Real-Time Processing and Visualization of Intensive Care Unit Data

by

Shruthi Narayanan

B.S., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

© 2017 Massachusetts Institute of Technology. All rights reserved.
Real-Time Processing and Visualization of Intensive Care Unit Data

by

Shruthi Narayanan

Submitted to the Department of Electrical Engineering and Computer Science on July 28, 2017, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Intensive care unit (ICU) patients undergo detailed monitoring so that copious information regarding their condition is available to support clinical decision-making. Full utilization of the data depends heavily on its quantity, quality and manner of presentation to the physician at the bedside of a patient. In this thesis, we implemented a visualization system to aid ICU clinicians in collecting, processing, and displaying available ICU data. Our goals for the system are: to be able to receive large quantities of patient data from various sources, to compute complex functions over the data that are able to quantify an ICU patient’s condition, to plot the data using a clean and interactive interface, and to be capable of live plot updates upon receiving new data. We made significant headway toward our goals, and we succeeded in creating a highly adaptable visualization system that future developers and users will be able to customize.

Thesis Supervisor: Roger G. Mark
Title: Professor of Electrical Engineering and Distinguished Professor of Health Sciences and Technology
Acknowledgments

I am extremely grateful to my supervisor Roger Mark for giving me the opportunity to work with LCP for the past two years. I hadn’t known how much I wanted to work with medical data until Roger’s commitment and dedication to the subject matter inspired me to follow him into the field. I thank him for all the advice and experience he has offered me - I will try my hardest to use that knowledge well and pay it forwards.

I would also like to thank Alistair Johnson for mentoring me throughout my thesis. He entrusted me to figure out challenging issues on my own but always provided advice when I needed it. Thanks to him, I can safely say that my thesis has been the most enjoyable and fulfilling project I have ever worked on.

And, of course, I want to say thanks to all of my family for unquestionably supporting me through all of my struggles and successes. Special mentions to my father, who always, always inspires me to work harder and to be a better person; and my mother, who somehow tricked me into becoming a computer scientist, mostly by being my biggest role-model.
## Contents

1 Introduction .................................................................................. 11
   1.1 Motivation ............................................................................. 11
   1.2 Project Description ............................................................... 13

2 System Design ............................................................................. 15
   2.1 Data Stream ........................................................................ 15
   2.2 Database .............................................................................. 17
      2.2.1 d_patients ...................................................................... 18
      2.2.2 labevents ...................................................................... 19
      2.2.3 vitalevents ..................................................................... 20
      2.2.4 admissions .................................................................... 21
      2.2.5 transfers ....................................................................... 21
      2.2.6 Database - Summary ...................................................... 22
   2.3 Python Code ......................................................................... 23
      2.3.1 DataFetcher ................................................................... 23
      2.3.2 Bokeh Code .................................................................. 26
         2.3.2.1 Plot Class ............................................................... 27
            2.3.2.1.1 Axes ................................................................ 27
            2.3.2.1.2 Plot Variations and Layouts ......................... 30
         2.3.2.2 Dataset Class ........................................................... 30
            2.3.2.2.1 Plotting Logistics: Glyphs ............................... 31
            2.3.2.2.2 Preparing Data for Plotting ......................... 32
            2.3.2.2.3 Using a FuncObj to Calculate Data ............... 33
4 Future Work

4.1 Unresolved Issues

4.2 User-Suggested Improvements

4.2.1 Tester Demographics

4.2.2 Positive Comments

4.2.3 Desired Improvements

4.2.3.1 Data Improvements

4.2.3.2 Interface Improvements

4.3 Further Optimizations and Expansions

4.3.1 Optimizing Datasets for Fixed Plots

4.3.2 Optimizing System Delay

4.3.3 Expanding Input Data Flow

4.3.4 Expanding Server Capability

5 Summary
Chapter 1

Introduction

1.1 Motivation

Patients in intensive care units (ICUs) experience heavy monitoring so that detailed information regarding their condition can be recorded and used to make the best decisions for patient care. However, the sheer volume of the resulting data makes it nearly impossible for the ICU staff to organize and interpret all of it within a reasonable time frame. William Knaus wrote of how in 1978, on his first day as an attending physician in the ICU, he was asked to resuscitate a woman undergoing severe septic shock [1]. He only had minutes to review all of the patient data and come to a decision as to what to do. The data was extremely detailed and complete, but he could not look over all of it in enough time to save the patient. "The quality of my care," he said, "would never be better than the quality of the information behind it," yet he was forced to make decisions without proper support from the available data.

The Knaus anecdote highlights that it is vital to improve data processing and analysis in ICU institutions for the sake of better treatment for patients. However, there are many other potential benefits: for example, physician time is a scarce resource that should be allocated as efficiently as possible, and providing decision support gives physicians the ability to process data much faster and come to a decision in less time. It also should be noted that the ICU discipline accounts for over 1%
of national gross domestic product, which is a massive quantity of funds that ICU institutions need to handle well [2]. Concurrently, the patients paying for ICU services are collectively spending a staggering amount of money. More efficient ICU care may allow for better allocation of expensive ICU resources and less of a financial burden on patients.

ICU data is underutilized on a patient level, but the data also could be made better use of on a larger scale. If a patient of a certain demographic is undergoing a certain condition in one ICU, it is possible that other patients from a similar demographic have experienced or are experiencing the same condition in different ICUs. If the data of the other patients was available to the caretaker of the first patient, it would likely improve the caretaker’s ability to make the best decision regarding the first patient’s care. The concept of using other patients’ outcomes to aid in decision making for a new patient is called decision support [2]. Decision support could be more robust if physicians had access to ICU data collected from multiple institutions, and ideally this data would be available to those physicians at the bedsides of patients for immediate use [1].

Additionally, there are vast differences in common practices and treatments in ICUs across different institutions, and often this is because it is hard for ICU clinicians to remember all of the small details that comprise best practice while they are rushing to treat a patient [3]. One estimate claims that about 75,000 deaths could have been avoided in the United States in 2005 if every state had provided care in the same manner as the best-performing state [4]. There have been a few studies that have attempted to use ICU data to encourage best practices. One such example is a tidal volume (TVe) ventilation study by Bourdeaux et al, where tidal volume is routinely monitored in ventilated patients in the ICU [3]. There was evidence indicating that keeping TVe low in patients was beneficial, but it was difficult for clinicians to constantly calculate the TVe. The study attempted to aid the clinicians by creating a dashboard that periodically recalculated TVe on its own and would display an alarm if the TVe was too low. The alarm gave clinicians an easy indication that something needed fixing without taxing the clinicians with the burden of calculating the value
by themselves.

It is clear from the above examples that ICU data has the potential to spark many improvements in patient care. However, the data needs to be utilized correctly to do so. Full utilization seems to depend heavily on the quantity of data presented to a physician at a patient’s bedside, the quality of the presented data, and the amount of ease in viewing and interpreting the data.

1.2 Project Description

Our project goal was to build a visualization system that can help physicians utilize ICU data more efficiently. The visualization should display a series of plots containing data that physicians would find particularly relevant when treating a patient in the ICU. Since every physician and every patient is different, we want the system to be designed so that the displayed plots could be tuned to show data at the preference of the physician. Additionally, if the data is not easily interpretable or cannot be interacted with, then physicians will not want to work with the system even if the system contains a plethora of useful data. We want to design the user interface of the visualization system with care so that clinicians will be willing to use the interface.

We expect that such a system would be used on a patient-by-patient case, and so the system will only show data relevant to one patient at a time. However, we want users to be able to calculate values for a patient based on data from multiple patients - an example of this would be a mortality score calculated for one patient by using a model trained from many similar patients. Thus, the visualization system should be able to access data from many different patients, and should be capable of expanding to accept data from various sources. This way, the system would have access to a vast quantity of data that could be processed and calculated to display complex values similar to the TVe value that we discussed previously.

The visualization system would not be clinically useful if it could not display the most recent data available. Therefore, the system should be capable of live plot updates upon receiving new data. We were very fortunate to have access to a data
stream containing live ICU data that originates from Beth Israel Deaconess Medical Center in Boston, Massachusetts. The existence of this data stream is due to a collaboration between our laboratory (Laboratory for Computational Physiology, at the Massachusetts Institute of Technology; LCP-MIT), Beth Israel, and Philips, and it provides a valuable source of information for analysis. The data stream contains information from January 2016 onwards and currently acts as our system’s lone source of data. The stream allows our system to visualize new data and implement decision support tools in real time, which could potentially be very powerful when our visualization system is used at the bedside of a patient.

In Chapter 2, we will discuss the implementation details of our proposed visualization system first. Then, in Chapter 3, we will show what our current version of the visualization system looks like and elaborate on the different visualized components. We will review any complications with the system that we could not solve, present the user feedback we received when testing the system, and then outline the major steps for future work in Chapter 4.
Chapter 2

System Design

We will present a high-level overview of the visualization system before describing each part of the system and its implementation in detail. The system diagram is displayed in Fig. 2-1 and can be split into four approximate parts; the data stream, the database, the Python plotting code and the webserver. The data stream sends over ICU data from Beth Israel as soon as the data has been collected and/or entered. The data is pulled from the stream and put into a Postgres database so that the data can be easily accessed in the future. Then, the Python code accesses the data and plots it with the help of the Python-based Bokeh visualization library [5]. Finally, we use a combination of the server capability of the Bokeh library and the Flask web framework to securely serve the plots to a webpage [6]. The server and the Python code handle any user requests to modify the plots.

2.1 Data Stream

The Beth Israel data stream sends patient data directly via a secure VPN to an encrypted server at the LCP-MIT lab. We call this stream the HL7 stream because the data it sends over is in a format called HL7 v2, which is a common format used when encoding electronic messages containing medical data.

The HL7 format combines multiple individual messages for one patient into data packets. Each packet has a header code, which indicates what type of data is expressed
by the messages, and a PID (Patient Identifier) line that contains the basic information of the patient connected to the packet. There are two headers that are used by the data stream. An ADT ("Admission-Discharge-Transfer") header indicates that the HL7 packet contains non-medical patient information. If the full header code is ADT-001, then the packet specifically contains information regarding a patient’s admission into a hospital. An ORU (Observation Result) header contains the medical information associated with a data observation for one patient.

The rest of the lines in the packet communicate different pieces of information. The first message line is always the PID line, which contains information such as name, address, and social security number. In the case of an ADT-001 packet, another line might contain the information from the admittance visit, which could include which doctor(s) the patient saw during their visit. Every message line has an identifier that indicates what information is in the line as well as the order in which the information is listed. Here is an example PID line (with artificial data):

```
PID||PATID1234~5~M11||JONES~WILLIAM~A~III||19610615|M-||C|1200 N ELM STREET~GREENSBORO~NC~27401-1020|GL|(91-9)379-1212|(919)271...
```

From simply looking at this line it is difficult to tell what each section of the line
might represent; therefore, all PID lines are required to have the exact same format and ordering so the information in the line can be easily extracted.

Our HL7 stream has two substreams. We call the first substream the ADT/labs-stream, and the second substream the vitals-stream. The ADT/labs-stream contains all ADT packets from the hospital as well as some ORU-header packets that only contain laboratory observations, which are observations that are taken in the lab from patient samples (i.e. blood, urine). Both the ADT packets and the lab ORU packets are created whenever the hospital staff enters new ADT or laboratory data, which occurs sporadically, and then the packets are sent over the HL7 stream immediately.

The vitals-stream only contains ORU-header packets, and these packets are directly generated from the bedside monitors of the patients in the Beth Israel ICU. Specifically, the stream is hooked up to the monitors of ICU wards CC7C and CC7D, which are located in Beth Israel’s Medical Intensive Care Unit 7 (MICU-7). When the stream is running, it generates packets from the bedside monitors at regular intervals and sends them over. In the past, the vitals-stream packet rate has ranged from one packet per hour to one per minute. Unfortunately, the vitals-stream is not currently running live. However, it was up and running for approximately the first half of 2016. The LCP-MIT server at that receives HL7 stream stores all the data sent to it, so we have access to the vitals data sent in 2016.

The LCP-MIT server parses the data packets from the HL7 stream via a Perl based service which looks for packet headers and line identifiers to properly extract the data. The Perl script then saves the data to a Postgres database - we will discuss how the data is saved in the next section. This script existed previously to this project and so we did not have to write or modify the script in any way.

2.2 Database

The HL7 database, built using the PostgreSQL Database Management System, stores all data that has ever been received from the HL7 stream so that the data can be eventually extracted for plotting. The HL7 database has many tables, but only a few
of them are relevant to us; \texttt{d\_patients}, \texttt{labevents}, \texttt{vitatevents}, \texttt{admissions} and \texttt{transfers}. We will discuss the structure of these tables and how the data from the HL7 stream is formatted to be entered into the tables.

We would like to note that the data does not completely match up across tables; for example, we have basic personal information (name, age, etc.) for many more patients than we have laboratory data for, and we occasionally will have laboratory data for a patient for whom we have no personal information. This data mismatch is influenced by the particular selection of data packets that are inputted into the stream for us to receive.

### 2.2.1 \texttt{d\_patients}

The \texttt{d\_patients} table contains patient demographic information. Every patient that has had data sent through the HL7 stream will have their own row in \texttt{d\_patients}. This information is gathered from ADT packets. If the stream receives a packet that is for a new patient, the Perl script will insert a new row into \texttt{d\_patients} for that patient. If the packet received is for a patient that is already listed in the table, then the Perl script will only update the existing row if any patient information has been changed. \texttt{d\_patients} currently contains information for more than 200,000 patients.

\texttt{d\_patients} has three columns that are of interest to us: \textit{mrn}, \textit{sex} and \textit{dob}. The \textit{mrn} field lists the patient’s Medical Record Number (MRN), a 7-digit number that is a unique identifier for a patient. When other tables in the HL7 database need to associate their information with a certain patient, they will include the patient’s MRN with the information. Therefore, as we will show in future sections, we heavily rely on the MRN of a patient in order to be able to pull all of that patient’s data from the database. \textit{sex} and \textit{dob} are the patient’s sex and date of birth respectively; we use \textit{dob} to calculate a patient’s age. We use this age value along with \textit{sex} and \textit{dob} as basic patient information that could be useful when trying to analyze a patient’s condition.
2.2.2 labevents

The labevents table contains all information regarding any laboratory observation that was sent over by the ADT/labs substream. Every row in the table corresponds to one observation. For example, the ADT/labs-stream may send over an ORU packet that contains the glucose, sodium, and potassium levels of a patient’s blood sample. Each of the three observed values will get its own row in labevents. Currently, labevents holds approximately 3 million different observations from about 10,000 different patients.

One note about the entries in labevents that we hinted at previously is that we may have an observation for a certain patient MRN, but we might not have an entry in d_patients for that same MRN. This is because d_patients data is only derived from the ADT packets, and we might not have received any ADT packets for the patient. While all ORU packets do contain a PID line with some patient information, the Perl script chooses only to create new entries in d_patients from ADT packets, which are guaranteed to have all the information to fill a d_patients row.

The columns of labevents that we will use in future sections are: mrn, observation_identifier, specimen_source, observation_timestamp and observation_value. mrn tells us which patient the observation is from; observation_identifier tells us what type of value was observed. For example, if the observation value type was the sodium level of the patient, then the observation_identifier would equal "LC B A4 SODIUM CCC 2951-2 SODIUM LOINC". This is quite a long identifier, but it is of a standard format used by many hospital databases in the United States. specimen_source tells us where the value was obtained from - for example, it could have been pulled from a patient’s blood sample or from a urine sample. The observation_timestamp is the time the observation was taken, and observation_value is the obtained numerical value.
2.2.3 vitalevents

The vitalevents table is almost identical in structure to the labevents table; the main difference is that vitalevents only incorporates ORU packets sent over the vitals substream. However, as we mentioned previously, the vitals-stream is not currently active. vitalevents contains almost 7 million data observations for about 2,500 patients. The data was taken from the MICU-7 bedside monitors between 2016/1/1 and 2016/6/10. The columns of vitalevents that we use are: mrn, observation_identifier, observation_timestamp and observation_value. Vitals data has no specimen source since we know the data is measured by the bedside monitors.

One issue that we have encountered is that it generally takes a long time to request data from vitalevents; in fact, it takes more time than it would to do the same for labevents. Most queries to labevents return in under half a second, whereas vitalevents queries run for two seconds on average and can take up to half a minute. The reason for this delay is as follows. The queries we tend to run on labevents and vitalevents are of the following format:

```
select * from [labevents/vitalevents]
where mrn = 'someMRN'
and observation_timestamp between 'time-1' and 'time-2'
order by observation_timestamp asc;
```

When running this query, the database manager will filter the rows of the appropriate table by mrn and observation_timestamp (not necessarily in that order) and then the rows will be sorted by observation_timestamp. vitalevents has three times the data of labevents, so it takes longer to perform the filtering step. However, the significant slowdown comes from the fact that vitalevents has data for fewer MRNs than labevents, so vitalevents has on average 2800 observations per MRN as compared to labevents with approximately 300 observations per MRN. If the database manager carries out the above query for vitalevents, it will take quite a while to filter for about 2800 observations and then sort them by timestamp.

To help alleviate the slowdown in executing vitalevents queries, we built a
b-tree index on the \textit{vitalevents.\text{observation\_timestamp}} column. The index helps to lower the time it takes for the database manager to filter and sort by \textit{observation\_timestamp}. However, \textit{vitalevents} queries that use the index still take more time than their counterpart \textit{labevents} queries.

\subsection*{2.2.4 admissions}

The \textit{admissions} table contains information pertaining to a patient’s admission to a hospital - this data is obtained from ADT packets sent over the ADT/labs-stream. The columns we use from this table are \textit{mrn}, \textit{admit\_dt}, \textit{admit\_location} and \textit{fiscal\_number}. \textit{mrn} is the MRN of the patient that was admitted; \textit{admit\_dt} is the date and time of the patient’s admission. \textit{admit\_location} is the name of the hospital bed that the patient was admitted into - this information is important to us if we want to figure out which patient is in a certain MICU-7 bed at the current time. \textit{fiscal\_number} is a number used for billing an individual hospital visit; therefore, it is unique to one instance of a patient’s stay in a hospital. It is also used by the \textit{transfers} table, which we will discuss next.

\subsection*{2.2.5 transfers}

The \textit{transfers} table keeps track of whenever a patient moves from one hospital bed to another. This data is also obtained from ADT packets. Every row in the table represents one patient transfer. The relevant columns are \textit{mrn}, \textit{prior\_location}, \textit{assigned\_location}, \textit{transfer\_timestamp} and \textit{fiscal\_number}. \textit{mrn} identifies the patient who was transferred; that patient was transferred from the bed name stored in \textit{prior\_location} to the bed name in \textit{assigned\_location}. \textit{transfer\_timestamp} indicates the time of the transfer.

We use the \textit{transfers} table to check for two scenarios; whether a patient has transferred into a MICU-7 bed, and whether a patient admitted into a MICU-7 bed has been transferred out of the bed. We need to join the \textit{transfers} and \textit{admissions} tables in order to perform the checks. We could use the patient’s MRN as the join
key but the MRN is not unique for a patient visit. If a patient has visited multiple
times in the past, the transfers during older visits do not matter to us and we do
not want to deal with a joined table that includes old transfers. Instead, we join on
the fiscal_number column, which is guaranteed to uniquely identify a patient’s visit.

### 2.2.6 Database - Summary

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Description</th>
<th>Relevant Column Names</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>d_patients</strong></td>
<td>personal patient information</td>
<td>mrn, sex, dob</td>
</tr>
<tr>
<td><strong>labevents</strong></td>
<td>laboratory results</td>
<td>mrn, observation_identifier, specimen_source, observation_timestamp, observation_value</td>
</tr>
<tr>
<td><strong>vitalevents</strong></td>
<td>vitals data from ICU bedside monitors</td>
<td>mrn, observation_identifier, observation_timestamp, observation_value</td>
</tr>
<tr>
<td><strong>admissions</strong></td>
<td>data relating to patient admissions into hospital</td>
<td>mrn, admit_dt, admit_location, fiscal_number</td>
</tr>
<tr>
<td><strong>transfers</strong></td>
<td>lists when patients transfer from one hospital bed to another</td>
<td>mrn, prior_location, assigned_location, transfer_timestamp, fiscal_number</td>
</tr>
</tbody>
</table>

*Table 2-2 - Summary of all database tables and their relevant columns.*
Table 2-2 contains a summarization of the various database tables that we make use of. We have included a short description for each table along with the column names for that database table. Of these five tables, **labevents** and **vitalevents** contain the patient data and are therefore the tables we query the most, but the remaining tables provide valuable auxiliary information for our visualization system.

### 2.3 Python Code

The Python code is responsible for pulling data from the Postgres database and displaying the data on a webpage by using the Bokeh visualization library [5]. Bokeh has the capability to create clean plots that can interact with in many ways and that can handle large amounts of data without slowing down. The code can be approximately split into three modules: the object that handles the data fetching, the Bokeh code that handles plot creation and display, and a thread that runs the data fetching and plot updating in real time.

#### 2.3.1 DataFetcher

The DataFetcher (DF) object handles all connections with the Postgres database and locally stores the obtained Postgres data. When initialized, DF objects set up a connection with the database through the Python psycopg2 package, which allows a DF object to create cursors that can make requests of the database. The connection parameters are specified in a separate configuration file. After the connection is made, any DF method can use the connection to query the database.

Every DF object is associated with exactly one Medical Record Number (MRN), which as we mentioned before is a number that uniquely identifies a patient. The majority of the data stored by a DF object is the labs and vitals data of that patient. Laboratory data (labs) is pulled from the **labevents** table in the database, and vitals data is from the **vitalevents** table. We decided to restrict DFs to storing only one MRN's data at a time because we expected that a user would never need to look at two patients’ data at the same time. If the user wants to switch between looking at
different patients, the DF object can change its associated MRN and will begin to store that MRN’s data instead.

The DF storing mechanism works as follows: when a DF object is first initialized with an MRN or given a new MRN, the DF resets by clearing all of its stored data, which is contained in the DF.\texttt{all\_data} attribute. Fetches are triggered by calling DF.\texttt{fetch\_data()}. When the DF is first commanded to fetch data from the database after resetting, it will pull all labs and vitals data observations that were taken before the current time and then store the data in the \texttt{all\_data} attribute. From then onwards, the DF will keep track of the last time it was asked to fetch from the database, and any subsequent calls to \texttt{fetch\_data()} will only ask for data observations taken between the last fetch time and current time. The data fetched in subsequent calls is added onto the data already stored in \texttt{all\_data}, so that \texttt{all\_data} contains all data that has ever been fetched for one MRN. However, \texttt{fetch\_data()} will always return the new data it has just fetched as opposed to simply returning \texttt{all\_data}. This is because Bokeh plots store all the data they are displaying, so updating most plots can be done solely with the newly fetched data. However, some plots might want to display values that require a patient’s full data history to be computed properly. These plots can then access DF.\texttt{all\_data} directly.

One important consideration when designing the DF object was that we noticed the initial fetch after a DF reset took a very long time, which resulted in a slowdown of all other operations. In particular, we found that the execution of the vitals fetch query would slow user interactivity with the webpage. To counteract this delay, the fetch method requests and waits for all fetch queries in a separate process. While the process separation significantly improved the speed of all Bokeh operations that were experiencing slowdown, it introduced many other issues. One example was that psycopg2 connections cannot be shared across multiple processes. Thus, we created a separate connection object for the fetch method to use, and all other methods use the DF object’s main connection. The other issues will be discussed in a future section, since they relate to objects and operations that have not yet been introduced.

Currently, the DF object has two other methods that fetch from the database.
The first method, `get_basic_info()`, queries the database for personal information associated with an MRN. The information obtained includes the patient’s date of birth (DOB) and sex, pulled from the `d_patients` table, and the patient’s admit date from the `admissions` table. It is possible to pull more information than this: the database also includes, for example, patient name. However, we have chosen to only pull information that doesn’t allow for immediate identification of the individual in question since this data will be displayed on a webpage.

The second method, `update_bed_map()`, queries for the MRNs of the patients that are currently in a given ward of the hospital and creates a map of bed names to MRNs. The visualization system is intended for use in ICU wards and we wanted the user to have access to the information for multiple beds in an ICU ward so that they can switch between the data of patients in different beds easily. By storing the MRNs of every bed in a wing, we can quickly switch a DF object’s MRN to begin pulling the data of a patient in a nearby bed. This bed MRN information is not located in just one table; it is split between `admissions`, which lists the bed a patient was admitted into, and `transfers`, which records when patients have been transferred from one bed to another. We had to combine the information from those two tables to determine which patient is located in a bed at the current time.

All of the data saved by a DF object is stored in Pandas DataFrame objects. Pandas is a commonly used Python library for data analysis, and DataFrames is Pandas' main data structure: it stores rows of multi-dimensional tuples. Whenever a psycopg2 cursor executes a given query, the results are returned in a DataFrame. These DataFrames contain rows of the queried table and column titles, which is why DataFrames are the preferred data structure over other options such as nested lists or NumPy arrays (another data analysis structure) that do not preserve row numbers or named columns. DataFrames can be easily manipulated to retrieve subsets of data, which plays an important role when the data is used by the Bokeh code to produce plots.

When designing the DF object, we wanted to make sure that it would be safe to use one DF across multiple threads. It is possible that different threads may want
to call different DF methods or access DF attributes at the same time; however, certain methods and attributes can interfere with each other. For example, we do not want DF.all_data to be accessed by a plot at the same time that DF.fetch_data() is appending to DF.all_data. We therefore created a Python threading.Lock object, stored in the DF.data_lock attribute, that is used to lock access to certain functions and attributes. We use DF.data_lock to specifically restrict access to DF.all_data and the DF’s MRN value.

2.3.2 Bokeh Code

The code responsible for managing the data plots is heavily reliant on the Bokeh visualization library. Bokeh provides extensive flexibility in plot creation and layouting, and allows plots to be updated live. More importantly, Bokeh automatically handles user interaction with plots efficiently and supplies many accessory tools to customize interactivity. Therefore, we did not have to code any of the user interface from scratch, which was a huge benefit. Bokeh can also serve plots to a local webpage. We found that Bokeh fit the project’s needs very well.

We wrote four modules that together are responsible for handling the plots and the UI. The Plot class handles creating plots and provides functions to aid with regular updates of new data. The Dataset class is responsible for a specific set of data points: for example, one Dataset object might handle the data for patient glucose levels, while another may keep track of heart rate. The file components.py is a set of functions that create different accessory tools for the plots, and in particular includes a function that assembles all of the components into the final layout of the webpage. Lastly, the file stream_plot.py uses the Plot class to create plots with the desired specifications. stream_plot.py is intended to be heavily customizable by users who wish to easily change the plots displayed on the webpage.

In the following sections, we will interchangeably discuss Bokeh classes and classes written by us that may have the same name, such as Plot. To distinguish, all Bokeh classes and methods will have the "bok-" prefix: bok-Plot refers to the Bokeh Plot class, whereas Plot refers to our class. The lowercase version (plot) of a class (Plot)
will refer to the representation of the object on the webpage.

2.3.2.1 Plot Class

The Plot class is a wrapper object for the bok-Plot class. Every Plot object creates a bok-Plot object on initialization and sets basic attributes of the bok-Plot like the plot height. Plot also sets and keeps references to the bok-axis objects of the bok-Plot, which are responsible for the x and y axes of the displayed plot. Plots can be initialized to have two y-axes as opposed to just one, or can split one plot into two subplots. Since each Plot can represent many separate Bokeh objects, Plot also has a layout attribute that contains the full webpage layout of all Plot-related objects. For reference, a diagram of a Plot object and all of its referenced objects is shown in Fig. 2-3.

2.3.2.1.1 Axes

We chose to design our plots so that the x-axis reflects time; Bokeh allows for this by converting times to their equivalent Unix timestamp (seconds since the epoch) before plotting them, but Bokeh keeps the axis labels in readable time formats. On the other hand, the y-axis is a generic axis with no units since the user might want to plot data points with different units (such as heart rate in beats per minute and temperature in Celsius) on the same plot. The bok-axis objects were very useful to us since we needed to update the axis boundaries over time. A bok-axis has start and end attributes that can be set or reset at any time and the bok-Plot will immediately reflect those changes on the webpage. Bokeh automatically sets the x-axis and y-axis start and end attributes so that all of the plot data is visible: however, we decided to overwrite this behavior because it did not exactly suit our needs. For example, if a patient has current data but also has data from a visit that occurred a year ago, we do not want the plot to expand its view to accommodate for the year-old data. The Plot object has multiple attributes and functions that ensure the bok-axis objects display the correct behavior.
Figure 2-3: A diagram showing all of the objects that are connected to one Plot object. Bokeh objects are indicated with a "bok-" prefix. Objects with dashed outlines may not exist depending on how the Plot object is initialized.

One issue with the automatic axis updates was that any update would scale an axis just enough to include a new data point but no more than that. From a user viewability standpoint, it is difficult to see points plotted on the edges of a plot, so we wanted to include a small margin on both ends of all axes. The Plot class has two functions update_x_axis() and update_y_axis() that handle all axis scaling: they are called approximately once every five seconds. Both of these functions add a margin 1/10th the size of the intended axis range to the start and end of every axis.

We thought that it would be unreasonable to display the entire time range of one patient’s data because some of that data could have been taken years ago and might be currently irrelevant. Instead, we decided to limit the x-axis to displaying a few days’ worth of data immediately before the current time. The specific number of days shown is set to three by default, but the user can modify the plot to show anywhere between one and ten days of data via a slider object. The slider is created by the Plot class as a bok-Slider attribute, which calls update_x_axis() to scale the x-axis whenever the slider value is modified.

Since update_x_axis() adds a margin onto the end of the x-axis, it is hard to tell exactly where along the x-axis the current time lies. To visually indicate this,
we added a bok-Span attribute to Plot that draws a vertical line on the plot at any specified location. The location of the vertical line (span) is updated to the current time whenever `update_x_axis()` is called. It is worth noting that we could update the span location very often so that it keeps in sync with the current time, but the Bokeh overhead for doing so would be unnecessarily large and would likely slow down interactivity on the webpage.

The `update_x_axis()` function constantly updates the x-axis range as time passes but keeps the size of the range finite with respect to the bok-Slider. By contrast, the `update_y_axis()` function only updates a Plot’s y-axis object whenever a new data point is added that lies outside of the previous y-axis range (not including margins). This behavior can be disabled by initializing a Plot to have a fixed y-axis range that will never update. A fixed y-range may place outlier values outside of the view of the plot, so we chose not to make a fixed range the default behavior.

Bokeh provides a toolbar that contains many plot tools for user interaction with the webpage plots. Many of these tools allow zooming in and out or panning to show different areas of the plot. While this high level of interactivity is one of the reasons we chose to use Bokeh, it also interfered with our desired axis behavior. If a user had decided to pan back to an earlier set of dates from a year ago, within five seconds the plot’s `update_x_axis()` function would be called and the x-axis range would be forced back to current time. We decided to fix the forcing issue by keeping track of whether the plot was in "standard view" or not via a `standard_view` attribute. The standard view is defined as the axes ranges that would exist if `update_x_axis()` and `update_y_axis()` were allowed to run. If `standard_view` is set to True, then the x-axis will update constantly with time and the y-axis range will include all data points. As soon as a user starts to use a zoom or pan tool, `standard_view` is set to False, and the axis update functions will not be allowed to run. The user can re-enter standard view by pressing a reset button on the toolbar which in turn sets `standard_view` to True.
2.3.2.1.2 Plot Variations and Layouts

Plot objects generally create one bok-Plot instance with one x-axis and one y-axis. There are two alternative options that can be chosen at Plot initialization. The first is to create a single plot with two y-axes, with one y-axis on the left and one on the right. The double-y-axis plot is intended for use cases where a user may want to plot two types of values on the same plot but those values generally do not occupy the same range. For example, if a user wanted to plot creatinine levels (typically ranges from 0.5-1.2 mg/dL) and platelet count (range 150-400 K/uL) on the same plot, they would assign creatinine to one y-axis and platelet count to the other. However, it is possible that a user would like to plot two value types with different ranges and view them together, but would not require them to be on the same plot. In that case, Plot can be initialized to create two smaller plots in the space where one plot would normally be on the webpage. Each subplot has its own bok-Plot instance. The subplots share the same bok-axis object for the x-axis but have separate y-axes and are vertically stacked on each other.

When Bokeh assembles the full layout of the webpage, it places Bokeh objects inside bok-Layout objects and then orders the bok-Layouts in the desired layout. Since a single Plot can be in control of many Bokeh objects, it is easiest for Plot to create a bok-Layout attribute `layout` on initialization that contains its bok-Plot(s) and slider. The code responsible for putting together the full layout (discussed later) can then simply reference the `layout` attribute. This method of assembling the layout would scale well in the future if someone wanted to associate more Bokeh objects with a Plot.

2.3.2.2 Dataset Class

The Dataset class is responsible for handling the data that is plotted on a bok-Plot. An individual Dataset deals with one type of data value. Every Dataset object is associated with one bok-Plot object and therefore one Plot object, but Plots can have multiple Datasets. If a user wanted to have a plot that plotted glucose, sodium
and chloride values, the representative Plot object would have three Datasets, one for each value type. The relationship between a Dataset and its Plot can be seen in Fig. 2-3, along with the objects that a Dataset keeps references to.

### 2.3.2.2.1 Plotting Logistics: Glyphs

First, we will explain how a bok-Plot goes about plotting data. Bokeh creates plots by drawing glyphs on a plot grid at specified locations. The specific form, size and color of the glyph is specified by a bok-Glyph object. Every bok-Glyph has an underlying data structure called a bok-Source, which contains a list of plot coordinates along with any other relevant information associated with each coordinate. When a bok-Glyph is added to a bok-Plot, the plot draws the representative glyph of the bok-Glyph at every coordinate listed in the bok-Source. Thus, one bok-Glyph controls many individual glyphs on the plot, and that set of glyphs is the webpage representation of the Dataset.

Every Dataset object takes in a bok-Plot as an initialization argument, then creates a bok-Glyph and bok-Source specifically for that data value type and adds them to bok-Plot. Dataset also takes a name argument that characterizes the Dataset’s value type (e.g. "glucose"). Once the initialization is completed, updates to the plotted data can be made by streaming new data to the bok-Source. Additionally, all glyphs associated with a Dataset can be removed by clearing its bok-Source.

The default shape for a glyph is a circle. However, if the glyph’s Plot object has two y-axes and is intended to be plotted on the second axis, we use an inverted triangle glyph instead. The color of the glyph varies by Dataset and is provided as an initialization argument. We mentioned earlier that we pull two categories of data from the database: labs and vitals. Labs data tends to be sporadically taken, whereas vitals data is taken at regular intervals (such as every hour or every minute). Therefore, we have chosen to plot labs data as a scatter plot of circles, whereas vitals data glyphs are smaller circles with lines connecting consecutive glyphs.

bok-Glyph objects can be associated with a bok-Hover tool, which creates a popup box with relevant information whenever a user mouse-hovers over a data point on the
Datasets create a bok-Hover tool and add it to the bok-Glyph object. The popup box contains the time and value associated with the data point. If the underlying bok-Glyph represents a Labs dataset then the box will also contain the data point’s specimen source, which shows where the lab value was acquired from (e.g. a blood sample). The box is useful when a user quickly wants to see when a value was taken as opposed to trying to figure out the time from the x-axis, or when there is more data associated with a point then can be displayed on a plot (such as the specimen source).

### 2.3.2.2.2 Preparing Data for Plotting

Datasets require information about how new data from the database should be processed before the new data is streamed to the bok-Source. The DataFetcher (DF) object blindly fetches all labs and vitals data for one MRN: it does not distinguish between different value types of data. However, all pulled data comes with a `observation_identifier` label that lets users know what type of value each data point is. For example, if we were looking through the `labevents` table for glucose values, we would check rows of the table where `observation_identifier` equals "LC B A1 GLUCOSE CCC 6777-7 GLUCOSE LOINC". Every value type has a similar identifier.

There are two ways that this information can be passed to a Dataset. The first, which is the simplest way, is by providing the identifier string for a value type. A Dataset takes the correct `observation_identifier` as an argument and stores it as an `identifier` attribute. For convenience, we do not require that the full identifier be passed in; we only need a substring of the identifier that is guaranteed to be unique. In the glucose case, we could use the sub-identifier "A1 GLUC". The `identifier` must be accompanied by information regarding which category of data it applies to (labs or vitals); the category information is stored in a `id_type` attribute. Once a Dataset has an `identifier`, new data can be filtered using `identifier` to only keep values of the desired type; then those values alone are streamed to the Dataset’s bok-Source.

Sometimes, a Dataset value type may not exist in the database because it needs to be calculated from a combination of value types. One example of this is the ratio
of the partial pressure of oxygen in the blood (PO2) to the fraction of inspired oxygen (FiO2). This ratio is clinically relevant because if a patient’s inspired oxygen is high but their blood oxygen levels are low, the PO2/FiO2 ratio will be very low, which could indicate subpar lung function. PO2 and FiO2 values are both provided by the labevents table, but the ratio needs to be calculated from those values. An identifier will not suffice here. Instead, a Dataset can take a FunctionObject (FuncObj) that will perform the necessary calculations on the data.

2.3.2.2.3 Using a FuncObj to Calculate Data

A FuncObj is an object with a method `func()` that, when run, returns data that has been processed and computed correctly. `func()` is called after every time a DF pulls new data, and func takes that DF and the latest pulled data as arguments so that it can use the latest data or directly access DF.all_data. `func()` can choose to access all previously pulled data via DF.data. The FuncObj can define any other helper methods it may need and it can store any attributes it might need across multiple computations. The only methods it is required to have are `func()` and `obj_reset()`. The `obj_reset()` method is called whenever internal attributes of the FuncObj need to be cleared. For example, a FuncObj might need to store the last time a FiO2 value was taken for a certain MRN. If the MRN of the DF is changed, then this time value needs to be reset.

The FuncObj is very versatile. It allows for simple computations across values, such as in the PO2/FiO2 example: a FuncObj responsible for calculating this ratio only has to filter out PO2 and FiO2 values from the DF data and then perform a rolling computation of what the ratio would have equaled as time had progressed. However, it can be used for much more complex operations; if a user wanted to run a machine learning algorithm over the data and plot its results, they would be able to do so with a FuncObj.

Both the Dataset.identifier and the Dataset.func_obj provide instructions for how to process any new data that is streamed to the Dataset’s bok-Source. However, Dataset is not responsible for the update itself. A separate section of code is respon-
sible for using the FuncObj or identifier to perform the streaming update; this code will be examined in later sections.

### 2.3.2.2.4 Relationship to Plot Object

Datasets are created via the `Plot.add_dataset()` function. `add_dataset()` first needs to check which bok-Plot and which y-axis to add the new Dataset to. If the Plot object in question has subplots, then the Dataset could be added to either of the two subplots. Similarly, if the Plot object has two y-axes, the Dataset’s underlying bok-Glyph object needs to be assigned to one of the two y-axes. `add_dataset()` takes in arguments that specify which subplot or y-axis a Dataset should be added to, and then passes in the appropriate bok-Plot and bok-axis objects as arguments when initializing the new Dataset. Lastly, `add_dataset()` adds the new Dataset to a Plot attribute `datasets` that is a list of all Datasets associated with the plot.

A Dataset keeps and constantly updates a reference to the minimum and maximum values in its bok-Source. The min/max data is then used by Plot whenever Plot has to update its y-axis range during a call to `update_y_axis()`. As we described in the Plot section, `update_y_axis()` scales its range to include all values on the plot. This computation is made much easier because of the existence of Plot. `datasets`. `update_y_axis()` simply loops through each Dataset and its min/max data to find the overall minimum and maximum of all of the plot’s data points.

### 2.3.2.2.5 Custom Datasets

Occasionally, a user may find that they want to plot some data that cannot be fully represented by a circle glyph alone. For example, perhaps a dataset represents the mean value of a collection of data over time, and a user would like to add an error bar to every data point that represents the standard deviation of the collection at that time. A normal Dataset is not capable of handling this scenario because Datasets can only create one bok-Glyph. An error bar is a different Bokeh object that needs to be drawn on the plot separately. Additionally, the bok-Source for this dataset would need to take in three values per timestamp (lower value, middle value, higher value)
as opposed to just one for a normal bok-Source.

In order to allow users to create datasets that do not fit the standard Dataset model, we created a CustomDataset class that inherits from Dataset. Any user can then write their own class that inherits from CustomDataset, and that class can modify the normal Dataset behavior as desired. The one requirement for a CustomDataset is that it must have a `custom_update()` function that dictates how to update the dataset. As mentioned previously, the bok-Source associated with a CustomDataset may be structured differently than a normal Dataset. Therefore, the code section that normally deals with streaming new data to each Dataset’s bok-Source will not be able to stream to a CustomDataset. `custom_update()` therefore must handle the processing of the new data and then stream it in the format that the CustomDataset requires.

### 2.3.2.3 Components

The `components.py` file (`comps`) sets up all of the non-plot components of the webpage. Every component has a function in `comps` that creates the component and returns it as a bok-Layout object. One example component that we decided to create is the information box. It lists the MRN, age, sex and date of birth of the patient that the webpage is currently displaying data for. The other components that we designed will be described in detail later. There are two other functions in `comps` that are important for setting up the webpage. The function `get_datafetcher()` initializes a DataFetcher object and feeds it the correct connection parameters for properly connecting with the database. The other function, `setup()`, is responsible for calling all of the component creation functions and assembling the full webpage layout.

The layout is essentially a collection of Bokeh objects ordered as they are intended to appear on a webpage. Bokeh provides functions to organize bok-Layout objects into rows and columns that can be nested to achieve the proper order. Since we wrap all of our components in bok-Layout objects and all of our Plots have a Plot.layout attribute, we can simply arrange the components and plots into the final layout of our choice.
Once the layout is created and stored in a bok-Layout object, it needs to be prepared to be served as a webpage. To explain how this works, we need to first introduce the bok-Document (bok-Doc) class. The bok-Doc is a container that can hold any Bokeh object that has a JavaScript representation and can be displayed on a webpage. It listens for user interactions on the webpage and carries out the appropriate responses to those interactions. Most of those responses are default reactions from the bok-Doc, like when a zoom tool is used on a plot and bok-Doc changes the axes ranges of that plot, but Bokeh allows us to add UI responses and Python callbacks to almost any webpage activity. bok-Docs can also run operations on Bokeh objects that are requested by the Python code (as opposed to requests from the webpage). The main responsibility of the bok-Doc is to handle both webpage and Python requests for Bokeh object manipulation in an efficient manner.

The bok-Layout that `setup()` generates contains all of the Bokeh objects we want to appear on the webpage. To convert the layout to a JavaScript webpage, the layout simply needs to be added to a bok-Doc (see Fig. 2-4). The bok-Doc then converts the bok-Layout to the webpage layout. `setup()` does not create a bok-Doc, but takes one as an argument and adds the fully assembled layout to it.

`setup()` performs two final operations on the plots that can only be executed after the layout is added to the bok-Doc. As mentioned before, the bok-Doc can listen to events from the entire webpage, and certain user interactions can only be detected through the bok-Doc listener. One example of such an interaction is the pressing of the reset button that switches a Plot back to standard view. `setup()` is responsible for adding a callback to the bok-Doc listener to handle the reset button behavior. Then, `setup()` adds legends to every bok-Plot. We chose to wait to perform this operation until after adding the layout to the bok-Doc because new Datasets cannot be added to Plots that are in the bok-Doc layout. Every y-axis associated with a Plot has its own legend, which lists the names and representative glyphs of each Dataset attached to that y-axis.
Figure 2-4: A diagram outlining the connection between a webpage and its plots/components. Bokeh objects are indicated with a "bok-" prefix. The Bokeh server connects exactly one webpage to each bok-Doc.

2.3.2.4 StreamPlot

The *stream_plot.py* (*splot*) file is the main code file. It builds the interactions between all of the separate Python code sections that have been discussed up until this point. Most importantly, it is the file that is used to serve the webpage.

*splot* creates the bok-Doc that is eventually passed to *comps.setup()* for final setup. Bokeh can serve that bok-Doc to the local web via a Bokeh server, which can be started up by calling the *bok-server* terminal command with *splot* as an argument. A Bokeh server works by connecting to a local URL and every time a GET request is made to that URL, the server runs *splot* and generates a bok-Doc. From then on, the server is responsible for catching the user interactions on the displayed webpage and sending those interactions to the webpage’s underlying bok-Doc for handling. Once the bok-Doc has finished handling the interaction, it will communicate with the server to update the webpage accordingly.

It is important to note two key behaviors of the Bokeh server. The first behavior is that if the server URL is accessed by multiple users, each user will be viewing a different bok-Doc. This behavior, which is depicted in Fig. 2-4, is favorable
because any view changes or tool manipulations performed by one user will not affect other users. Secondly, the Bokeh server is only capable of serving to the local web (localhost). In order to serve to the general web, we will need to augment the Bokeh server with the Flask web framework, which we will discuss in a future section.

Once splot has initialized a bok-Doc, splot proceeds by creating a DataFetcher (DF) object using the `comps.get_datafetcher()` method. Any single bok-Doc is only associated with one DF, and similarly a DF only fetches data for one bok-Doc. splot is also responsible for creating all Plot objects that are intended to be displayed on the webpage. As discussed previously, Plots are made by initializing new Plot objects and adding Datasets to them via the `Plot.add_dataset()` method. Currently, splot creates five plots of our desired specifications. However, the code can be easily modified to add new Plots or remove unwanted ones. Once the Plots are created, splot calls `comps.setup()` to compile the full layout, and `setup()` takes splot’s bok-Doc and Plots as inputs.

Lastly, splot sets up and starts the thread that runs the DF and manages the connection between the DF and the bok-Doc. So far, we have discussed the fetching mechanism of the DF, but we have made no mention of how the data is fetched or how the fetched data is then sent to the bok-Plot objects that are now part of the bok-Doc. splot has a function `run_plots()` that loops at regular intervals. At a high-level of understanding, each iteration of the code loop tells the DF to fetch new data, pulls the new data from the DF and updates every Dataset of all Plots with the new data. `run_plots()` is then added to a Python Thread object as its run target, and the thread starts the code loop. However, as we will discuss in the next section, the thread has to deal with simultaneous operations that add significant complexity to the code loop.

### 2.3.3 Code Loop

As we stated in the last paragraph, the code loop is responsible for instructing the DataFetcher (DF) to fetch new data at regular intervals. Once the new data is returned, the code loop then updates all Plots by calling an `update()` function that pro-
cesses the data appropriately for each Dataset and streams the data to each Dataset’s
bok-Source. These operations, on their own, are fairly simple to perform and present
no significant complications. However, on occasion a webpage user may want to
switch to looking at data from a different patient, and will therefore cause a DF
Medical Record Number (MRN) change. MRN changes require that a reset() func-
tion be invoked, which clears all data and all plots. The reset needs to be handled
correctly even though it could be triggered at any stage of the code loop. It is this
reset operation that poses major complications to the code loop.

We will provide necessary insight regarding how Bokeh handles simultaneous oper-
ations. Then we will describe what the functions reset() and update() are responsible
for. Once this background information has been explained, we will go into detail
how the code loop, the DF, update() and reset() all accommodate for each others’
operations.

2.3.3.1 Bokeh Operation Processing

Whenever a Bokeh server is started up, a thread is created that handles all bok-Docs
initialized by that server. All changes to objects in the server’s bok-Docs can only
happen from functions run in this thread, which we will call the Bokeh thread. Most
webpage user interactions that have callbacks attached to them are communicated
straight to the underlying bok-Doc, so those callbacks are run on the Bokeh thread.
This includes manually added callbacks such as the one comps.setup() attaches to
the bok-Doc. However, the Python code might want to run a function on bok-Doc
objects at Python-defined times - update() is a good example of this since it needs
to update Plot data. update() calls are triggered by the code loop thread, which
means that the code loop thread has to somehow push the call to the Bokeh thread
to be run properly. The bok-Doc provides a function for just this purpose: bok-
Doc.add_next_tick_callback() takes a function as an argument and will run that
function during the next free time slot (tick) of the Bokeh thread. update() and
reset() are always called either from bok-Doc callbacks or called from the code loop
via bok-Doc.add_next_tick_callback(), so they are guaranteed to run in the Bokeh
The existence of separate Bokeh and code loop threads means that Bokeh interaction callbacks can be run when the code loop thread is idling and vice versa. However, it is important to note that both threads are still run by the same process. If the code loop is running a computationally intensive operation that blocks the process, then the Bokeh thread cannot handle user interactions over the duration of that operation. If a webpage user tried, for example, to zoom into a plot at the same time as an intensive code loop operation, they would have to wait for the operation to complete before the zoom in would take effect. This delay is highly undesirable: the user should never have to wait for simple UI interactions to complete. Therefore, it is critical that the code loop should always perform blocking computations in a separate process. The DF.fetch_data() operation is the only function run by the code loop that has demonstrated long blocking times. As we have explained previously, DF objects perform data fetches in a separate process so that the fetches do not block the Bokeh thread.

We would like to note that the Bokeh server creates a new bok-Doc and spins up a new code loop thread every time the URL is accessed, but all bok-Docs run their callbacks in a single Bokeh thread. If the server is being accessed by five different users, then there are five code loop threads and one Bokeh thread running within the same process. Thus, if the code loop threads use up too much of the process time or if they block for long periods of time, the webpage interactions for all five users will slow considerably. We would like to avoid this situation as much as possible.

2.3.3.2 Reset Function

The reset() function is quite simple. Its main purpose is to scrub the webpage plots of all data. However, the very first step of reset() is to notify the code loop thread that a reset is occurring so that the code loop can respond accordingly. It does this by putting an object in a Python.multiprocessing.Queue object that we call the reset-queue. This Queue object is threadsafe, which means that different threads can safely put and retrieve objects from the reset-queue. Therefore, the code loop thread can
safely get the object that was put in from the Bokeh thread. The object put in the reset-queue does not carry any significant data - its existence is what matters. The reset-queue is stored by the DF in a DF.reset_queue attribute, and the DF is passed to reset() as an argument. We chose to store the queue in the DF object because every section of Python code that is involved in a reset has access to the DF object, and as we will discuss later, the DF object needs access to the reset-queue as well.

Then, reset() completes its main operation by looping through all Plots and their Datasets and clearing each Dataset’s bok-Source. reset() also takes in a special_reset() function, which can be used to perform any operation that needs to accompany the reset. Regardless of what special_reset() does, a user should choose to pass in special_reset() because for some reason the function cannot be called outside of reset(). After special_reset() is called, reset() needs to reload a component created by components.py that we have discussed previously: the information box (info box). Information in the info box might have to change depending on why the reset was triggered, so to be safe we reload the info box at the end of every reset().

To understand what sort of functions qualify to be passed in as special_reset(), let us assume for the moment that special_reset() is a function that sets the DF’s MRN to a new value. If special_reset() is called before reset(), then the DF MRN will change before the Plots are cleared and before the code loop is informed of the reset. The code loop could then potentially fetch data for the new MRN and update the Plots with incorrect data. On the other hand, special_reset() cannot be called after reset() because of the info box. The info box needs to load information for the new MRN, but this can only happen after the MRN change. Therefore, we have a special_reset() function that needs to be called after the Plots are cleared but before the info box is reloaded, so it needs to be called inside of reset().

The special_reset() argument gives us the power to call reset() for a variety of reset operations that have different purposes but share the need to clear all Plot data. We have discussed one use case for special_reset() and therefore for reset() as a whole, which is for changing the MRN - this happens to be the primary reason for why we call reset(). Later, we will outline the occasions where we need to call reset() with a
different special_reset() function.

2.3.3.3 Update Function

update() is responsible for updating all Plots whenever the DF has fetched data. update() takes the following as arguments: the latest pulled data (which comes in two DataFrames, one for the labs data and one for the vitals), the MRN associated with the latest fetched data (which we will call the fetch-MRN), the DF, and all Plots. We decided for simplicity that update() should never call DF.fetch_data() - it is better that the overhead involved with fetching occur in the code loop thread as opposed to clogging up the Bokeh thread. Thus, update() takes the data returned by fetch_data() as an argument. The DF is also passed in so that update() or functions that update() might call will be able to directly access the DF.all_data attribute that contains all fetched data associated with one MRN.

update() performs the following operations. It loops through all Plots. For each Plot, it loops through each of its Datasets. If the Dataset is a CustomDataset, it calls Dataset.custom_update(), which takes care of the full update process for that Dataset, and then moves on to the next Dataset. Otherwise, it checks whether the Dataset has a FuncObj or an identifier. If there is a FuncObj, then FuncObj.func() is run and it returns three lists. The first list is the timestamps that are the x-axis location of every data point. The second list is the values (y-axis locations) of every data point. The third list contains the specimen source of each data point. As we mentioned previously, the specimen source only exists if a data point is from the labevents table, so the third list can be optionally empty. In the case where the Dataset has no FuncObj, update() has to generate these three lists. update() uses Dataset.id_type to determine whether to filter from the labs DataFrame or the vitals DataFrame. Then it filters the appropriate DataFrame by comparing Dataset.identifier to the observation_identifier column of the DataFrame and selecting the matching rows. It removes all rows with missing values and then organizes the data into the three necessary lists.

Once these steps are over, the three lists only contain new data of the value type
corresponding to the current Dataset. However, there are a few more actions that need to occur before the data can be streamed. First, all timestamps are converted to the current timezone, and then they need to be stripped of their timezone information. This operation is necessary because bok-axes that display time need to be passed data points with naive timestamps (no timezone information) or they will automatically convert the timestamps to UTC (Universal Coordinated Time). bok-axes do this because they internally represent timestamps as seconds since the epoch, which easily converts to UTC but can run into issues with more complicated timezones that, for example, change due to Daylight Savings. By providing naive timestamps we can trick the bok-axes into thinking we are providing timestamps in UTC and then the times will be displayed as the correct time in the current timezone. When the timestamp conversions are complete, we make a fourth list that contains string versions of the timestamps. The string timestamps are used by the bok-Hover tool to display the time when a webpage user hovers over a data point.

The last step before streaming the data is to recalculate the current Dataset’s minimum and maximum values, which are used to calculate the y-axis ranges for the Dataset’s bok-Plot. This is done by pulling the minimum and maximum values from the new data lists and calling Dataset.update_minmax(), which takes care of comparing the new data’s min/max values against the preexisting values and updating them if necessary.

We have not yet mentioned the purpose of the fetch-MRN argument; it is used in a check right before update() streams data to a Dataset’s bok-Source. The check verifies that the fetch-MRN is still the MRN being viewed by the webpage user. This check is necessary because the following order of operations is fairly likely:

1. The code loop begins to fetch data associated with fetch-MRN, and the code loop thread idles while waiting for a child process to finish the fetch.

2. The webpage user decides to change the MRN to new-MRN. They trigger this change from the webpage, which in turn triggers bok-Doc to schedule reset() to be called during the next free time slot (tick) of the Bokeh thread.
3. The code loop calls bok-Doc.add_next_tick_callback() so that update() will stream the fetch-MRN data on the next available tick of the Bokeh thread. The code loop has no way of knowing that the Bokeh thread has scheduled a reset, and will only be informed of the reset once reset() is called.

4. reset() is run as scheduled. The MRN is changed to new-MRN, and all plots are cleared.

5. update() runs as scheduled, and adds data from fetch-MRN to plots that are meant to display new-MRN’s data.

This situation cannot be permitted to happen since it could result in false data displayed on the webpage. update() should never stream data from an outdated MRN to the plots. Therefore, every time update() is about to stream data, it checks whether or not the fetch-MRN is still in use. If the fetch-MRN is not in use, update() will terminate immediately.

Once every Dataset has been streamed to and all Plots have been updated with new data, update() will ask every Plot to update its bok-axes accordingly by calling Plot.update_x_axis() and Plot.update_y_axis(). If a Plot has two y-axes or two subplots, update() will call Plot.update_y_axis() twice.

2.3.3.4 Code Loop Operations

Now, we can discuss exactly what the code loop does and how it handles resets. As a reminder, the code loop is initialized by a run_plots() function in stream_plot.py. run_plots() takes a single argument (loop_time) that states the minimum amount of time that one uninterrupted iteration of the loop should run for. We have chosen to set loop_time to eight seconds. When there are no resets, the code loop runs the following steps:

1. Fetch data using the DF object.

2. Schedule update() call on Bokeh thread.
3. If steps 1 and 2 have taken longer than \textit{loop\_time}, then immediately go to step 1; otherwise, wait for the time since initiating step 1 to equal \textit{loop\_time}, then go to step 1.

Step 2 is nearly instantaneous, but steps 1 and 3 can take quite a while and will cause the code loop thread to idle, which gives the Bokeh thread a chance to run \texttt{reset()}. Therefore, we need to be able to detect a reset that is executed during either step 1 or 3. Once the reset is detected, the desired behavior is to ignore all results from the previous loop and immediately reloop so that data can be fetched for the recently-scrubbed Plots. This behavior is important because we want the time between a webpage user’s reset request and the plots showing new data after a reset to be as short of a delay as possible.

To be clear, we do not care about whether a reset has been scheduled during steps 1 or 3. The only scheduling issue that could arise is the one that we heavily dissected in the previous section, which is when a \texttt{update()} call is scheduled after a \texttt{reset()} call is scheduled but before the scheduled \texttt{reset()} actually runs. \texttt{update()} deals with this behavior and will terminate if necessary. The code loop only needs to worry about when \texttt{reset()} is actually run.

\textbf{2.3.3.4.1 Handling Resets During Data Fetch}

For step 1, the code loop simply calls \texttt{DF.fetch\_data()} and waits for a response. Therefore, the reset handling has to occur in \texttt{fetch\_data()}. \texttt{fetch\_data()} performs the following operations:

A. Prepare the fetch query that the database should run.

B. Create a pipe (\texttt{Python\_multiprocessing.Pipe()}) that can communicate the results of the query back to the current process.

C. Send the fetch query and the pipe to a separate process, which we call the \texttt{query\_handler}, via the DF attribute \texttt{query\_queue} (a \texttt{Python\_multiprocessing.Queue} object).
D. Wait for the `query_handler` to complete the fetch query and return the resulting data via the pipe.

E. Append the results to the DF. `all_data` attribute and then return the results.

Operation D is the only operation that runs for long enough that the Bokeh thread would have a chance to run `reset()`. We could just make D block while waiting for the `query_handler` and then no resets could be run. However, this also prevents any other Bokeh UI interactions from running, which as we have previously mentioned is not desirable. Therefore, we have to ensure that D does not block the process, and since this allows resets to be run over the duration of D, we have to be prepared to handle resets.

To prevent D from blocking, we run a mini-loop that performs three actions. First, the mini-loop checks whether the `query_handler` has put anything in the pipe; if so, we get the results, break out of the mini-loop and proceed to operation E. Otherwise, the mini-loop then checks whether the reset queue (stored by the DF in the `reset_queue` attribute) has an object in it; if it does, then a reset has occurred. If a reset has not occurred, then the mini-loop sleeps for 0.1 seconds before looping. The sleep is important because it lets the code loop thread be idle for a gap of time during which the Bokeh thread can run any webpage interaction callbacks, including any call to schedule `reset()`.

If the mini-loop detects a reset, then the data that the mini-loop is waiting for from the `query_handler` process is now useless. There is no reason to wait for, save and return the fetched data since all Plots are going to be scrubbed and the fetched data will never be streamed to the Plots. However, we cannot just terminate `fetch_data()` as soon as we detect a reset. If we do immediately terminate, then the `query_handler` process will still complete the fetch and deposit the results in the pipe at some unknown time in the future. Then, a future call to `fetch_data()` will see those results and assume they are correct, even though they are stale results. Canceling the fetch would be the optimal solution, but we could not find a way to do that without completely corrupting the psycopg2 connection object that handles
communications with the database. Simply waiting for `query_handler` to return is not an option either, since the purpose of detecting a reset is to quit step 1 of the code loop and start a new data fetch as quickly as possible.

Therefore, before `fetch_data()` can terminate, it spins up a new process which has the sole purpose of waiting for the stale results to appear in the pipe so that the results can be removed. Since the new process will not interfere with the main process containing the code loop and Bokeh threads, `fetch_data()` is now free to terminate, and it does so by throwing a FetchException. The code loop line that calls `DF.fetch_data()` is wrapped in a try-except block that catches the FetchException thrown in case of a reset. As soon as the code loop catches the exception, it starts a new iteration of the code loop, which in turn will begin a new fetch to fill the scrubbed Plots with new data.

However, from here on we might encounter another issue. It is possible that when a reset is triggered and `fetch_data()` terminates without waiting for its fetch to finish, the abandoned fetch will take a very long time to return any data, which results in the `query_handler` process being blocked until the abandoned fetch returns. If the reset is supposed to cause the code loop to restart and execute a new fetch, the new fetch has to wait to run until the `query_handler` finishes running the abandoned fetch. This could result in a significant delay before the cleared plots are filled with data again. To avoid this, the `query_handler` process does not run the database fetches. Instead, it spawns a child process for every fetch that it needs to run, and the child process will send the fetch results to the pipe. This way, if `query_handler` needs to start a new fetch while an abandoned fetch is still running, it can simply start the new fetch in a different child process.

The downside to spawning a child process for every fetch is that a user could trigger multiple resets in a row, which could cause more than two child fetch processes to be running at once. While, realistically, there will rarely (if ever) be a high enough number of child processes running to slow down all other processes, it is a potential issue and it would be much cleaner to kill the child processes instead of letting them run to the finish. As we mentioned before, though, we could not find a way to kill
the child processes without corrupting the DF. The second downside is that psycopg2 connection objects cannot be shared across different processes. Therefore, whenever query_handler has to spawn more than one child process, it needs to create new connection objects for each additional child process. Once again, this is an undesirable setup that could cause load problems if there are too many child processes running.

2.3.3.4.2 Handling Resets During Code Loop Wait

In contrast to the previous section, resets that occur during step 3 of the code loop are very easily handled. The code loop runs a mini-loop similar to the one used by DF.fetch_data(). The mini-loop first checks DF.reset_queue to see if a reset has occurred. If so, the mini-loop exits and the code loop enters its next iteration. If there was no reset, then the mini-loop checks whether enough time has passed before entering the next loop (which is set by loop_time). If enough time has passed, the mini-loop breaks; otherwise, the code loop thread sleeps for a short amount of time before the mini-loop restarts. Once again, the sleep is important because it allows the Bokeh thread to run user interaction callbacks while the code loop thread is idling.

2.3.3.5 Exiting the Code Loop

The code loop is set to run forever, but there is no need for it to continue running if the bok-Doc that the code loop is controlling has been closed. Recall that whenever a user accesses the URL for the Bokeh server, a new bok-Doc is created specifically for that user and a new code loop thread is spun up to run that bok-Doc from the Python side. However, if the user closes the webpage they are looking at, then the bok-Doc is no longer in use and will never be again. The bok-Doc will realize that it is not being used within a minute of its webpage being closed; then, bok-Doc will flip an internal flag (bok-Doc.session_context.destroyed) from False to True. The code loop will occasionally check to see whether this flag has changed; when the code loop sees that the flag is True, it will exit. In turn, the thread running the code loop will also terminate.
2.3.4 Customizability of Implementation

When writing the Python code, it was very important to us that future contributors to the project should be able to easily modify certain sections without needing to overhaul all of the code. We wanted to make it very simple to change aspects of the code that we thought many users would want to experiment with, and we strove to make the code very modular so that users could make serious edits to certain modules without needing to touch others.

We expect many future users will want to have the following options:

1. Change the number of Plots.
2. Change the type of each Plot (single-y-axis, double-y-axis, subplots).
3. Change the Datasets associated with each Plot.
4. Add complex Datasets that require heavy calculation on data before plotting.
5. Add new components to interact with the plots.
6. Change the layout of the webpage.
7. Add methods to DataFetcher to pull other sets of data.

Options 1, 2 and 3 are very easily done - stream_plot.py sets up Plots and Datasets very quickly with the Plot() and Plot.add_dataset() functions. Changing how the Plots are initialized and what Datasets are used is as simple as changing the arguments to those functions. Option 4 requires a little more work, but the FuncObj was created for this purpose. A user will have to write their own FuncObj in stream_plot.py to get the desired Dataset behavior, but the FuncObj has an extreme amount of leeway in what it is allowed to do and so the user is afforded a lot of flexibility. If the FuncObj is not enough, then users can create their own CustomDataset. All CustomDatasets are declared in a separate file called custom_datasets.py, and since they inherit from Dataset the user will be able to preserve most of the important Dataset behaviors.
To perform options 5 and 6, users can edit the *components.py* file. Adding a component requires Bokeh knowledge since all components need to be added to the bok-Doc and therefore need to be Bokeh objects. However, a user only has to write a function to generate the component and then add a call to that function in *components.setup()* to add the new component to the layout. Similarly, the layout of the webpage is almost entirely designed by *components.setup()* and the user can modify the layout by modifying the function.

If a user designs a component that needs to pull a certain set of data from the database, adding a method to the DataFetcher to pull that (as in option 7) is fairly simple. As previously discussed, there are two DF methods aside from DF.*fetch_data()* that pull personal patient information and bed information respectively. Any future methods will likely be similar to these existing DF methods.

We tried to make the different classes and files very self-contained. Datasets are only created by Plots, and Plots are only created in *stream_plot.py*; the DataFetcher class is fairly complex and many functions and callbacks interact with it, but it acts as the sole access point to the database and therefore any optimizations made to increase database communication efficiency need to only be applied to the DataFetcher. *components.py* collects all non-plot Bokeh objects in one place. *stream_plot.py* is the main file for the server to run, but any file can be written to initialize Plots and Datasets and that new file could also be the Bokeh server target. The only section of code that would be hard to work with is the code loop, since its functionality is confusing at times. But even then, the code loop only interacts with three functions: DF.*fetch_data()*,*update()* and *reset()*.

2.3.5 Python Code - Summary

**Figs. 2-3** and **2-4** provide a comprehensive overview of the section of the visualization system that is controlled by our Python code. The structure of the code allows for a highly versatile visualization system and provides room for expansion and improvement by future contributors. The DataFetcher may currently only use the data stream, but can be improved upon to pull data from any desired data source. The
Dataset object provides tools for customizing the processing of the data, and the Bokeh architecture creates an interactive interface for the plotted data. Therefore, our Python code facilitates the visualization system as a tool to utilize ICU data efficiently and act as clinician decision support in the future.

2.4 Serving to the Web: Flask

When discussing the Python code, we mentioned that the Bokeh server can only serve to a localhost URL. However, we want the plot webpage to be accessible outside of the machine running the server. Specifically, we want an application that merges with the Bokeh server to serve the webpage to a general URL that can be accessed by anyone with the correct security permissions. An example of an such an application is Flask, a Python web framework that Bokeh is capable of integrating with [6]. While we did not have the time to write a full Flask application that serves the webpage as we would like, we created a test app to prove that the Flask app could accomplish what we wanted it to.

Bokeh has a function called Bokeh.embed.autoload_server() which takes in a URL as an argument and returns a script tag that can be put in any HTML file. The URL input is the localhost URL that the Bokeh server is currently serving to. The returned script tag contains the webpage representation of the bok-Doc that the Bokeh server repeatedly generates. Our test Flask app simply generates this script tag, puts it in an empty HTML file, and serves the HTML to a new-URL of our choice. In order for autoload_server() to work correctly, the Bokeh server needs to be informed that a Flask app is going to embed the Bokeh webpage on another webpage at new-URL. To do this, we call the bok-server command (which is responsible for spinning up a Bokeh server) with the flag "-allow-websocket-origin=[new-URL]."

Our test app is a proof of concept - it does not handle all the security considerations that we would need to accommodate for if the visualization system were to be formally deployed. It is critical that only users with the correct permissions be able to access the webpage because the webpage displays patient data that has not been de-identified.
and therefore is not suitable for public consumption. However, the Flask framework is capable of adding security checks to the new-URL. The implementation of those checks was outside of the scope of this iteration of the project and so will have to be completed in the future.
Chapter 3

Visualization Interface

We have examined the visualization system’s implementation in detail. Now, we will present the current version of our webpage visualization, and we will explain the design decisions behind our webpage. We have split up our description of the visualization into three sections - the overall layout, the plots and the additional components.

3.1 Webpage Layout

Our layout is depicted in Fig. 3-1 and is designed as follows. The upper section of the page is reserved for three components that act as special tools for manipulating the information displayed by the plots. The names of these components, from left to right on the webpage, are the Bed Selection Map, the MRN Input, and the Retrospective Plotting Input. The two input components are placed inside a tabbed section - only one of them is visible at a time, which saves space on the webpage. Tabbed sections are created via a bok-Tab class, where each individual tab takes a title and a bok-Layout object. In the future, if users wanted to add more components to the page, they could avoid clutter by adding a new tab to accompany the input components.

The rest of the page is split into two halves. The left half contains the plots. Once again, we make use of bok-Tabs so that each tab contains the layout corresponding...
Figure 3-1: The current design of our webpage.

to one Plot object. There are four plots that can be viewed individually; Biochemistry, Respiratory, Vitals and Custom. Each plot is accompanied by a slider object. Underneath the slider, there is a fifth plot that we call the Score plot. The Score plot is outside of the tabs and is meant to be viewed in conjunction with any of the tabbed plots. The right half of the lower page is reserved for textual information that could be useful when viewing and interpreting the data on the plots - this information box is a component that we have mentioned previously.
3.2 Plots

As we mentioned before, there are five plots that we have chosen to display on the webpage. Each of these plots embodies various use cases for the plots and the visualization system as a whole. The plots also demonstrate the wide range of plot and data interactions that a webpage user can take advantage of.

3.2.1 General Plot Characteristics

Fig. 3-2 shows a general example of a plot produced by our Plot object. Since this is a default Plot, it does not have any subplots and only uses one y-axis. The x-axis is a time axis and labels its ticks with their corresponding times. The y-axis, as we have discussed previously, has no units because it is likely that the units of the plotted datasets are all different. There is a dashed blue line (span) running down the right end of the plot that represents the current time. The span and all axes are updated every time that the plot is updated by the code loop, which is approximately once every eight seconds.

This particular plot has four datasets. Each dataset’s name and representative glyph is listed in the legend on the top left hand corner of the plot in Fig. 3-2. The data for these datasets originates from the labevents table, so the glyphs for each dataset are circles only with no connecting line. Information for every data point can be seen by hovering over the desired point, as shown in Fig. 3-3. The legend allows users to hide datasets they do not want to see by clicking on the datasets’ names in the legend. Clicking a hidden dataset’s name again will restore the dataset’s visibility.

The toolbar on the upper right side of the plot in Fig. 3-2 provides tools for a user to interact with the plot. A closeup of the toolbar can be viewed in Fig. 3-4. The lines between certain tools indicate tool groups that are mutually exclusive. For example, since the first two tools use the left mouse button, they cannot be selected...
Figure 3-2: An example plot created by our Plot class.

at the same time. The tools are, from left to right:

- Box Zoom: click and drag with the left mouse button to select a box region on the plot, and the plot will resize its axes to zoom-in to that box region.

- Drag: click and drag with the left mouse button to move the plot view around.

- Wheel X-Pan: scroll to pan the plot view along the x-axis.

- Wheel Y-Zoom: scroll to zoom-in and zoom-out with respect to the y-axis.
• Reset: if the user has modified the plot view by using a tool or set of tools, clicking Reset will return the plot view to the standard view.

• Save: click to save the plot in a PNG file.

Bokeh provides many more tools than what we have selected our toolbar, but we chose the tools that we thought would be the most useful. The Wheel Y-Zoom tool can be switched to zoom only along the x-axis, or along both axes equally. However, we thought that users would be more likely to pan across the x-axis than zoom in or out along it, so we chose to disallow all x-axis zooming.

We have discussed the Reset tool before in relation to the standard view of the plot. The standard view is the plot view that is shown when the x and y axes’ ranges are assigned by the code. Whenever the user interacts with the plot via the provided tools, these ranges change, and so the user has exited standard view; any code updates to the ranges will be disabled. Clicking the Reset tool returns the plot to standard view.

![Figure 3-3: Example of using the hover tool on a plot. The hover box lists the exact time and value associated with the data point. Since this data point is pulled from the labevents table, the hover box also includes the specimen source of the value.](image)

![Figure 3-4: Closeup of plot toolbar from Fig. 3-1.](image)
The Slider object below each plot controls the number of days’ worth of data displayed by the plot. The slider can be changed to show anywhere between one and ten days worth of data at a time. An example of the slider change is shown in Fig. 3-5. Moving the slider does not cause the plot to exit the standard view; rather, the slider’s value determines the x range of the standard view. If a user wants to view less than a day’s worth of data, they can do that by using the Box Zoom tool; however, there is no way to look at more than ten days of data. The ten day limit was an arbitrary choice and can be easily increased.

One note is that all of the plots on the webpage share the same bok-axis object for their x-axes. If a user pans along the x-axis of one plot to twenty days into the past and switches to look at another plot, they will find that the second plot has automatically adjusted to twenty days in the past. This behavior is useful if a user wants to look at the same time range across multiple plots; they would not repeat their pan action for every plot. The shared x-axis is not automatic Plot object behavior. We modify the Plots and their underlying bok-Plot objects when creating the layout in components.setup() and set their x-axes equal to each other.

Figure 3-5: Slider at two different positions - each position scales the x-axis to a different number of days. Top slider is set at three days, bottom slider is set at seven.
3.2.2 Biochemistry Plot

The Biochemistry plot shown in Fig. 3-6 is an example of a Plot object with two subplots. The top subplot plots creatinine and bilirubin, which tend to numerically range from 1 to 8 and 0.5 to 2 respectively. The bottom subplot contains glucose and sodium, which range from about 80 to 130 and 135 to 145 respectively. These ranges show that plotting all four datasets on the same axis would not be ideal since their

Figure 3-6: Biochemistry plot, created via a Plot object with two subplots. The top subplot displays creatinine and bilirubin datasets, while the bottom subplot shows glucose and sodium. All the datasets for this plot pull from the labevents table.
ranges are so different, so we chose to plot them on two different subplots. All four of these datasets are pulled from the `labevents` table by using identifiers.

We chose to include this plot for two reasons. The first was that the datasets included in this plot are commonly used by clinicians to quickly evaluate how a patient is doing. Creatinine levels provide insight into a patient’s kidney functions, and bilirubin levels can be indicators of liver and gallbladder health. Glucose and sodium blood levels are important for overall body function. The second reason was that the Biochemistry plot is the simplest type of plot that we could have created. The code for creating this plot is five lines long; we initialize a `Plot` object and then call `Plot.add_dataset()` four times. Adding or removing datasets is as simple as modifying the `add_dataset()` calls. Therefore, this plot is an example of how easy it is to create Plots and add modifications to the visualization system.

### 3.2.3 Respiratory Plot

The Respiratory plot in Fig. 3-7 is an example of a use case for two different code features. The first feature is that the plot has double y-axes. The left y-axis plots pO2 (partial pressure of oxygen in the blood), pCO2 (partial pressure of carbon dioxide in blood) and FiO2 (fraction of inspired oxygen) - all three of these value types are pulled from the `labevents` table by using identifiers. The right y-axis plots pO2/FiO2, which is the ratio of oxygen in the bloodstream to the fraction of oxygen that was inhaled. Note that the glyph for pO2/FiO2 is a triangle; this is to distinguish it as a dataset plotted on the right axis so users can know to not use the left y-axis. Each y-axis also has its own legend next to it to help users see which datasets belong to which axis.

The second code feature is that, as we have discussed before, the pO2/FiO2 ratio does not exist as a direct value in the database; therefore, we must calculate it ourselves. The pO2/FiO2 dataset uses a `FuncObj` instead of a `identifier`. The `FuncObj` pulls the pO2 and FiO2 data from the database and computes the ratio for every FiO2 value with the closest pO2 value taken within the twenty-four hours before the FiO2 value. Defining the `FuncObj` is not trivial, but the `FuncObj` demonstrates that it is
possible to perform user-defined calculations for a dataset.

We could have chosen to use two subplots for the Respiratory plot instead of two y-axes; however, FiO2 values are less common in the database when compared to pO2 and pCO2 values. Therefore, calculated values of the pO2/FiO2 ratio are rare, and reserving one subplot for such a sparse dataset seemed to be an underutilization of screen space.

**Figure 3-7:** Respiratory plot - created via a Plot object with double y-axes. The left y-axis displays pO2, pCO2 and FiO2. The right y-axis is used for the pO2/FiO2 ratio. All the datasets for this plot pull from the labevents table.
3.2.4 Vitals Plot

The Vitals plot displayed in Fig. 3-8 contains datasets pulled from the vitalevents table. vitalevents data tends to be taken frequently at regular intervals, so each dataset is represented by both a circle glyph and a line connecting every circle to

![Vitals Plot](image)

**Figure 3-8**: Vitals plot - a normal single y-axis Plot. The datasets on the plot are; heart rate, O2 saturation, respiratory rate, body temperature and mean blood pressure. The red-orange band in the background accompanies the mean blood pressure line and represents the systolic and diastolic blood pressure measured with each mean blood pressure value. All the datasets for this plot pull from the vitalevents table.
make the data easier to view. The underlying Plot object has the default settings of no subplots and one y-axis.

The five datasets plotted on the Vitals plot are: heart rate, O2 saturation, respiratory rate, body temperature, and blood pressure. These datasets can show heavy correlation. A lower O2 saturation in a patient could prompt a higher heart rate and/or respiratory rate in an attempt to bring more oxygen into the patient’s bloodstream. Similarly, high temperatures can lead to high heart rates, and blood pressure and heart rate tend to rise and fall together.

The blood pressure dataset is in fact created by a BPDataset object, which inherits from CustomDataset. The blood pressure dataset needed to be custom because it is responsible for the thick band that runs across the background of the vitals plot in Fig. 3-8. The band accompanies the mean blood pressure line, and the upper and lower values of the band correspond to the systolic and diastolic blood pressure values taken at the same time a mean blood pressure value was recorded. Standard Datasets are not capable of handling this band, since normal Datasets can only store one value per time whereas the blood pressure data has three values per time (systolic, mean, and diastolic blood pressure). Therefore, we wrote BPDataset to update the band in conjunction with the mean blood pressure line. The BPDataset is an example of how the visualization tool can be used to customize how data values are plotted.

3.2.5 Custom Plot

The Custom plot is a heavily-modified Plot object that allows the webpage user to choose what datasets they would like to see plotted (as shown in Fig. 3-9). The plot has one y-axis with four datasets that are initialized with no name and no identifier. A user can select up to four value types from the checkboxes at the bottom of Fig. 3-9, below the Custom plot. Once the user clicks the "Plot" button, the plot datasets will be updated with the names and identifiers corresponding to the selected checkboxes. Currently, the checkboxes only correspond to identifiers for the labevents table.

This plot exists as a way for webpage users to see certain datasets that might not be available on the other predefined plots. The other plots’ contents are selected
Figure 3-9: Custom plot - a normal single y-axis Plot with four unspecified datasets. Users can select which four value types they would like to plot by using the checkboxes below the plot. The Custom plot in the image currently has three out of four datasets specified as hematocrit, hemoglobin and platelets.
carefully so as to be used often by a wide range of users. However, sometimes a user will want to check a particularly obscure dataset, or will want to plot a certain set of datasets together in a way that we would not be able to predict. The Custom plot offers the freedom for a user to do so, but at the loss of fixed-plot features such as multiple axes or subplots.

The Custom plot is also an example of how Plot objects can be tinkered with to produce very powerful and atypical plots. Out of all the plots on the webpage, the Custom plot is the most complex, and encoding this plot required a strong knowledge of Bokeh. The fact that it was possible to make this plot, though, speaks to the immense flexibility that the visualization system can provide for plot creation.

3.2.6 Score Plot

Fig. 3-10 contains an image of the Score plot, which is used to display scores. A score is a value calculated from multiple other data values that is a predictor for a patient’s overall state. The y_logit score that is currently plotted on the Score plot is a simple linear calculation (using eleven patient values) that is passed through a sigmoid function. The y_logit score is intended to be an indicator of mortality, with a score of zero representing a 0% likelihood of death and one representing 100% likelihood of death. The full formula of the y_logit score is located in an appendix. The Dataset object backing the y_logit score is a standard Dataset that uses a FuncObj to calculate the score. This FuncObj pulls data from both labevents and vitalevents.

The Score plot is unique in that it is the only plot on the webpage that is not contained within the plot tabs. It can always be viewed in conjunction with any of the tabbed plots, as is visible in Fig. 3-10. We chose to place the Score plot in this manner because if the Score plot indicates that a patient’s condition is worsening, then the user will want to specifically find out which other data values correlate with the bad score. Therefore, the user will likely want to be able to look at the score plot and all other plots at the same time.
Figure 3-10: Score plot (the lower plot in the image) - a normal single y-axis Plot that has been customized to show calculated scores that can indicate a patient’s overall condition. The score plot’s x-axis is always in sync with the plots above it. The datasets for this plot pull from both the `labevents` and `vitalevents` table.

We have customized the Score plot in three ways. First, it is shorter than all other plots. We chose to do this so that the tabbed plots and the score plot could fit on the browser screen at the same time. We also restricted the plot’s single y-axis to a fixed range of [0,1]. The y_logit score cannot be outside of the [0,1] range, so restricting the axis was reasonable. Lastly, the Score plot has no toolbar because it does not need one; its y-axis is fixed and its x-axis is linked to all other plots, so using tools on the tabbed plots will result in x-axis view changes on the Score plot.

3.3 Components

The four components on the webpage are the information box, the bed selection map, the MRN input and the retrospective plotting input. All of these components are created in `components.py` and are Bokeh objects that are either callers or targets of callbacks executed by the Bokeh thread. Of the components listed above, the latter three call the `reset()` function when interacted with.
Figure 3-11: Information box (info box) component. Includes basic information about the current patient being viewed on the webpage. Patient-identifying information has been blurred.

3.3.1 Information Box

The information box (info box, Fig. 3-11) holds two types of information. The first type consists of basic patient information: MRN, sex, age, date of birth (DOB) and admit date. The MRN is used as the patient identifier. The sex, age/DOB and admit date information can help to put into context a patient’s data. The second type of information held in the info box is admission notes - these are notes that include justification for why a person was admitted to the hospital and they may provide initial instructions for the patient’s care. These notes are integral to understanding a patient’s condition. The data stream does not currently send over admission notes, so we have left a placeholder for the notes on the webpage.

The info box does not trigger any callbacks, but it is updated every time reset() is called. This is because the special_reset() function called by reset() could change the information that needs to be displayed by the info box. For example, if reset() switched the webpage data to a different MRN, then the info box would have to load information for that new MRN. The info box loads its new information by calling DF.get_basic_info().
3.3.2 Bed Selection Map

The bed selection map (bed map, Fig. 3-12) is a visual representation of the beds in the ICU wards in Beth Israel. Since we receive data from the CC7C and CC7D wards, the bed map creates buttons for each bed in those wards and lists each bed’s number. When the button for a bed is clicked, the MRN displayed on the webpage changes to that of the patient in the clicked bed. If a bed in the ICU wards currently has no patient, then the button corresponding to that bed will be greyed out and cannot be clicked on. Otherwise, the button will be active, and will list the bed patient’s sex and age information if available.

The bed map is actually a bok-Plot object with selectable square glyphs that represent beds. Bokeh can trigger callbacks when glyphs are selected, so selecting a bed square will call reset() with a special_reset() function that changes the DF’s MRN to the MRN of the patient in the selected bed. The map is generated from a nested list containing the names of every bed in the relevant wards. Therefore, adding and removing wards from the bed map can be done by modifying the bed names list.

3.3.3 MRN Input

The MRN input component, shown in Fig. 3-13, takes in a new MRN and switches the webpage’s MRN to displaying the new MRN. The component does this by calling
reset() in the same manner as the bed map. Since it is very easy for a user to accidentally submit an invalid MRN, the MRN input component also checks any entered MRNs to make sure they are valid.

### 3.3.4 Retrospective Plotting Input

The retrospective plotting input (retro input, Fig. 3-14) takes in a past date and time and resets the webpage so that all plots and components act as if the webpage is in the past. If the current date and time is 2017/11/1 at 10:50pm and a user would like to view the webpage as it would have looked on 2016/11/1 at 1:00pm, then the user would enter the date and time that they want to switch to and then click the “Start” button. From then on, the website will progress as if it were live; at 10:53pm in real time, the webpage would update to show data that became available at 1:03pm in the past. To end retrospective plotting, the user can click the “End” button and the webpage will return to plotting in real time.

The retro input affects all the plots, the info box and the bed map. The span on the plots that indicates current time is now moved back to the date and time specified by the retro input. The plots’ shared x-axis range is also shifted back to the retro input time. The underlying DF will only pull data that would have been available at

![Image of MRN input component](image.png)

**Figure 3-13:** MRN input component. Takes a new MRN and changes the MRN of the webpage to the new MRN.
Figure 3-14: Retrospective plotting input (retro input) component. Allows the webpage to act as if it were in the past at a specified retro time. The past time and therefore the plots will only display that past data. The info box will display the age that the patient would have been in the past. The bed map displays the information of the patients who were in the ward beds in the past; clicking on a bed will switch the webpage MRN to the patient in that bed in the past.

While the retro input affects many aspects of the webpage, the callbacks it triggers are fairly straightforward. We created a TimeKeeper object that has many methods that return the current time in various formats. Whenever our code has to obtain the current time, it must do so by calling a TimeKeeper method as opposed to using the default Python time libraries. When the retro input is started, it switches the TimeKeeper object to retro mode, and all TimeKeeper methods will then return the retrospective time that corresponds to the real current time. The retro input also triggers a call to reset() that clears the plots. The special reset() argument sent to reset() does not change the MRN though; instead, it reloads the bed map so that the map can adjust to the retrospective time.

We created the retro input primarily because the data stream was not sending over vitals data, so we had to find a way to view the vitals data we had from 2016. By using the retro input component, a user can view all the vitals data taken in 2016 along with any other past data that might be of interest to them.
Chapter 4

Future Work

While we have attempted to create a highly-flexible and usable visualization system, we acknowledge that much more work has to be done before it can be deployed for use, either in a clinical or research setting. We ran into many issues with our current implementation of the system that we were not able to resolve in time, but that need to be resolved before the system can be formally used. We also asked user testers to try the visualization interface and provide feedback; this feedback, while generally positive, gave us many suggestions for future improvements. Lastly, our current system needs to be optimized and expanded in many ways before it is ready for clinical use.

4.1 Unresolved Issues

There were many issues that we encountered while creating the visualization system; examples include the vitalevents pull delay and the complexity surrounding the \texttt{reset()} function. We were able to resolve the majority of these issues, but there were a few problems that we either did not have the time to resolve or that we could not solve. We will discuss each of the unresolved issues individually.

\textbf{Abandoned Fetches:} In previous sections, we have defined an abandoned fetch as a fetch process spawned by a DataFetcher (DF) that will return a result that we
no longer need. The most common reason for abandoning a fetch is in the case of an MRN change; the abandoned fetch will be executing a database query for an outdated MRN. To abandon a fetch, we let its corresponding fetch process run to completion but we do not wait for its results. We cannot kill the fetch process while its query is executing because doing so would corrupt the connection between the DF and the database. However, it would be optimal to halt the execution of the query altogether so that we can kill the fetch process safely and the database does not have to continue to run an unnecessary query. As of now, we have not found a way to halt the query execution, but finding a way to do so would be optimal.

**Timezones and the bok-axis:** The bok-axis objects representing the x-axes for all Plots display times in terms of Coordinated Universal Time (UTC) exclusively. While this is a logical choice for the Bokeh architecture because UTC is a simple timezone based only off of the seconds since the epoch, Beth Israel and LCP-MIT are both located in the Eastern Standard Time/Eastern Daylight Time (EST/EDT) timezone. Therefore, any clinician viewing the data in Beth Israel would have to mentally convert all times into EST/EDT, which is unideal. Our workaround for this has been to strip our data timestamps of their timezones so that the bok-axis representing the x-axis is tricked into interpreting the data points as if they were in UTC.

The issue with this workaround arises when EDT switches back to EST near the end of the year and the hour between 1:00am and 2:00am is repeated. If a data point is plotted on a webpage plot between 1:00am and 2:00am on the day of the timezone switch, then users will not be able to tell which hour the data point actually belongs to. We initially attempted to tackle this issue by including a note in the hover box for every data point that would state the amount of time since the time the data was taken. Thus, if the current time were 3:00am EST and a data point was placed at 1:30am EST/EDT, then the hover box would tell the user whether or not the data was taken 1.5 or 2.5 hours ago. Unfortunately, this solution was not workable; the hover box does not allow us to display a value that changes over time. We have
decided to leave the issue alone for now since the issue will only cause problems for users for one hour every year.

**Bed Map Refresh:** The information in the bed map component is only ever loaded or reloaded in two situations; when the bok-Doc containing the component is initialized, and when the retro input component loads the bed map for a time in the past. Therefore, if a user is looking at the webpage in real time and keeps the page open for a while, it is likely that a patient has left an ICU bed or a new patient now occupies a previously-empty bed, but the user will not be made aware of that information because the bed map has not loaded that new patient information. The architecture exists to reload the bed map, but we have not decided how frequently we want the map to update. For now, the bed map will only refresh when the whole webpage is refreshed, which triggers the creation of a new bok-Doc.

**Band Visibility Change:** All plot legends have a feature where the visibility of the datasets listed in a legend can be toggled on and off. However, the band created by the BPDataset will not change its visibility whenever its corresponding mean blood pressure line changes visibility. We wanted to be able to hide the band as well, so we attempted to add an event listener to every bok-Doc that would listen for visibility changes in the mean blood pressure line and trigger a callback to change the visibility of the BP band to match. However, for undiagnosed reasons, the callback would take upwards of seven or eight seconds to trigger. We thought the callback delay was unacceptably long, so for now we have deactivated the event listener so the band never changes visibility.

**Custom Plot Legend Errors:** The versatility of the Custom plot comes from our ability to change the plot’s datasets freely. However, changing a dataset requires that its name in the plot legend be updated as well. Due to the underlying Bokeh architecture of the legend, changing a name in the legend can cause the legend to stop accepting name updates or to disallow visibility changes. In the worst case, the
Legend errors can stop the whole Custom plot from functioning properly. The Bokeh developers are aware of these legend errors and plan to fix them for the next release of the Bokeh library, so we have chosen to allow the legend errors for now.

**Exceptions thrown by Bokeh thread:** We are capable of catching and handling any exceptions that are thrown by the code loop thread or any callbacks that the Python code adds to the Bokeh thread. However, we occasionally see exceptions thrown by the callback handler of the Bokeh thread. These exceptions do not stop the overall webpage functionality but can prevent the `update()` callbacks that are triggered by the code loop thread from running. We have no way of knowing whether a triggered `update()` has run or not, so if a specific `update()` callback has not run, we permanently lose the plot data that the callback should have streamed to the plots. To fix this issue, we might need to track `update()` callbacks and retrigger a callback if we cannot tell that it has finished. Unfortunately, we did not have time to address the issue.

### 4.2 User-Suggested Improvements

In order to gauge how usable our visualization system was, we asked users to test the system and see what they liked and didn’t like about the user interface. We attempted to get feedback from a variety of potential users. Overall, the feedback indicated that the visualization system had the power to be very helpful, but the system would require significant additions or modifications before formal deployment.

#### 4.2.1 Tester Demographics

We asked seven different users to test our system. These users’ job titles were as follows:

- Nurse Specialist (1)
- Postdoctoral Associate/Fellow (2)
Aside from the nurse specialist, who works at Beth Israel, these users work at or with LCP-MIT. All of the users have experience with clinical work and/or medical data.

### 4.2.2 Positive Comments

Almost all of the user testers were pleased with the simplicity of the interface. They appreciated the cleanness and colors of the plots, and one user mentioned that they enjoyed the lack of unnecessary clutter on the webpage. Users liked the various interactive elements that were provided; four of the users specifically stated that they highly approved of the bed map, which "made the data feel very tangible".

The users appreciated the contents of the plots as well. One user commented that they found "the ability to visualize patient trajectories (e.g. lab values) over a specified time range" to be very useful. Three users mentioned that the Custom plot gave them a high amount of flexibility that they enjoyed, and two users said that the ability to view the Score plot in conjunction with the other plots was advantageous.

Three of the users specifically noted that the visualization interface would be useful for their daily work. The nurse specialist said that the interface would help to easily identify short- and long-term trends in a current patient. By contrast, one of the research scientists said that they would primarily use the interface’s retrospective plotting component to view patient data as an alternative to having to query the data for themselves. The postdoctoral associate suggested that they would use the visualization system’s ability to add new score algorithms to the Score plot for testing said algorithms. These three users’ feedback demonstrated to us that there were more use cases for the visualization system than the clinical setting we had designed the system for; these alternative use cases should be taken into account when augmenting the visualization system in the future.
4.2.3 Desired Improvements

There were two categories of suggestions for improvement. The first category included ideas for what types of data should be included and how the data could be better presented. The second category consisted of other improvements for the interface itself.

4.2.3.1 Data Improvements

While users generally liked the data that they could view, there were requests for other types of data, such as treatment information. We hope that the admission notes that we plan to add in the future will contain some treatment information, but it is good to know that users do want more auxiliary information pertaining to the patient (e.g. ventilation levels). One user suggested that the default tabbed plots could be organized by organ function; for example, the Biochemistry plot consists of values that do not relate to the same organ, and splitting those values into separate plots titled "kidney function" and "liver function" could prove to be useful. Another user asked that the plots be able to extract the data they are displaying and save it in a CSV (comma-separated values) file for further analysis, which is a use case for the data that we had thought of previously.

Users also asked for improvements to the webpage elements supporting the data. Three different users mentioned that someone using the webpage many not know what each dataset represents. For example, one of the dataset options provided by the Custom plot is called "PTT". Certain medical professionals may realize that PTT stands for partial thromboplastin time and understand what PTT data indicates with regards to a patient, but it is very likely that other users will not reach the same conclusion easily. The users suggested displaying a description for every dataset, possibly when hovering over each dataset’s name. Two users noted that the lines connecting vitals datasets in the Vitals plot made the data much more readable and requested that all other data be connected by lines as well. One user asked for improvements to the y-axis. They thought the option to fix the y-axis ranges to a
normal range would be useful, as opposed to letting the y-axis automatically scale to show the least and greatest values. Interestingly, they also requested that the y-axis include units. We specifically decided not to include units because we often plot datasets with very different units along the same axis; therefore, displaying units on the axis will not be trivial. However, it may be sufficient to include the units of a data value in its hover box.

Lastly, one user mentioned that long-term users of the visualization system would probably want to personally set the datasets for their fixed plots. While our current fixed Biochemistry plot displays values that are generally considered useful, it is possible that a specific user would prefer to, for example, permanently switch out the bilirubin dataset for the lactate dataset (another highly relevant biological value). There are a few ways that we could provide users with this form of customizability. One option is to simply make every plot a Custom plot, but that would remove a user’s ability to open the webpage and quickly glance at familiar plots to get an idea of a patient’s prognosis. Fixed plot configurations should be allowed to vary from user to user, but we want each user to be able to rely on their fixed plots. The restrictions on the Bokeh architecture means that this customizability can likely only be provided through modifications to the server - we will discuss this in detail in a future section.

4.2.3.2 Interface Improvements

The main interface suggestion was that users wanted a way to distinguish a blank plot that was loading data for the first time from a blank plot that simply had no data. Currently, the only way to differentiate between the two is to blindly wait until data appears, which of course is not a useful way because plots with no data will never show any data. To implement this, we would have to show a loading message or a loading symbol whenever update() is running either for the first time or right after reset() is called.

We also received many comments on how the user interface could be improved across different browsers and for tablets. We developed the webpage using Firefox 54.0 on a desktop monitor running Ubuntu 16.04, but we did see issues in the webpage
layout and in certain user interactions when using, for example, Chrome - these are issues that we would want to diagnose and remedy in the future. One user noted that the webpage layout is non-responsive, meaning that when the browser window is resized, the layout will not adjust to the resize. Having a responsive layout is key for using the visualization system on screens of various sizes. We currently generate the layout using the bok-Layout object, which does not allow for responsiveness; we would have to use Flask to embed divided sections of the layout in a webpage that correctly responds to resizing.

Lastly, one user made a suggestion to create a feedback tool on the webpage. Since the visualization system is not yet ready for deployment, it will likely be used by many user testers who will have feedback for us. Providing a tool on the webpage that allows them to directly enter their feedback would be very useful for future development of the system.

4.3 Further Optimizations and Expansions

In order for our visualization system to be deployable, there are at least four tasks that need to be completed: optimizing the set of datasets plotted by the fixed plots, optimizing delay across the entire system; providing more input data to be integrated into the system, and expanding the server capability in order to serve securely to the desired users. There likely will be many more tasks that need to be completed, but these four tasks are particularly critical for the long-term usability of the visualization system.

4.3.1 Optimizing Datasets for Fixed Plots

The fixed plots on the webpage (Biochemistry, Respiratory, Vitals, Score) contain datasets that we chose because we thought they would go well together or because they demonstrated a use case for us. However, it is very likely that users of the visualization system will want different fixed plots that are more clinically relevant. Future contributors will have to ask potential users what fixed plots they would like to
see on the page. The set of datasets chosen for the fixed plots may vary depending on the type of user; for example, a research scientist may want to view different datasets than a physician. In that case, developers need to find a way to serve different versions of the webpage to different users. We will explore this idea more when we discuss expanding server capabilities.

4.3.2 Optimizing System Delay

When describing the visualization system architecture, we indicated many instances where the user would experience a delay. In the future, these delay times should be optimized to be as small as possible. For example, the `vitalEvents` pull time almost certainly cannot be lessened by modifying the Python code - any major pull time improvements will come from optimizing the database query manager. We also noticed that there is a three to four second delay between the user pressing the Reset button in a plot toolbar and the plot view resetting to standard view. This delay can likely be shortened by modifying the Bokeh code.

4.3.3 Expanding Input Data Flow

Currently, our only source of data is the data stream. While the data stream is very valuable because it is carrying live data, it is also missing relevant data from Beth Israel that we would like to have in our system (e.g. live vitals, admission notes). It is a lengthy process to add new data to the data stream, but the expanded set of data will definitely prove to be advantageous to users of our system.

Additionally, LCP-MIT is interested in expanding the system to be able to incorporate data from other sources. One such source is MIMIC III (Medical Information Mart for Intensive Care), which is a LCP-managed critical care database with data from over 40,000 Beth Israel patients. By creating functionality to incorporate different sources of data, the visualization system will increase its usage flexibility for future users.
4.3.4 Expanding Server Capability

Our server, which is a combination of the Bokeh server and the Flask web framework, is capable of serving to any URL; however, it is currently not prepared to do so. There are two main reasons for this. The first, as we have discussed before, is that we cannot simply serve the interface to a general URL because the data contained on the page is confidential patient data. There needs to be stringent security on the webpage so that only authorized users can access the data. The second reason is that the Bokeh server does not scale well. If there are ten webpages open at once, then the server will have generated ten bok-Docs and will be running all ten documents’ interaction callbacks in a single Bokeh thread. One thread will not be able to handle the ten webpages’ interactions without significant slowdown of the entire system. It is likely that multiple Bokeh servers will have to be spun up to accommodate for the demand.

We would also have to generate multiple Bokeh servers if different users wanted to access webpages with different sets of fixed plots. We have mentioned the need for personalized fixed plots in a previous section. No single bok-Doc can have two or more sets of fixed data sets without becoming increasingly slow to react to webpage interactions, and one instance of a Bokeh server can only make multiple copies of the same bok-Doc. Therefore, to generate different bok-Docs with personalized fixed plots for different users, we would need to run a separate Bokeh server for each personalized bok-Doc. Future developers will have to modify the overall server architecture carefully in order to be able to spin up and manage several Bokeh servers at once.
Chapter 5

Summary

We believe that the version of the visualization system described in this thesis provides a solid foundation to be expanded upon in the future. Our system can easily be manipulated and customized at almost all stages of the data flow. Future developers will be able to add data to the system either via the data stream or by directly adding data to the database. Researchers can currently add and test their data algorithms by implementing custom datasets in the visualization system. Clinicians can ask for certain default plots that they would want to use frequently, and they can take advantage of the interface’s custom plotting for any additional data they might want to view. Our visualization system will require extensive improvement in order for it to be usable on a day-by-day basis, but the current version of the system has made significant progress towards our goal of utilizing ICU data in an efficient manner.
Bibliography


