Univ., Cambridge, MA, USA, Tech. Rep., 2017. [Online]. Available: https://science.nrao.edu/facilities/alma/alma-develop-old-022217/VisualizationPortal.pdf

[23] M. Alberts and F. Joubert, "The MeerKAT graphical user interface technology stack," in *Proc. ICALEPCS*, 2015, pp. 1134–1137.

[24] A. Ledeul, A. Savulescu, G. S. Millan, and B. Styczen, "Data streaming with apache kafka for CERN supervision, control and data acquisition system for radiation and environmental protection," in *Proc. 17th Int. Conf. Accel. Large Exp. Phys. Contr. Syst. (ICALEPCS)*, 2019, pp. 1–5.

[25] F. S. Boukredera, A. Hadjadj, and M. R. Youcefi, "Drilling vibrations diagnostic through drilling data analyses and visualization in real time application," *Earth Sci. Informat.*, vol. 14, no. 4, pp. 1919–1936, Dec. 2021.

[26] N. Iftikhar, B. Lachowicz, A. Madarasz, F. Nordbjerg, T. Baattrup-Andersen, and K. Jeppesen, "Real-time visualization of sensor data in smart manufacturing using lambda architecture," in *Proc. 9th Int. Conf. Data Sci., Technol. Appl.*, 2020, pp. 215–222.

[27] P. Chen, "Visualization of real-time monitoring datagraphic of urban environmental quality," *EURASIP J. Image Video Process.*, vol. 2019, no. 1, pp. 1–9, Dec. 2019.
[28] Y. Ma, Y. Zheng, J. Cheng, and S. Easa, "Real-time visualization method for estimating 3D highway sight distance using LiDAR data," *J. Transp. Eng., A, Syst.*, vol. 145, no. 4, Apr. 2019, Art. no. 04019006.
[29] Z. Lv, X. Li, B. Zhang, W. Wang, Y. Zhu, J. Hu, and S. Feng, "Managing big city information based on WebVRGIS," *IEEE Access*, vol. 4, pp. 407–415, 2016.
[30] A. Protopsaltis, P. Sarigiannidis, D. Margounakis, and A. Lytos, "Data visualization in Internet of Things: Tools, methodologies, and challenges," in *Proc. 15th Int. Conf. Availability, Rel. Secur.*, 2020, pp. 1–11.
[31] C. Malliarakis, M. Satratzemi, and S. Xinogalos, "Integrating learning analytics in an educational MMORPG for computer programming," in *Proc. IEEE 14th Int. Conf. Adv. Learn. Technol.*, Jul. 2014, pp. 233–237.
[32] T. Galati, "Exploring game analytics solutions for data-driven user research in indie studios," Ph.D. dissertation, Dept. Sci., Univ. Ontario Inst. Technol., Canada, 2017.
[33] S. Khan, P. H. Nguyen, A. Abdul-Rahman, B. Bach, M. Chen, E. Freeman, and C. Turkay, "Propagating visual designs to numerous plots and dashboards," *IEEE Trans. Vis. Comput. Graphics*, vol. 28, no. 1, pp. 86–95, Jan. 2022.
[34] R. C. Hoetzlein, "Graphics performance in rich internet applications," *IEEE Comput. Graph. Appl.*, vol. 32, no. 5, pp. 98–104, Sep./Oct. 2012.
[35] *HTML Canvas Graphics*. Accessed: Feb. 15, 2022. [Online]. Available: https://www.w3schools.com/html/html5_canvas.asp
[36] D. E. Kee, L. Salowitz, and R. Chang, "Comparing interactive web-based visualization rendering techniques," in *Proc. IEEE Conf. Infovis*, Jan. 2012, pp. 1–2.
[37] Z. B. Babovic, J. Protic, and V. Milutinovic, "Web performance evaluation for Internet of Things applications," *IEEE Access*, vol. 4, pp. 6974–6992, 2016.
[38] M. Schwab, D. Saffo, N. Bond, S. Sinha, C. Dunne, J. Huang, J. Tompkin, and M. A. Borkin, "Scalable scalable vector graphics: Automatic translation of interactive SVGs to a multithread VDOM for fast rendering," *IEEE Trans. Vis. Comput. Graphics*, vol. 28, no. 9, pp. 3219–3234, Sep. 2022.
[39] S. Lee, J.-Y. Jo, and Y. Kim, "Performance testing of web-based data visualization," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Oct. 2014, pp. 1648–1653.
[40] G. V. Dongen and D. V. D. Poel, "Evaluation of stream processing frameworks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1845–1858, Dec. 2020.
[41] *CBF SDP Emulator Metrics Generator*. Accessed: Feb. 15, 2022. [Online]. Available: https://gitlab.com/ska-telescope/cbf-sdp-emulator-metrics-generator
[42] J. A. Miller, H. Zhu, and J. Zhang, "Guest editorial: Advances in web services research," *IEEE Trans. Services Comput.*, vol. 10, no. 1, pp. 5–8, Jan. 2017.
[43] S. Khan, P. H. Nguyen, A. Abdul-Rahman, E. Freeman, C. Turkay, and M. Chen, "Rapid development of a data visualization service in an emergency response," *IEEE Trans. Services Comput.*, vol. 15, no. 3, pp. 1251–1264, May 2022.
[44] S. Khan and D. Wallom, "A system for organizing, collecting, and presenting open-source intelligence," *J. Data, Inf. Manag.*, vol. 4, no. 2, pp. 107–117, Jun. 2022.
[45] *Streaming Data Visualization and Benchmarks*. Accessed: Feb. 15, 2022. [Online]. Available: https://github.com/saifulkhan/streaming-vis-perf
[46] *Managed Apache Kafka Clusters*. Accessed: Feb. 15, 2022. [Online]. Available: https://www.cloudkarafka.com
[47] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. USENIX Int. Workshop Netw. Meets Databases*, vol. 11, 2011, pp. 1–7.
[48] *Rendering Performance*. Accessed: Feb. 15, 2022. [Online]. Available: https://web.dev/rendering-performance/
[49] *JavaScript Spectrogram Library*. Accessed: Feb. 15, 2022. [Online]. Available: http://arc.id.au/Spectrogram.html
[50] *React—A JavaScript Library for Building User Interfaces*. [Online]. Available: https://reactjs.org
[51] *SKA SDP QA Metric Generator*. Accessed: Feb. 15, 2022. [Online]. Available: https://gitlab.com/ska-telescope/sdp/ska-sdp-qa-metric-generator
[52] *SKA SDP QA Data API*. Accessed: Feb. 15, 2022. [Online]. Available: https://gitlab.com/ska-telescope/sdp/ska-sdp-qa-data-api
[53] *SKA SDP QA Display*. Accessed: Feb. 15, 2022. [Online]. Available: https://gitlab.com/ska-telescope/sdp/ska-sdp-qa-display

SAIFUL KHAN (Member, IEEE) received the D.Phil. degree in engineering science from the University of Oxford. He worked as a Software Engineer at ABB, Oracle, and International Seismological Centre; and a Data Scientist at Horus Security Consultancy. He is currently a Postdoctoral Researcher with the University of Oxford. His research interests include data processing, modeling, search, and visualization techniques for data from various domains, such as, radio astronomy, seismology, building information management, and security-intelligence.

ERIK RYDOW received the M.Sc. degree in mathematical and theoretical physics from the University of Oxford. He is currently a Researcher with the Oxford e-Research Centre. Since 2022, he has been working on the science data processor for the Square Kilometre Array (SKA) Project. He is a member of the High-Performance Computing and Code Optimisation Team. Previously, he worked on using visualization to support sensitivity analysis of epidemiological models.

SHAHRIAR ETEMADITAJBAKHSH received the Ph.D. degree in engineering (communication systems) from The Australian National University. He joined the Square Kilometre Array (SKA) Project as a Postdoctoral Research Software Engineer with the Oxford e-Research Centre. He is currently working on radio frequency interference mitigation for SKA Project. Previously, he worked as a Postdoctoral Researcher with the University of New South Wales and the University of Oxford. Also, he has been a Knowledge Transfer Partnership Associate between Moogsoft and the University of Sussex.

KAREL ADAMEK received the Ph.D. degree in theoretical astrophysics from Silesian University in Opava, Czech Republic, in 2016. From 2015 to 2019, he was a Postdoctoral Research Associate with the Oxford e-Research Centre, working on accelerating algorithms on GPUs for the Square Kilometre Array radio telescope. He is currently a Senior Researcher and the Departmental Lecturer with the Department of Engineering Sciences and a Product Owner of the SKA Software Development Team.

WES ARMOUR received the M.Phys. degree in fundamental particle physics and cosmology and the Ph.D. degree in lattice gauge theory. He is currently the Director of the Oxford e-Research Centre and a Professor of scientific computing with the Department of Engineering Science, University of Oxford. He began his career in theoretical particle physics and then spent several years advancing algorithms and methods in protein crystallography. Then, he moved to the University of Oxford, where he leads the Scientific Computing Group, Oxford e-Research Centre.