



FINAL REPORT

RESIDENTIAL HVAC MONITORING & REPORTING SYSTEM
Software Engineering CSCE 3444

THE OFFICIAL GROUP

MATTHEW FREDERICKSEN

IAN OCASIO

LUIS ROMAN

ERIC EASLEY

Contents

1. System Overview.....	2
1.1. System description.....	2
1.2. System Features.....	2
2. Implementation	3
2.1. Data Gathering	3
2.2. Data Storage and Monitoring	6
2.3. Data Interaction	7
2.3.1. Live View	7
2.3.2. Historical View	7
2.3.3. Settings.....	8
3. Challenges	9
4. Test Cases.....	10
5. How to Use?	11
6. Future Works	12
7. Member Contribution	12
8. Appendix: Code	13
8.1. Sensors API.....	13
8.2. Configuration API	13
8.3. React App	15
8.3.1. Prometheus Queries	15
8.3.2. Live View	16
8.3.3. Historical View	17
8.3.4. Settings.....	19

1. System Overview

1.1. System description

Give a description about your system, what is it called? what does it do?

Our Residential HVAC Monitoring & Reporting System... monitors and reports on a residential HVAC (heating, ventilation, and air conditioning) unit. It does this by measuring and recording statistics on the performance of the HVAC unit, and allowing the user to view these metrics through a web app interface. The system also has the capability to alert the user via email when certain conditions occur, such as when the performance is projected to fall below a designated threshold within a certain amount of time.

1.2. System Features

Describe the features or functionalities of your system. Were you able to implement all your functional requirements?

These requirements have been copied from our original Software Requirements Specification document. The following color scheme will be used:

- **Green:** Complete
 - **Yellow:** Mostly complete
 - **Orange:** Partially complete
 - **Red:** Not implemented
- Collect all relevant statistics on the AC system via sensors or third-party APIs.
 - Relevant statistics include incoming air temperature, outgoing air temperature, air pressure, outside air temperature, count of on/off cycles, and any others which are later determined to be reliable predictors of system events.
 - 2 temperature sensors and one air pressure sensor will be used. These sensors will be read using a Raspberry Pi.
 - The user will be responsible for installing the sensors on their AC unit. Detailed instructions will be created for the installation process¹.
 - Weather data will be gathered using OpenWeather or another free API.
 - Record statistics in a database long-term capable of trend analysis.
 - Data will be recorded with timestamps at 15-second intervals.
 - The database will be hosted by the user.
 - Detailed instructions on database configuration will be created for the user.
 - Provide a web interface for a user to view the recorded statistics.
 - The web interface will be hosted by the user. Detailed instructions on how to set up and run the web interface will be created.
 - The web interface will provide a live view of data being gathered from sensors and other sources².
 - The web interface will provide a historical view of statistics consisting of aggregated values (e.g. min/max/average/sum) over an arbitrary range of time selected by the user.
 - Alert the user when statistical trends are detected which indicate an approaching system event (e.g. cooling failure, filter replacement, etc).

- The initial product will only detect system events based on current state. E.g. an alert would be created if air pressure rises too high (indicating a dirty filter) or if temperature differentials become too small (indicating a cooling failure)³.
 - Later versions will be able to detect trends and predict when a system event will occur.
- Provide an interface for a user to configure which alerts they receive, how they receive them, and the frequency at which they are received.
 - Alerts will be sent via email.
 - The user will be able to set a single email address at which to receive alerts. Later product versions may incorporate the ability to add several email addresses.
 - The user will be able to select from a list which alerts they would like to receive and how often the alerts should be sent until their issues are resolved⁴.

Notes on the above list:

1. We successfully installed our sensors on an HVAC unit, but have only completed a [brief outline of the setup process](#) for the Raspberry Pi.
2. The live view is very nearly complete. However, the data is unformatted, units of measurement are not displayed, and it is not intuitively obvious that a timer is refreshing the data every 15 seconds.
3. The alerting system is fully operational, but queries have not yet been added for specific error states. However, the system does use linear regression to predict system failure.
4. The foundation for this requirement is complete (alerts are functional and a server can receive configuration requests), but an interface has not been created for opting out of specific alerts.

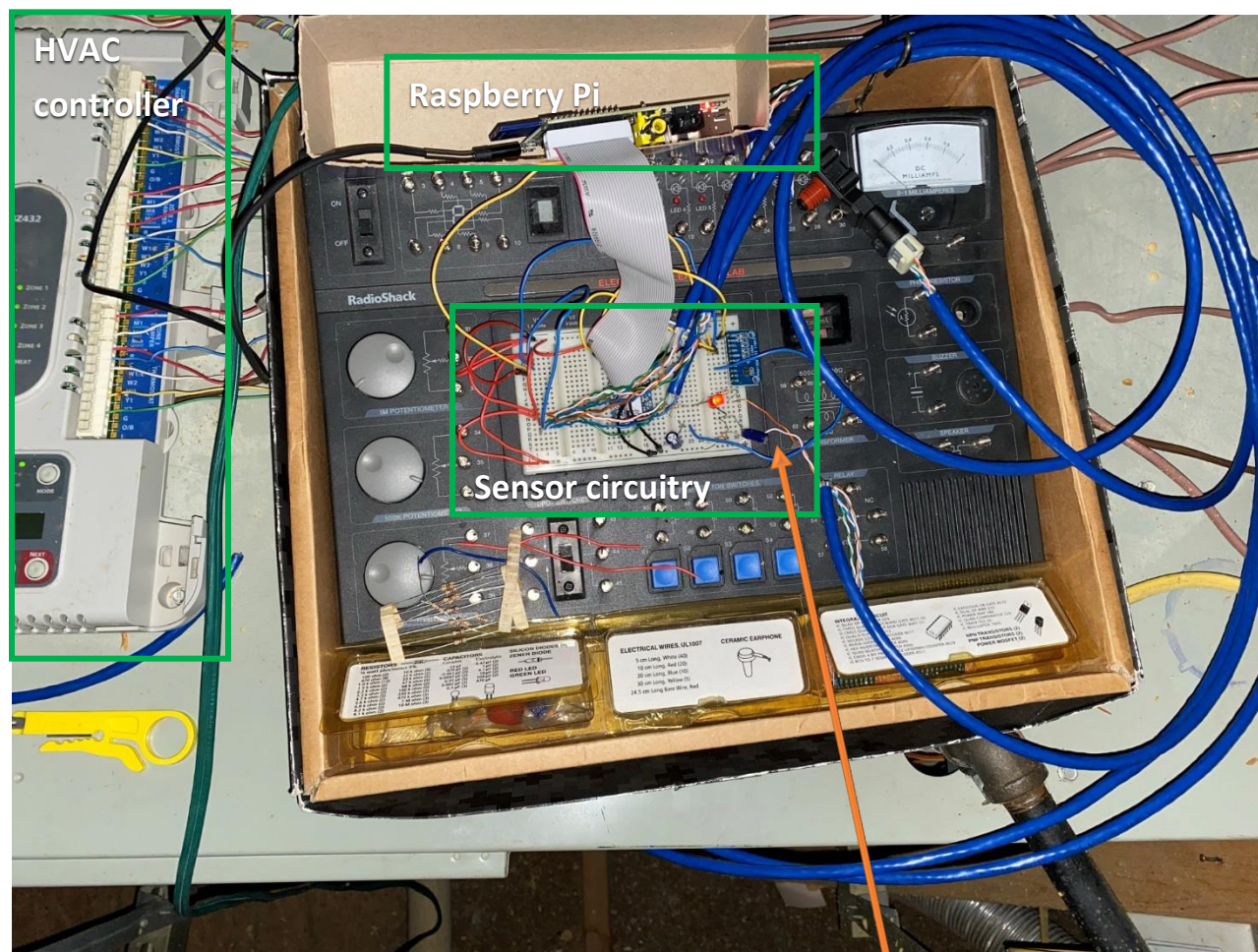
2. Implementation

Describe how you implemented your system. What technologies did you use? What modules, pages or interfaces do you have? How did you implement each one? Include images of the interfaces in your system e.g. home page.

Our system can be logically divided into 3 categories: data gathering, data storage and monitoring, and data interaction. The data gathering process is responsible for transforming real-world information into digital values, and making this information accessible to other processes. The data storage and monitoring process is responsible for long-term data retention and examining data values and trends to check for warning and failure events. The data interaction process is responsible for creating an interface allowing a user to view useful information as easily as possible.

2.1. Data Gathering

We implemented the data gathering process using a Raspberry Pi with Adafruit sensors and a Flask web server. The code can be [viewed on GitHub](#). Adafruit provided Python libraries for each of their sensors, which made integrating the software with the hardware fairly simple.

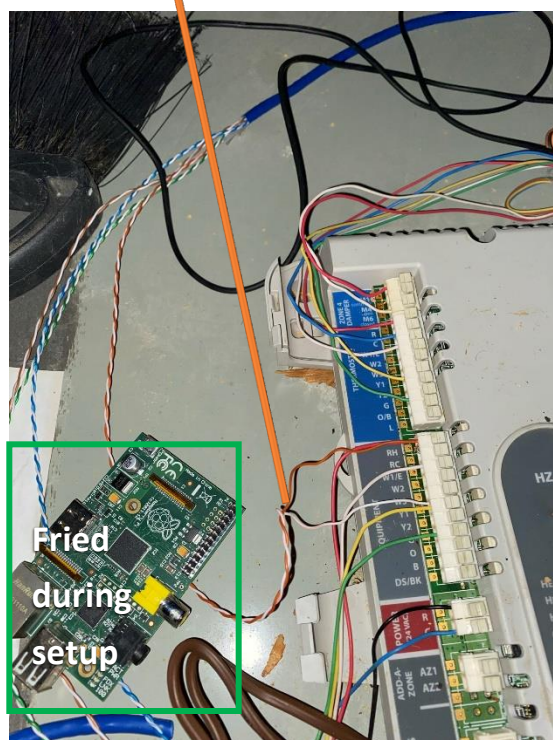


The hardware setup required:

- 1 [Raspberry Pi Model 2B](#).
- 2 Adafruit [MCP9808](#) temperature sensors.
- 1 Adafruit [BME280](#) pressure sensor.
- 1 Adafruit [ADS1115](#) analog-to-digital converter + some additional circuitry to safely read AC controller on/off signal.
- Blood, sweat, and tears.

The orange cables connected by the arrow are used to detect whether the unit is on or off. The long blue cables are used to extend sensors into the unit for reading internal temperatures and pressure.

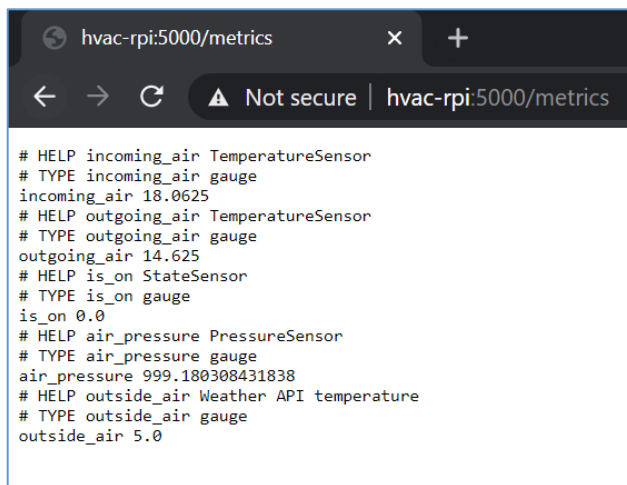
We will not report further on the circuitry configuration here, since this project is primarily focused on software development.



For the software, we used the following modules:

- The aforementioned Adafruit modules.
- [flask](#), a web server framework.
- [flask-restful](#), used to simplify our API.
- [prometheus client](#), used to expose sensor data for collection by Prometheus.
- [pytest](#), a Python testing framework.
- [Other supporting modules](#).

We created two HTTP endpoints using Flask: one showing real-time sensor data in JSON format, and the other for metric scraping performed by Prometheus.

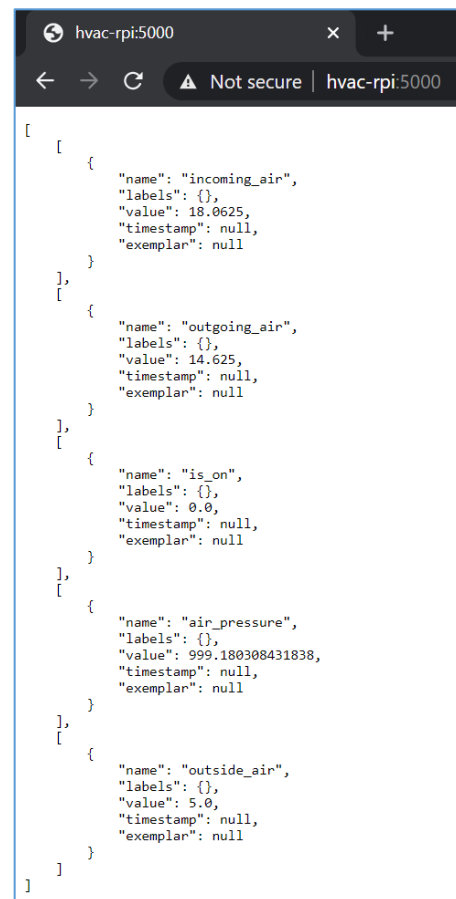


```
# HELP incoming_air TemperatureSensor
# TYPE incoming_air gauge
incoming_air 18.0625
# HELP outgoing_air TemperatureSensor
# TYPE outgoing_air gauge
outgoing_air 14.625
# HELP is_on StateSensor
# TYPE is_on gauge
is_on 0.0
# HELP air_pressure PressureSensor
# TYPE air_pressure gauge
air_pressure 999.180308431838
# HELP outside_air Weather API temperature
# TYPE outside_air gauge
outside_air 5.0
```

The JSON endpoint is not used by other processes in our system, but it is used by a PyTest script to verify that all sensors are reading values correctly.

The /metrics endpoint is accessed by Prometheus in the data storage process. The returned HTML is automatically generated by using metrics classes provided by [prometheus client](#).

If for any reason a sensor is unable to read a value (e.g. due to an I2C communication failure), the application is configured to return a “NaN” value, which allows the rest of the system to continue operating, instead of crashing.



```
[
  {
    "name": "incoming_air",
    "labels": {},
    "value": 18.0625,
    "timestamp": null,
    "exemplar": null
  },
  {
    "name": "outgoing_air",
    "labels": {},
    "value": 14.625,
    "timestamp": null,
    "exemplar": null
  },
  {
    "name": "is_on",
    "labels": {},
    "value": 0.0,
    "timestamp": null,
    "exemplar": null
  },
  {
    "name": "air_pressure",
    "labels": {},
    "value": 999.180308431838,
    "timestamp": null,
    "exemplar": null
  },
  {
    "name": "outside_air",
    "labels": {},
    "value": 5.0,
    "timestamp": null,
    "exemplar": null
  }
]
```

2.2. Data Storage and Monitoring

The data storage and monitoring process is performed entirely by [Prometheus](#), in conjunction with Prometheus's [AlertManager](#). Prometheus is an “open-source systems monitoring and alerting toolkit,” and manages a time-series database to perform these functions. Most of the development for this process was simply learning how to create the correct configuration files, which can be [viewed on GitHub](#).

Prometheus exposes stored data through an HTTP query API, allowing our data interaction process to retrieve and display data over an arbitrary range.

Our Prometheus instance is configured to read values every 15 seconds from the `/metrics` endpoint exposed by the data gathering process. A rules file also specifies a set of database queries which act as alert triggers; if they match any values when executed, Prometheus sends the data to the AlertManager, which aggregates alerts and manages the notification process.

As of the time of writing, three alert triggers have been configured. The first two, for development and testing purposes, trigger every time the system turns on or off. The third executes the following query once a day:

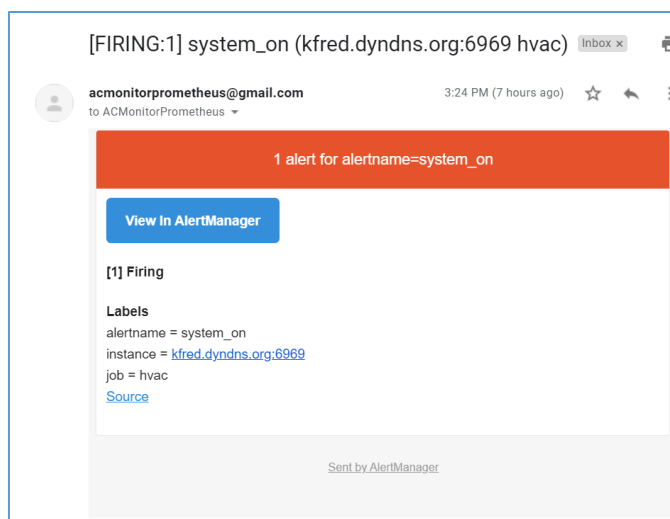
```
predict_linear(max_over_time((incoming_air - outgoing_air)[1d:15s])[2w:1d], 60*60*24*30) < 0
```

This query first collects the maximum difference between the incoming and outgoing air temperatures of the HVAC unit every day for two weeks. Then, it performs linear regression on the values to get a trendline and predict what the value will be in a month (60*60*24*30 seconds). Currently, this alert triggers if the returned value is less than 0; this is because we have not had sufficient time to gather data and determine an appropriate threshold value.

An example alert is shown on the right.

The “Source” link in the email automatically takes the user to a graph showing the data which triggered the alert.

The AlertManager has been configured to send emails as soon as alerts are received, although no more than once per minute. The email recipient is configured by the user through the data interaction process, using our [configuration API](#).



2.3. Data Interaction

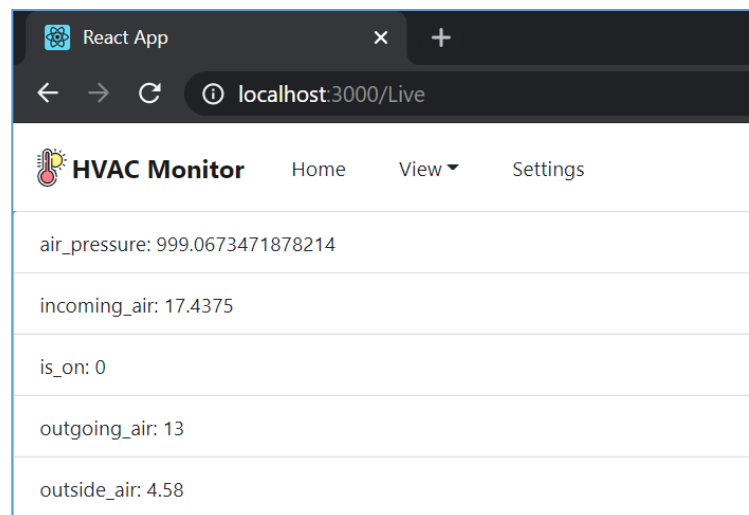
The data interaction process is the user-facing component of our system. Using [ReactJS](#), we created a web application with two main views: Live, which displays the most recent values scraped by Prometheus and automatically updates every 15 seconds, and Historical, which provides an intuitive interface allowing the user to select a time range over which to retrieve data. We also created a Settings page, which is incomplete but allows the user to configure AlertManager's recipient email address.

A significant portion of the logic for data interaction is contained in [prometheus.js](#). All direct interactions with the data storage process are encapsulated by objects and functions in that file. The code for other core UI components can be [viewed on GitHub](#).

2.3.1. Live View

As mentioned in §1.2, the [Live view](#) is incomplete. However, it can retrieve the most recently scraped metrics from Prometheus and display them, albeit without units or other formatting. The air pressure is reported in kPa, temperatures are in degrees Celsius, and the system state is a Boolean value.

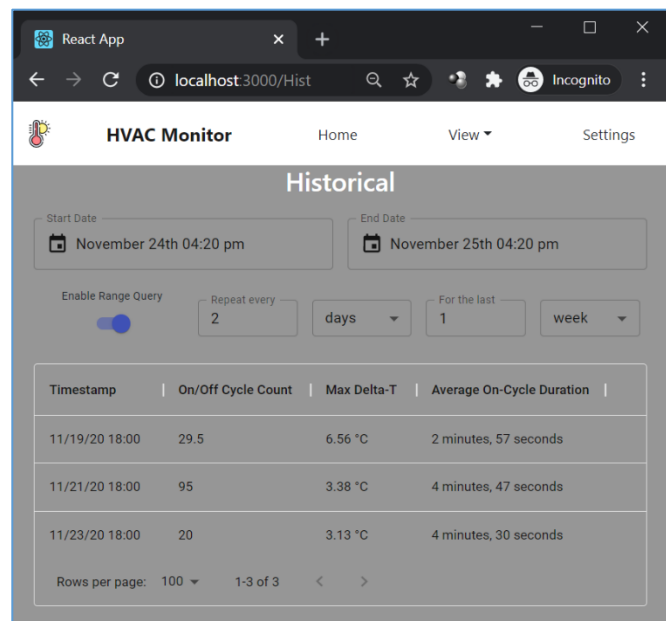
The Live view uses the JavaScript `setInterval` function to create a recurring timer that triggers an asynchronous update function every 15 seconds. This function queries the current metrics from Prometheus and returns the values to the Live view, which then displays them.



2.3.2. Historical View

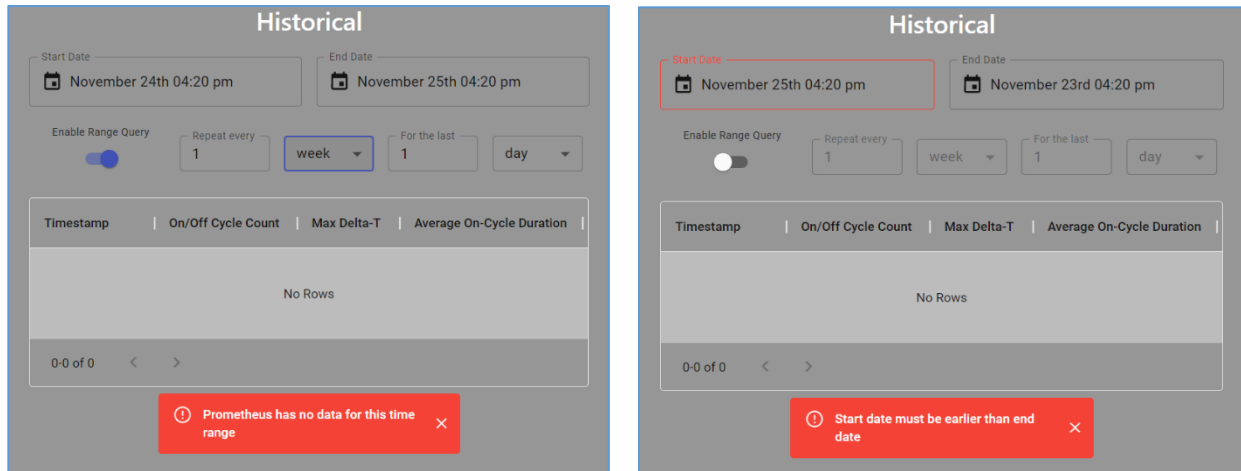
The [Historical view](#) allows the user to create queries using a visual interface and interact with the results. As with the Live view, all data retrieval occurs asynchronously with JavaScript Promises, so the UI remains fully interactive while data retrieval occurs.

Every component of this view was built on [Material-UI](#). Behind the scenes, [Moment.js](#) is used to convert all inputs into duration objects, which allows convenient value checking and passing. After inputs have been validated, a function from [prometheus.js](#) is called, which dynamically creates and returns the results of queries based on the user's input.



The results are then loaded into Material-UI's [DataGrid](#) component, which automatically supports pagination for long inputs and sorting by column. The sorting is reliable even though the data is displayed in string format; the data is actually stored in integer form, and formatter functions are applied before displaying.

The Historical view also utilizes Material-UI's [SnackBar](#) and [Alert](#) components to display non-intrusive alerts when errors occur or when user input is invalid.



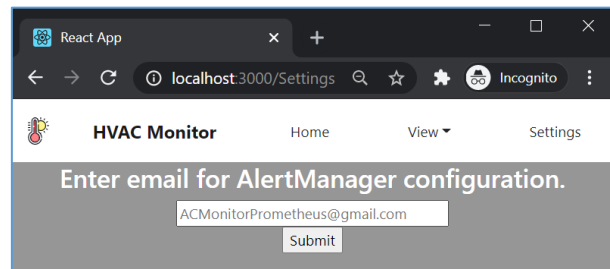
See §5 for more information on Historical view usage.

2.3.3. Settings

As indicated in §1.2, the Settings page is currently incomplete, although it still performs a useful function.

This page works in conjunction with our [configuration API](#), which runs on a Flask app on the same host as the Prometheus and AlertManager instances, so it has access to their configuration files. Our project contains templated versions of configuration files, allowing the configuration API to use [Jinja templating](#) to very easily replace specific values in the configuration. After changing values, the API will attempt to load the new configuration by sending a POST request to the Prometheus or AlertManager [reload endpoint](#). If this fails, the API will replace the configuration files with their previous contents.

Currently, the Settings page only supports updating the recipient email for the AlertManager. When the page loads, a GET request is sent to the configuration API's /email endpoint, which returns the currently configured email address. This value is then set as the input field placeholder, so the user knows what the current value is. When a new value is submitted, a PUT request containing the value is sent to the configuration API's /email endpoint. If a code 200 response is returned, then a success alert is displayed. If the AlertManager failed to load the new value, then a failure alert is displayed with an error reason.

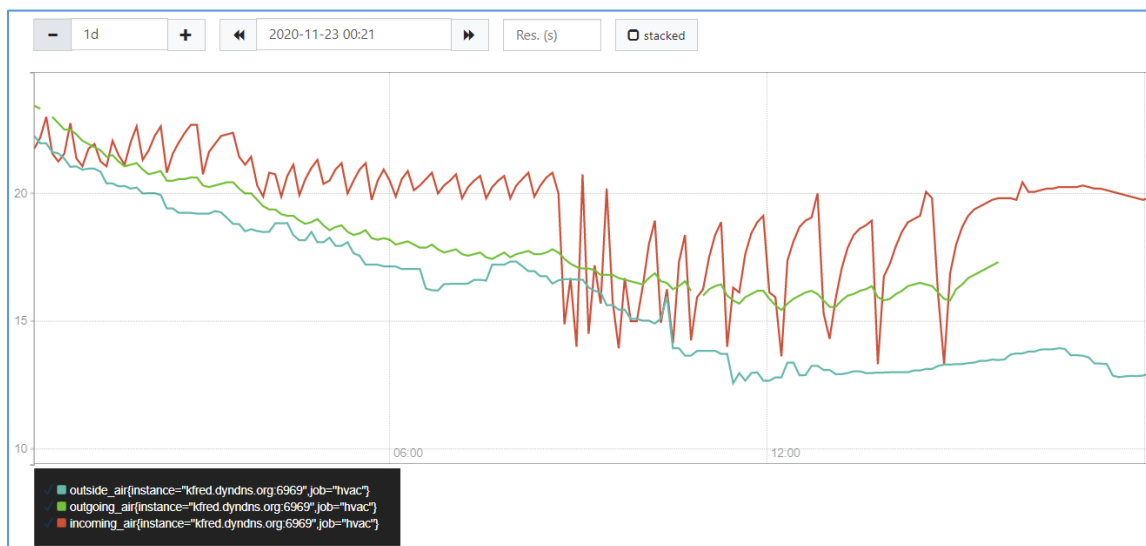


3. Challenges

What challenges did you face while developing this system? What changes did you have to make to your requirements or system features in response to these challenges?

1. **Lack of contributions from team members.** See §7.
2. **Hardware reliability.** When we selected the sensors for the data collection process, we prioritized convenience and price over accuracy and reliability. While we believe this is a good choice for development and prototyping purposes, it is apparent that for our system to be useful in production, higher quality sensors would be required.

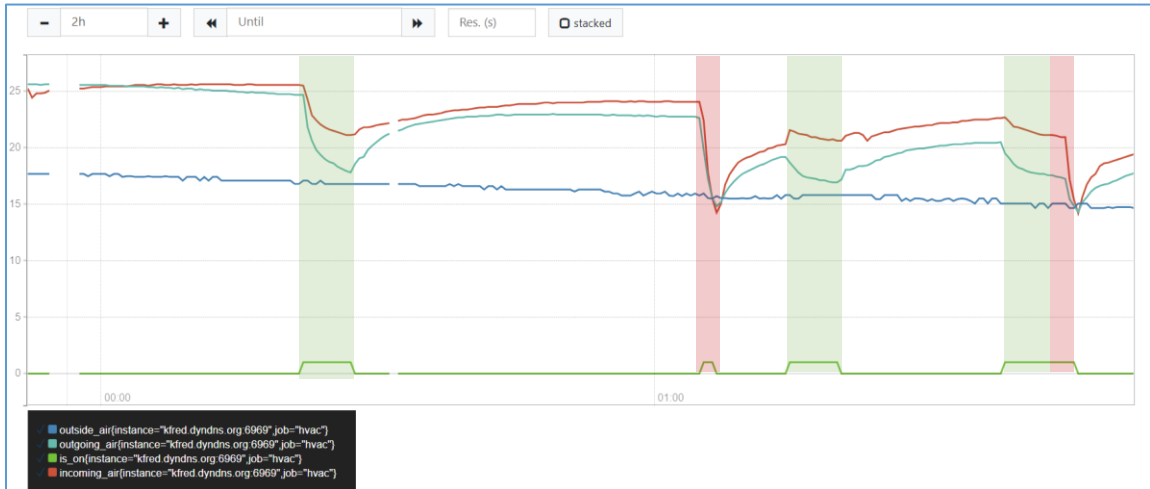
As shown in the image below, there were multiple times when the sensor readings became erratic or the sensors quit responding entirely. It is possible that our installation of the sensors was causing these issues, but it seems more likely that the problems were caused directly by the sensors.



We also experience issues with the I2C address of one of the sensors switching every few hours. Under normal circumstances, this should never happen. This problem suddenly stopped happening after a few days, which again indicates unreliability in the sensor hardware, rather than our installation process.

3. **HVAC unit design.** The unit we installed our sensors on used a zone controller with dampers, which allows a single HVAC unit to target multiple zones in a house. This caused two issues for us. First, we expected to be able to gather useful information based on the internal pressure of the unit. However, our collected data shows only a variance of 2-3 kPa between the on and off states of the unit. This might still be useful if this value were stable, but when multiple zones are active simultaneously, air flow becomes less restricted as dampers open, causing the pressure variance to decrease. Weather events also affect atmospheric pressures, shifting our sensor readings by as much as 30 kPa, which essentially removes all predictive value of this sensor unless much more advanced logic is added to our software.

Additionally, one of the zones is significantly smaller than the other zones. Unfortunately, this means that the unit opens a bypass valve to relieve pressure when only this zone is running, creating a cycle between the output and input on the unit. This causes our sensor readings to become essentially the same during this time, as show in the following graph. The larger zones' runtimes are shaded green, and the small zone's runtimes are shaded red.



4. **Project timing.** The timing of our project prevented us from collecting enough useful information on how the system should be configured for system alert thresholds. The purpose of the system is significantly more focused on AC performance than heating performance, and as we have approached winter during the final stages of the project we have become unable to gather information on AC performance, since the AC is no longer in use.

4. Test Cases

Include the test cases you used to test your system. Show test cases using the test case template provided in lecture 12.

We did not develop any high-level test cases as described in Lecture 12. We believe §7 provides a sufficient reason for this. However, some simple test cases were created for the sensors API using the PyTest framework, and can be [viewed on GitHub](#).

5. How to Use?

Provide a step by step description on how a user would use your application.

As specified in our Software Requirements Specification document, this system is designed to be installed by a homeowner who has some technical knowledge with software and a moderate understanding of the design and functioning of their home HVAC unit(s). As such, the first steps require the user to install the system.

1. Obtain and install hardware.
 - a. Accept full personal responsibility for any damages associated with modifying the HVAC unit.
 - b. Purchase and setup hardware as specified on our [Raspberry Pi setup instructions page \(incomplete\)](#).
 - c. Clone, setup, and launch the sensors API according to the [instructions on GitHub](#).
2. Initiate data storage process.
 - a. The user must have a computer always running to reliably retrieve and store data from the sensor API.
 - b. The user should follow the Prometheus and AlertManager [setup instruction on GitHub](#).
3. Launch the ReactJS web application.
 - a. Instructions do not yet exist for this setup process, although it is fairly simple. The user just needs to install [Node.js](#), clone the [React app source files](#), and launch the app.

Once setup and installation is complete, the user can interact with the application.

4. Go to `http://<local_server_IP:port>/Live` to view active sensor readings on the HVAC unit.
5. Go to the `/Historical` endpoint to view data from any time range during which the system was active.
 - a. *Range query mode off*: Select start and end datetimes. The application will create queries by applying these inputs to query templates, and a single row will be displayed containing the data for the selected datetime range. For example, suppose the user wanted to view useful aggregated data over the last day, instead of just the current values in the Live view. The user simply selects the previous day for the start date, and the application retrieves the data.
 - b. *Range query mode on*: Select start and end datetime, as well as how frequently the same query should be repeated, and for how long. For example, suppose the user wished to compare the data collected over this year's summer months to the previous 2 years' summer months. The user first selects this year's summer months with the start and end datetime pickers. Then, the user sets the remaining inputs to "repeat every 1 year for the last 3 years." Three rows will be loaded containing the aggregated values for the desired time ranges.
 - c. If enough rows are loaded that they cannot all be seen at once, the user can sort by column if desired to show the least and greatest values for the different aggregates.
6. Go to the `/Settings` endpoint to configure the email address to which system alerts should be sent.
7. (Passive) Receive alerts when HVAC system performance failure is predicted.

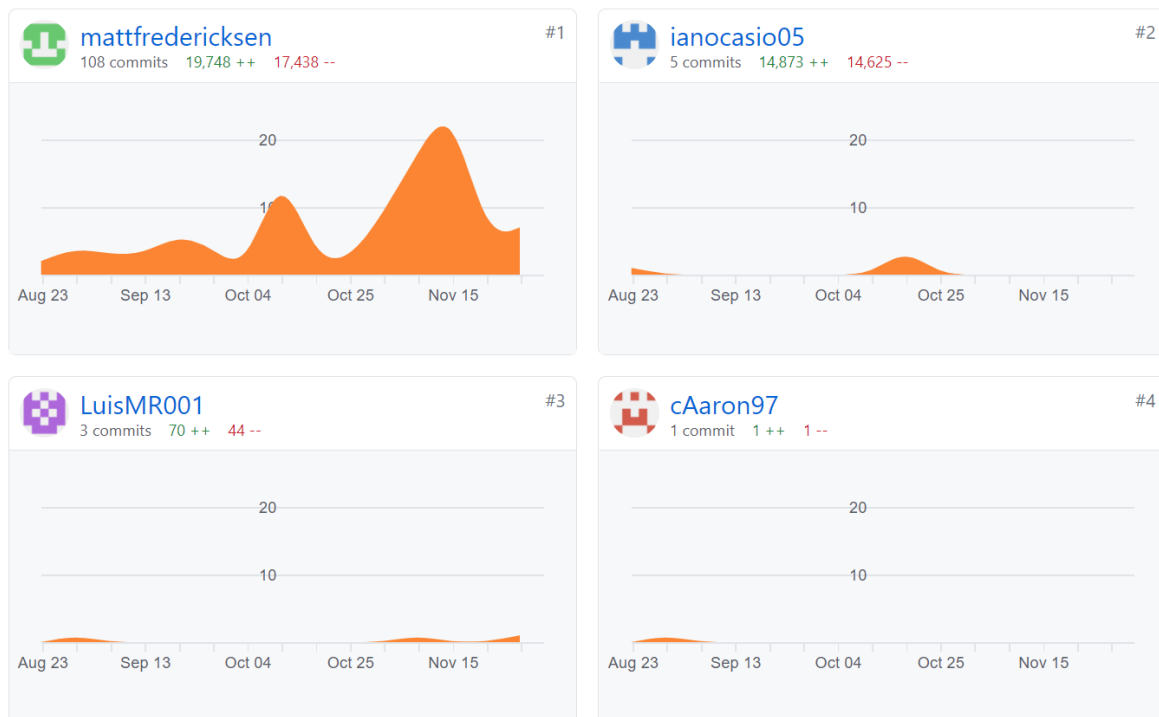
6. Future Works

Any future improvements you would like to make on your application.

- Discover more queries to display in the HvacDataGrid.
- Add [MUI Chips](#) so the user can select which queries will be shown.
- Add more configurations to the Settings page, such as alert subscriptions and alert frequencies.
- Add better data visualizations using [Grafana](#).
- General UI improvements
- Add input parameters to the URL so that when a page is refreshed the input values are not lost.

7. Member Contribution

Describe each member's contribution to the project.



The overwhelming majority of work on this project was performed by Matthew Fredericksen. At the time of writing, Ian Ocasio has some uncommitted UI enhancements in progress.

8. Appendix: Code

Include the code of some of your core functionalities.

8.1. Sensors API

From [sensors.py](#):

```
class Sensor(Gauge):
    ...

    def read(self):
        raise NotImplementedError

    def collect(self):
        # this override allows sensors to refresh
        # immediately before yielding metrics to Prometheus
        try:
            self.set(self.read())
        except Exception as e:
            print(e, f'Error reading sensor: {self._name}', sep='\n')
            self.set(float('NaN'))

        return super().collect()
```

This is the base class from which our other sensor classes inherit. The child classes are only required to implement a “read” method, which should return the current value of the sensors.

The “collect” method is overridden from the [Gauge class](#) of the `prometheus_client` library. Whenever Prometheus scrapes metrics from the sensors API, all instances of this class call their “collect” methods to return current metric values. By overriding “collect,” we are able to retrieve the current sensor values immediately before they are passed along to Prometheus for storage.

The following is a child class of `Sensor`, utilizing the simple API of our Adafruit hardware.

```
class TemperatureSensor(Sensor):
    def __init__(self, addr, name, *args, **kwargs):
        super().__init__(name, *args, **kwargs)
        self.device = MCP9808(i2c, address=addr)

    def read(self):
        # MCP9808 chip returns temperature in celsius
        return self.device.temperature
```

8.2. Configuration API

The [configuration API](#) is used by the Settings page of the React app for modifying server-side configurations. Currently, endpoints only exist for configuring the `AlertManager`’s recipient email address.

Sending a GET request to the `/email` endpoint parses the YAML configuration file to retrieve the current email address:


```

class Configuration(Resource):
    @staticmethod
    def get():
        """Returns the currently configured email address."""
        try:
            with open(args.config, 'r') as file:
                config = yaml.safe_load(file)
            return {'email':
                    config['receivers'][0]['email_configs'][0]['to']}
        except Exception as e:
            print(f"{e!r}\nUnable to parse file: \"{args.config}\"")

```

Sending a PUT request to the /email endpoint attempts to replace the configured email with a new value. If reloading the configuration with the new value fails, the configuration is reset to its previous state.

```

@staticmethod
def put():
    """Sets the receiver email for AlertManager."""
    req_args = req_parser.parse_args()
    with open(args.template, 'r') as file:
        config = Template(file.read()).render(email=req_args['email'])
    with open(args.config, 'r+') as file:
        old_config = file.read()
        file.seek(0)
        file.write(config)
        file.truncate()

    try:
        # address at which the AlertManager is running
        r = requests.post("http://localhost:9093/-/reload/")
        if r.status_code != 200:
            # restore the previous configuration
            with open(args.config, 'w') as file:
                file.write(old_config)
            return {'error': "Unable to reload AlertManager configuration",
                    'responseContent': str(r.content, r.encoding)},
                    r.status_code
        return {'email': req_args['email']}
    except Exception as e:
        # restore the previous configuration
        with open(args.config, 'w') as file:
            file.write(old_config)
        print(e)
        return {'error': "Unable to communicate with AlertManager",
                'errorType': str(type(e))}, 500

```

8.3. React App

8.3.1. Prometheus Queries

The Query class in [prometheus.js](#) is used to make handling queries and their executions as easy as possible. Each query has a name, template, and colSettings attribute. The name and colSettings attributes are used to correctly format the HvacDataGrid in the Historical view. The Query.execute method takes the user's input, properly formats it, creates a valid query from the template, sends the query as an HTTP request to Prometheus's API, and calls the Query.extract method to retrieve data from the HTTP response.

```
async execute(range, offset, duration, resolution) {
  const [rangeStr, offsetStr, durationStr, resolutionStr] = (
    [...arguments].map(durationString)
  );

  let query = this.template.replaceAll("{r}", rangeStr);

  if (duration && resolution) {
    query = query.replaceAll("{o}", "");
    query = `(${query})[${durationStr}:${resolutionStr}]{o}`;
  }

  query = query.replaceAll("{o}",
    offset && offset.asMinutes() > 1 ? ` offset ${offsetStr}` : "");

  return this.extract(await fetch(`${api}query=(${query})`), offset);
}
```

It is extremely simple to add a new Query to the application by creating a new entry in the “Queries” array.

```
export const Queries = [
  ...
  new Query(
    "Max Delta-T",
    "max_over_time((incoming_air - outgoing_air)[{r}:15s] {o})",
    {
      width: 115,
      valueFormatter: ({value}) => (
        isNaN(value) ? value : `${value.toFixed(2)} \xB0C`
      )
    }
  )
  ...
]
```

The “rowsFromQueries” function is also noteworthy; it takes all defined queries, executes them, and then matches the coinciding values from each query result based on their timestamp values. It automatically handles the cases when timestamps do not match or are missing. It is reproduced below with some small modifications.

```

export async function rowsFromQueries(range, offset, duration, resolution) {

  // create a sorted array of unique timestamps
  let timestamps = new Set();

  // get an array of pairs, first item is query name,
  // second item is array containing query results
  const data = await Promise.all(
    Queries.map(async (q) => ([
      q.name,
      Object.fromEntries(
        (await q.execute(range, offset, duration, resolution)).map(
          ([ts, value]) => {
            ts = Math.round(ts);
            timestamps.add(ts);
            return [ts, Number(value)];
          }
        )
      )
    ]))
  );
  timestamps = [...timestamps].sort();

  // For every timestamp, create a row object containing that timestamp
  // and the result of each query at the same timestamp.
  // Results with missing timestamps are evaluated as NaN.
  return timestamps.map(
    (ts, i) => ({
      id: i, Timestamp: ts,
      ...Object.fromEntries(
        data.map(([name, datum]) => {
          const value = datum[ts];
          return [name, isNaN(value) ? "NaN": value];
        })
      )
    })
  );
}

```

8.3.2. Live View

The Live view was the first component created for the React application, and was set aside quickly to prioritize the much more functionally complex Historical view. The only code which might be noteworthy is the use of [ReactJS's component lifecycle methods](#) to automatically begin refreshing the data when the Live view is loaded, and stop refreshing when another view is loaded.

```

componentDidMount() {
  // set this component to fetch newest values every 15 seconds.
  this.update();
  this.timerID = setInterval(
    () => this.update(),
    15000
  );
}

componentWillUnmount() {
  // stop polling the database when the component is unmounted.
  clearInterval(this.timerID);
}

```

The `LiveView.update` method asynchronously retrieves the most recent values and updates the state of the Live view, causing it to re-render.

8.3.3. Historical View

The [Historical view](#) is rendered with the following JSX:

```
<Container maxWidth={"md"}>
  <Grid container spacing={2} justify={'center'} alignItems={'center'}>
    <Grid item xs={8}>
      <h3 style={{textAlign: 'center', color: 'white'}}>
        {"Historical"}
      </h3>
    </Grid>

    <HvacInputGrid
      onChange={this.setState} setAlert={this.setAlert}
    />

    <Grid item xs={12}>
      <HvacDataGrid
        range={range} offset={offset}
        duration={duration} resolution={resolution}
        setAlert={this.setAlert}
      />
    </Grid>
  </Grid>

  <Snackbar open={Boolean(alert.message)}>
    <Alert severity={alert.severity} onClose={this.closeAlert}
      variant={'filled'}>
      {alert.message}
    </Alert>
  </Snackbar>
</Container>
```

It contains 2 major child components: the `HvacInputGrid` and the `HvacDataGrid`.

The [HvacInputGrid](#) encapsulates all user input components, and uses a technique called “[lifting state](#)” to pass these inputs from itself, to the Historical view, to the `HvacDataGrid` for data retrieval. Rather than passing the literal input values, this component converts the inputs into [Moment.js duration](#) objects, allowing the data to be easily passed along without needing to track units or parse strings. The [useEffect hook](#) performs this state update asynchronously so no UI lag is experienced.

```
useEffect(() => {
  onChange({
    range: (startDate.isBefore(endDate) ?
      moment.duration(endDate.diff(startDate)) : null
    ),
    offset: (endDate.isSameOrBefore(moment()) ?
      moment.duration(moment().diff(endDate)) : null
    ),
    resolution: (enabled ?
      moment.duration(resolution, resUnit) : null
    ),
    duration: (enabled ?
      moment.duration(duration, durUnit) : null
    )
  });
}, [onChange, enabled, startDate, endDate,
  resolution, resUnit, duration, durUnit]);
```

This component also performs alerting when invalid inputs are detected.

Once the inputs have been received by the [HvacDataGrid](#), it begins asynchronously retrieving data from Prometheus, using the `rowsFromQueries` function. This code is reproduced with minor modifications.

```
const queryID = useRef(moment(0));

useEffect(() => {
  ...
  // queryID.current will always contain the
  // timestamp of the most recent query
  const effectID = moment();
  queryID.current = effectID;

  rowsFromQueries(range, offset, duration, resolution)
    .then(rows => {
      if (effectID.isSame(queryID.current)) {
        setRows(rows);
        ...
      }
    })
    .catch(error => {
      if (effectID.isSame(queryID.current)) {
        setRows([]);
        setAlert(error.message);
        ...
      }
    });
}, [range, offset, duration, resolution]);
```

By using the [useRef hook](#) and comparing this against the `effectID` when the query results arrive, we guarantee that older data will not overwrite newer data from more recent user input.

8.3.4. Settings

The [Settings](#) page works with the [configuration API](#) to retrieve and modify the recipient email address for the AlertManager. The submitHandler function uses the incredibly simple API provided by the [axios](#) package to send the PUT request and check the configuration API's HTTP response.

```
const submitHandler = useCallback(
  (event) => {
    event.preventDefault();
    axios.put(api + 'email/', {email: email})
      .then(({request: {response}}) => {
        setEmail({target: {value: ""}});
        setPlaceholder(JSON.parse(response) ['email']);
        alert("Success");
      })
      .catch(err => {
        console.error(err);
        if (err.response) {
          alert(JSON.stringify(err.response.data, null, 2))
        }
      });
  }, [email, setEmail]
);
```