Disaster Tracker
Masters of Disaster
Brandon Cole       Jacob Peterson       Nsama Chipalo
Aaron Damrau       Jhossue Jimenez

Project Sponsor
Christopher Serafin, Sharon Malfesi

Faculty Coach
Samuel Malachowsky

Project Overview
Datto Inc. is a vendor of backup, disaster recovery (BDR), and Intelligent Business Continuity (IBC) devices. Customers purchase Datto’s solution in preparation for events such as power outages, floods, fires, hurricanes, tornadoes, and earthquakes. Currently, Datto has no way to inform customers of weather events that may affect their company. The goal of the Disaster Tracking and Alerting project is to provide Datto with a way to proactively warn customers of inclement weather events that may affect their business, ensuring they are prepared before disaster strikes and reducing the losses incurred in such a scenario.

In the scope of this project, the customers are solely Datto and the Managed Service Providers (MSPs) that Datto partners with. The end users of Datto devices are not customers for this project.

The team implemented two solutions to solve this problem; a web application for use by Datto and their MSPs and a mobile application for use by the MSPs. The web application will display all devices across the United States when used internally by Datto. Devices are color coded based on the current risk level of the device, allowing Datto to identify at risk areas across the U.S. at a glance. A sortable, searchable list of all devices sorted by risk level is also included to assist in locating at risk areas. When an MSP uses the web application, all the same features will still be available, although they will only see the devices that they own. The mobile application will display a list of all devices an MSP owns, sorted by risk level. When devices owned by an MSP pass a risk level determined as ‘high risk’, a push notification will be sent to alert the MSP that they have devices at risk. When opened, the application will display more detailed information about these devices such as the cause of the risk. A list of steps to prepare the device for a possible power outage or disaster scenario will also be included to ensure that customers take the necessary precautions.
These solutions utilize geolocation via MaxMind and weather tracking via NOAA (National Oceanic and Atmospheric Association) to generate a risk level for devices that is guaranteed to be up-to-date every ten minutes. A risk algorithm was designed to generate accurate risk values for varying weather situations. For more information on this risk algorithm, please refer to the Risk Algorithm section.

**Basic Requirements**

The following list consists of all of the basic functional requirements of the system that were identified and implemented for both the web and mobile applications:

1. The system shall calculate a risk value from 0-100% for every device, based on current weather conditions pulled from NOAA. This value shall correlate to the percentage chance that the device will experience a power outage.
2. The risk level of displayed devices should be easily identifiable by color coordination, using the red to green color scale to show device risk levels from 0 to 100.
3. Users shall be able to login to the application using an MSP ID and password.
   a. Datto employees will also use an MSP ID for login with ‘admin’ level privileges that allow them to see all devices.
4. Users shall be able to logout of the application.

The following list consists of all of the basic functional requirements of the system that were identified and implemented for only the web application:

1. The system shall display devices on a map as individual dots, correlating to the geolocations for each device given by MaxMind.
   a. If a Datto employee is logged in, all devices should be shown.
   b. If an MSP is logged in, only their devices should be shown.
2. A list of all devices on the map should be displayed in a table, sorted from highest to lowest risk. This list should contain the risk level of the device, the device ID, and the device location.
3. The list of devices should be sortable and searchable by any column in the table.
4. Selecting a row on the device list table should center the map on the point and zoom in to zoom level 4.
5. The user shall be able to filter by risk level which devices are displayed on the screen.
6. The user shall be able to toggle a doppler radar overlay onto the map.

The following list consists of all of the basic functional requirements of the system that were identified and implemented for only the mobile application:
1. The system shall display a list of all devices that the MSP owns, sorted by risk from highest to lowest. The list should contain the risk level of the device, the device ID, and the device location.
2. When a device passes the ‘high-risk’ threshold (currently set at 75%), the device row should become selectable, allowing the user to view more detailed information of the device. This information will include the cause of the risk, the risk level, the device ID, the location, and a list of steps to prepare for the disaster.
3. The system shall send a push notification to the user when one or more devices passes the ‘high-risk’ threshold.
4. The user shall be able to email the list of steps to prepare for the disaster directly through the application.

Constraints
As with any project there are many constraints that come with and arise during the creation of the product. During the requirements elicitation portion of the project, the team received the first set of constraints that revolved around the implementation technologies to be used.

One of the first was the programming language for development of the disaster tracker. Due to the sponsor having already fully adopted PHP as one of their main programming languages, they encouraged adoption of PHP for this project as well.

Google Maps was used to display the map. This meant that any visuals to be displayed on the map, be it heatmaps, dots, map overlays, etc, had to be supported by the Google Maps API.

Datto required a web application for their own use as well as Managed Service Provider use. In addition to the web application was a mobile application. With regards to the mobile application, Datto required a native iOS application.

In designing the first full design of the system, the team took into account being able to retrieve needed information from Datto. Services such as being able to retrieve device information and user authentication were expected to be handled by Datto, and the disaster tracker would simply programmatically request the information when needed.

While currently Datto uses SAML for its authentication, they indicated that they plan to move to OAuth once they had the time. Thus the team had two choices: either develop for what they have now and authenticate users through SAML, or design the system to use OAuth not knowing when Datto will actually fully move over to OAuth. It was decided that building for the future was the best choice and the OAuth approach was chosen. The team hoped that Datto’s OAuth implementation would be completed in time for testing.
However, due to time and personnel constraints, the OAuth implementation, in addition to a necessary API to expose device information, would not be completed in time for product testing or release. This resulted in the design of a mocked system that would implement the needed systems from Datto for the development stages (more information on this design is available in the System Design section). This put several constraints on the design of the system. First, the system had to be as modular as possible. An emphasis on low coupling was placed on the system because the team wanted Datto to be able to easily swap out the mock system for the real implementation when it came time to hand over the product. For geolocation information, Datto required that MaxMind be used. The mocked device API has to return device location information based on the format of data provided by MaxMind.

**Development Process**

For the project, the team chose to use Scrum with two-week sprints. Because Datto uses Scrum, our sponsors encouraged it. Furthermore, given the initial requirements and description of the project, it seemed that requirements volatility would likely happen early on. As such, an agile approach was the best approach. Being an agile process, scrum encouraged frequent communication with the sponsors to ensure the team was building the right product. This proved to be important because the team constantly had to communicate with the sponsors regarding requirements, general questions, etc. While response times from the sponsor were varied, the team followed through with the importance of reaching out to the sponsors.

As for roles, 7 specific roles were identified and taken up by the following team members:

- Point of Contact: Brandon Cole
- Project Manager/Team Lead: Jhossue Jimenez
- Development Leads
  - Web Application: Aaron Damrau
  - Mobile Application: Jacob Peterson
  - Backend Lead: Brandon Cole
- Test Lead: Nsama Chipalo
- Team Website Manager: Brandon Cole

While everyone was still a developer, the roles simply showed what that member would focus on or be the go-to person whenever anything related to their roles came up. The roles were followed appropriately, but as the project went on, the work done had everyone doing mostly everything.

**Project Schedule: Planned and Actual**

Towards the beginning of the project, the team laid out a series of project deliverables and expected release dates. Many of these deliverables and the dates assigned to them were based
of dates assigned to project deliverables such as the mid-term peer evaluation, the project poster, and the final presentation. Other deliverables and their expected deliveries, such as the prototypes and completions of various aspects of the system, were added by the team based off of the best date of completion. Within the project plan, a table was created to list out all deliverables, the type of deliverable, when the team expected to complete the deliverable according to semester schedule, and whether or not the feature was completed.

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Expected Delivery</th>
<th>Released</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Website</td>
<td>Week 2 - 9/02/14</td>
<td>Yes</td>
<td>Project</td>
</tr>
<tr>
<td>Initial Project Plan</td>
<td>Week 3 - 9/09/14</td>
<td>Yes</td>
<td>Project</td>
</tr>
<tr>
<td>Requirements Document</td>
<td>Week 4 - 9/16/14</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>User Stories</td>
<td>Week 5 - 9/23/14</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>UI Mockups</td>
<td>Week 6 - 9/30/14</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Design/Architecture Document (Version 1)</td>
<td>Week 7 - 10/07/14</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Mid-term peer evaluation</td>
<td>Week 7 - 10/07/14</td>
<td>Yes</td>
<td>Project</td>
</tr>
<tr>
<td>API Prototype</td>
<td>Week 10 - 10/28/14</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Give Interim presentation</td>
<td>Week 15 - 12/4/14</td>
<td>Yes</td>
<td>Project</td>
</tr>
<tr>
<td>Web Application Prototype (functioning UI, connected to API)</td>
<td>Week 16 - 12/9/14</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Mobile Application Prototype (functioning UI, connected to API)</td>
<td>Week 16 - 12/9/14</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Test Plan</td>
<td>Week 3 - 02/10/15</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Web Application Complete (UI)</td>
<td>Week 8 - 03/17/15</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Mobile Application Complete (UI)</td>
<td>Week 8 - 03/17/15</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Mock Systems (OAuth and Devices)</td>
<td>Week 11 - 04/14/15</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>API Completion</td>
<td>Week 11 - 04/14/15</td>
<td>Yes</td>
<td>Product</td>
</tr>
<tr>
<td>Code freeze</td>
<td>Week 12 - 04/21/15</td>
<td>Yes</td>
<td>Process</td>
</tr>
</tbody>
</table>
A series of weekly product deliverables was created for the team towards the beginning of the project to provide a good foundation for what the group will implement in the latter half of the first semester of the project. As time went on, the milestones listed on the deliverable chart began to become more spread out and generalized, marking major accomplishments within the weeks of the project, including the creation of the APIs, the front end, and their respective completions. Larger milestones were split up within the schedule in the form of story points that were kept on the Trello board. Since the team utilized the Scrum methodology, tasks were assigned to various team members to be completed within the two week sprint. Typically, most of the tasks assigned to team members were implemented into the project within the two weeks allotted for their completion; however, if the task proved to be too difficult or the team member was blocked in any way, the task would fall out and be assigned during the next two week sprint. Since tasks are just portions of much larger user stories and project deliverables, their delay had much more leeway when it came to overall completion.

The team adhered to the schedule fantastically; every deliverable established in the deliverable schedule was completed within the project time and almost every deliverable was finished on time, occasionally being completed prior to the set due date. Only one scheduled task, the code freeze, was not completed on the date assigned to it; the code freeze was completed a week late because of the preparations for the poster presentation as well as the unveiling of a few underlying issues within the system that needed to be addressed before a code freeze could happen. Even with this delay in the schedule, the team established a flexible schedule that could handle it.

**System Design**

At a minimum, the system needs to provide users a web application and native mobile application. To support these and any possible future interfaces, core logic is contained in an API layer that the interfaces can access. At a high level, the architecture is designed as illustrated in Figure 2.
The web and mobile applications both share common feature requirements. Both need to offer the user a way to login, and then need to retrieve the list of Datto devices and relevant information (including location and risk) to be displayed as appropriate. It is for this reason, in addition to a request from the sponsor, that the decision was made to have an API layer that can be accessed by any interface.

The API is RESTful (Representational State Transfer), and communication is performed over the Hypertext Transfer Protocol using standard HTTP verbs (GET, POST, PUT, DELETE). API responsibilities include:

1. OAuth authentication and resource access (through communication with the Datto API)
2. Periodic retrieval of a list of Datto device information
3. Periodic retrieval of weather information from NOAA
4. Calculating risk values for Datto devices
5. Persisting device and calculated risk information
6. Sending push notifications for at-risk devices
7. Exposing device information to interfaces

These responsibilities are discussed in detail in the following sections.

1. OAuth

The OAuth 2.0 Authorization Framework (http://tools.ietf.org/html/rfc6749) was chosen to meet the login functional requirement. This decision was made in collaboration with Datto, who is currently in a transition phase from the SAML standard to OAuth. Unfortunately, Datto’s timeline did not match the timeline of the senior project team, and at the time of this writing, the necessary OAuth server is in an incomplete state.

Therefore, the team decided to implement a temporary solution. A mocked OAuth server was created that complies with the OAuth 2.0 standards, and will match Datto’s implementation as well (assuming they also follow the standard). An overall illustration of the interactions with this mocked system can be seen in the next section in Figure 3. In addition to allowing the senior project team to avoid ending the project in an incomplete state, this decision was also made with careful thought to future integration plans. Once Datto completes their version of specified implementations, transitioning away from the mocked systems will require no code changes. Instead, only a simple update of URI locations in the configuration file will be required.

The OAuth Server enables three different OAuth grant flows: Authorization Code, User Credentials, and Refresh Token.

The Authorization Code grant type is intended for use by the web application. This grant type was chosen because of the additional security benefits it provides. In this flow, the token is never passed through the user-agent (the user’s web browser). It remains on the server, and is only passed between the API and Datto’s system (responsibility #2). This design is beneficial because having the token available to the user-agent can expose the user to unwanted vulnerabilities and attacks.

The User Credentials grant type is intended for use by the mobile application. Since the native mobile application cannot ensure the safety of the token (all code is located on the user’s device and cannot be considered secure), the authorization code flow is not appropriate. In addition, for mobile usability, it is desirable for the mobile application to be able to implement a custom login screen. The User Credentials flow best meets these requirements.

Refresh Tokens are issued to both the web and mobile applications upon successful authentication using one of the above grant types. A refresh token can be exchanged for a new access and refresh token, and allows the client to be re-authenticated after the access token expires. This flow is implemented on the server to allow potentially infinite authenticated sessions, which matches one of the designed use cases for Datto. Datto plans to have the web
application running continuously in a control room setting, which creates the need for lengthened sessions.

Once successfully authenticated and an access token is received, the access token is passed when subsequent requests are made to retrieve device information. Specifically, when a web or mobile interface makes a devices call to the API, the API then makes a request to the Datto device API with the access token. The Datto device API returns the authorized devices that the user has permission to view. More information on this API is discussed in the next section.

2. Datto Device Information

Datto contains and maintains an up-to-date database of current Datto device information. This includes information that is needed by the disaster tracker, including the device’s ID, the MSP that manages the device, and the device’s location.

In collaboration with the sponsor, the decision was made that this data would be available to the disaster tracker via a RESTful API on the Datto system that they would implement. Unfortunately, this API was not completed in time to match the senior project schedule. For reasons similar to those listed above for the OAuth server, the team decided to also mock an implementation of this API as well. The collection of these mocked systems and their interactions with the disaster tracker can be seen in Figure 3.
Figure 3 - Illustration of all interactions with Datto systems, which are mocked at this time.

The list of Datto device information is retrieved from the Datto system at a rate of once a day. After collaborating with the sponsor, it was decided that this timeframe was sufficient to check for any updates. An update could be a new device being added, a device being removed from Datto’s system, or a simple update such as a change in location or change in MSP ownership.

The team originally designed a different approach that would be responsible for device updates. An observer pattern was utilized, where the disaster tracker would register as an “observer” with Datto. When Datto performed a modifying action on their devices database, a notify action would be triggered that would notify all registered observers, including the disaster tracker. This was beneficial because it allowed for immediate synchronization across the two systems, and would minimize wasted calls and data transfer.

However, this approach also would have meant a much larger workload on Datto, and the sponsor indicated that the resources would not be available to implement this solution. Therefore, the team went with the backup design illustrated above (a simple pull model that runs once a day), and, as stated above, the sponsor was satisfied with this design choice.
3. Weather Information

As stated in the list of functional requirements, weather needs to be utilized for risk calculation. After considering many options to satisfy this requirement, the weather alerts provided by the National Weather Service were chosen. The National Weather Service is part of the National Oceanic and Atmospheric Administration (NOAA) government agency and is one of the most reliable and recognizable sources of weather information. All issued weather alerts across the United States are compiled into a single XML file hosted on the weather.gov website.

The first reason that this source was chosen for weather information was for its reliability. The National Weather Service is one of the leading weather sources in the United States, and the team actually found that other weather providers were pulling from the National Weather Service and simply reformatting the data. This solution was also free of cost and readily available for use by anyone.

The weather alert data is pulled at a rate of once every ten minutes, which was determined to be an adequate timeframe for the domain of weather updates and was approved by the sponsor. After the data is received and parsed, it is passed to the risk calculation algorithm to convert the weather alert to a numeric risk value.

4. Risk Calculation

The API is responsible for calculating a risk value from weather alerts. This is an in-depth topic, and is discussed below in the Risk Algorithm section.

5. Data Persistence

Data retrieved and calculated by the disaster tracker needs to be persisted. MySQL was chosen, largely because of its popularity and support. It also fairs well in terms of performance. The database design is relatively simple, and only consists of three entities: devices, cities, and a weather type.

A device contains information about its ID, location and MSP owner. It also maintains its current risk level that was calculated by the risk algorithm, and the cause of the risk if applicable. It has a foreign key to the city it belongs in.

A city contains the city name, state, county, and forecast zone that it belongs to. The county and forecast zone are needed to apply risk values to the devices. More information on this is discussed in the Risk Algorithm section.

A weather modifier simply correlates a weather type that is received in a weather alert to a numeric modifier. This is designed to be configurable. As more knowledge is gained about the
impact of a weather type on power outages, the values can be modified accordingly. More information on these values is discussed in the Risk Algorithm section.

The database acts as a sort of cache between Datto’s list of devices and the web/mobile interfaces. Although it may seem like poor design to have duplicated data between Datto’s database and the internal SQL database, this design actually makes sense for this project. For performance reasons, the weather to risk calculations are not done on-demand. Periodic calculations are performed instead on a set of known devices. Therefore, the set of devices returned by the API needs to be known, it cannot be volatile, otherwise the unexpected devices will have an unknown risk value. In other words, if an alternative design was chosen that simply relies on Datto’s device database, and Datto adds a device before the risk calculations have executed, the device will have an unknown risk until the calculations are run. The disaster tracker system needs to maintain its own list of devices and associated risk information.

6. Push Notifications

Sending push notifications to MSPs with an iOS device is one of the functional requirements and is performed in the API layer. For iOS devices, Apple has a push notification gateway. This can be a sizeable amount of work to interact with directly. The team considered a design that would directly interact with the gateway, but found multiple reasons to use an alternative service called Parse. There is a free tier that is being used at the time of this writing, but can be upgraded as Datto scales to a larger number of push notifications. The reasons for using Parse include reliability and maintainability. Parse is widely used, and correctly interacts with the gateway. This avoids the possibility of the senior project team accidentally using the gateway in an incorrect manner. If the gateway changes its interface or Apple introduces an update, Parse will be responsible for matching the gateway changes. The database design is also simplified as a result of this decision, as Apple device IDs do not need to be maintained and mapped to MSP IDs.

7. Exposing Device Information

Most importantly, the purpose of the API layer is to expose all necessary data to the interfaces. Specifically, device information needs to be exposed. This is accomplished through a RESTful design, meaning devices are represented as resources. The most relevant action being performed on a device is a simple retrieval, which is accomplished using the HTTP GET verb. A GET call to the /devices endpoint (assuming the session is already authenticated) will return a list of devices in JSON format. The POST and DELETE verbs are also implemented, but are intended to be protected behind a firewall. The use case for these verbs is for new or removed devices from the Datto database, following the observer pattern discussed in the Datto Device Information section above.
**Risk Algorithm**

A risk algorithm is necessary to produce numeric risk values for devices from weather information. Certain quality attributes were kept in mind while designing this algorithm, the most important of which being efficiency and accuracy.

At the time of this writing, Datto has sold over 60,000 devices across the United States. This means a naive risk algorithm that simply iterates through the list of devices one at a time, pulls weather information for the device’s location, and runs a conversion process would be impractical. A more efficient solution needed to be designed. Since weather information is coming from NOAA, the team decided to research how the National Weather Service issues alerts. NOAA recognizes two different types of areas: counties and forecast zones. Forecast zones resemble U.S. counties, but are smaller (for example, Northern and Southern Monroe County). Since weather alerts are issued using this zoning system, the senior project team decided to follow this design as well. Instead of making weather requests for each device, weather information is only needed for the forecast zones, and can then be applied to all devices within the zone.

Solving the accuracy concern takes a bit more work. Converting weather information to a risk value is not an easy task. For this project, the team defines risk as the chance of a power outage, since this is the most likely data loss cause that can be linked to a weather event. In the first release of the risk algorithm, risk was calculated by simply multiplying severity of the weather event by the certainty of the weather event. Severity ranges from minor to extreme, and certainty ranges from unlikely to observed. These values are given in the weather alert data that NOAA provides, so no real processing is done by the senior project team besides a simple multiplication equation.

In the next iteration of the risk algorithm, the team decided to consider the specific weather alert type in addition to the severity and certainty values. In other words, the team wanted to predict how much more likely it is that a hurricane will cause a power outage versus a fog weather alert. To accomplish this, predictive data modeling was used. In particular, a logistic regression model was developed based on historic data.

Two primary datasets were gathered. The first dataset consists of all power outage events reported by electricity providers across the United States for the last 15 years and their associated cause. Major electricity providers are required to report power outages to the United States Department of Energy (DOE) by filling out form OE-417 ("OE - 417 ELECTRIC EMERGENCY INCIDENT AND DISTURBANCE REPORT"). The DOE collects this information and publishes an annual report of the year’s events. This dataset compiles and standardizes 15 years of these annual reports into a CSV format. The second dataset consists of all issued weather alerts from NOAA over the last 15 years. This data is available for bulk download for every year dating back to 1950 in a CSV format. The last 15 years of data was collected and consolidated into one dataset for use comparing against the power outage dataset.
Next, the two datasets were combined by analyzing the times and locations of weather alerts versus power outages. The data was further normalized to a point where a logistic regression model could be formed.

A logistic regression was chosen for a few reasons. The dependent variable for this domain is whether or not a power outage will occur. This is a binary question; it can only take one of two values (yes or no). Therefore, a linear regression would be inappropriate since it produces a dependent variable that is continuous, and can take any value. A logistic regression solves this issue and constrains the output to a value between 0 and 1.

Logistic regressions are similar to linear regressions in that the independent variables still have coefficients. In this domain, the independent variables are the weather types (hurricane, fog, etc). Analyzing the coefficient of each independent variable gives an indication of how critical that weather type is in terms of causing a power outage. The team used these coefficients, in comparison to the other coefficients, to rank and determine the most important weather types.

A new concept, called a weather type modifier, was introduced in this next iteration of the risk algorithm. Each of the 58 weather types were given a numeric weather type modifier between 0 and 1.5. These values are configurable, so as more knowledge is gained, values can be fine tuned. The weather modifier is added to the risk multiplication equation. The final result is:

\[
\text{Risk} = \text{Severity} \times \text{Certainty} \times \text{Weather Type Modifier}
\]

Converting weather information to probability of a power outage is not a perfected field yet, but this equation provides a level of accuracy that the team and the sponsor are satisfied with.

**Process and Product Metrics**

For the project, the team at first employed sprint velocity, cyclomatic complexity, lines of code (LOC) per method, rate of requirements change, and estimation accuracy. The most important of these metrics were the sprint velocity and Estimation Accuracy. Since the team was using Scrum as the process methodology with two-week sprints, the sprint velocity helped to determine how many story points could get feasibly done. The average sprint velocity for the team ended up being 18 Story Points. When looking at all sprints, velocities ranged from 8 points to 23 points completed. These velocities greatly helped the team for the following sprints. However, the team still faced unforeseen blocks that reduced velocity. However, the team was able to successfully plan for sprints when it was known that there would be events that would impact the velocity. This can be seen in the sprints where there were breaks (Fall, Winter, Spring etc.) and department-required events like the Interim and Final Presentations. The team was able to look back at earlier velocities and use those to estimate how much could get done during these slower sprints. The use of an internal physical burn-down chart as well as Trello boards helped the team visualize progress.

The second important metric the team tracked was estimation accuracy. This metric was primarily used to track each member’s time put into the project, both estimated and actual.
Individually, each team member could see how they were doing in their own estimation of time they expected to put into the week/sprint versus how much time it actually took. If the estimations were higher than actual times, then they were overestimating, and perhaps the tasks were easier than expected or maybe not enough work had been taken up. On the other hand, if estimation was lower than the actual time, then they were underestimating, perhaps the tasks were much more difficult or unexpected blocks happened that impeded progress. This metric was helpful for sprint planning. If the sprint velocity was on the lower side and had stories that fell over, and the time spent was on the higher end, then this would be brought up in the sprint retrospective. Perhaps an adjustment needs to be made when estimating the next sprint.

The remaining metrics that were to be tracked either fell out after the first semester or did very little for the project. Rate of requirements proved very useful for the first semester. From the Rate of Requirements document, one can see there were requirement changes throughout the semester. While they lessened towards the end compared to the beginning, it was still something worth tracking. This metric worked well at that time because early on the team realized the requirements were not very well defined. This was an early indication that requirements could easily be added, removed and modified. This meant a high requirements volatility. Coupled with the fact that the team was using Scrum, the team knew it would have to adapt to any and all changes to requirements at any point of the project. So tracking the rate of requirements was a good indicator of the team’s communication with the sponsors regarding requirements elicitation and clarification. However, during the second semester, the metric fell out as by then the sponsors seemed to finally be concrete on requirements and development was well underway. Any changes or additions were more of suggestions and “nice to have” features rather than full on requirement changes. As such, the team decided the metric did not need to be tracked anymore.

The last two metrics that were planned to be tracked but ended up not really being used were cyclomatic complexity and LOC per method. Both were meant to be used as a means of keeping the codebase overall maintainable and ensure that good coding practices were followed. Originally, the team planned to use PHP Mess Detector (PHPMD) to keep tabs on the code and look out for possible things such as bugs, over complex methods, suboptimal code, etc. However, initial difficulties with the tool combined with the ever growing list of items with much higher priority caused the tool to be mostly forgotten. In the end, it did not really affect the team much as the code was well-maintained went through refactoring to ensure it was written as well as possible. The team did not need the tool to know when it was a good idea to refactor. This showed the team always kept good coding practices and maintainability in mind. After all, the product was planned to be integrated with an existing system, so the team needed to make sure that a well kept product was handed off.

Overall, the metrics helped show some of what went well and poorly throughout the project. The team’s estimation, velocity, and ability to adapt to changes throughout the project were an indication of what went really well. Despite the blocks and other issues that set the team back, all the requirements were met and the team fully handled the requirements volatility of the first semester. The team’s estimation accuracy for both the sprint velocity and individual Time
Tracking improved over the two semesters. Lower velocity sprints were usually caused by situations out of the team’s hand and not due to bad estimation. As for what went poorly or could have been improved, the team’s initial failure to fully elicit requirements perhaps led to a high volatility and changes up until the end of the first semester. The team learned to constantly and very frequently contact the sponsors outside of the weekly meetings. Also, initial story point estimates needed refinement, but these estimates improved over time as the team gained more experience.

**Product State at Time of Delivery**

Currently, all required functionality by Datto of the product has been completed to their satisfaction. The web application, the API, and the mobile application are completed and ready for handover to the sponsor.

The team had planned on adding pagination to the API as a way to handle returning large amounts of device data effectively. This feature resulted from the ever increasing number of devices owned by Datto. The API was expected to return a JSON list of all the devices with their specific information. However, as the number of devices grew closer to 100,000 it became increasingly difficult for the system to handle that many objects in memory. Thus, paging the returned devices was thought of to mitigate the memory issue. However, due to time and priority constraints it was decided to be left incomplete.

At the onset of the project, the team did not expect to have to design and implement an OAuth server or create an endpoint to retrieve Datto devices. However, due to timeline discrepancies it meant the creation of a mock Datto system that would provide the features needed from Datto.

With the mobile application, the team made a suggestion to Datto to add the ability for the customer viewing devices at risk to be able to easily and quickly share the information with others. This resulted in a share feature that allowed the customer to be able to email the device information and preparation steps to whomever the customer wanted.

While all the requirements are completed, the system is not in a production ready state. Datto will need to transition over to using their own OAuth implementation as well as their devices database before being able to fully launch this to production. The mobile application will also need to be transitioned from using development certificates and provisioning profiles to production certificates and provisioning profiles before it can be submitted to the Apple App Store.

**Project Reflection**

A very significant amount of work went into this project, and for that, the team is extremely proud. We faced our share of challenges, both technical and non-technical. Some of these challenges have already been discussed above, such as the difficulties in creating an accurate risk
algorithm, or the misaligned schedules between Datto and the senior project team. However, the team was successful at finding ways to overcome these challenges, and we would consider that one of our strong points. In the case of the risk algorithm, an extensive amount of research and data collection was used to achieve desirable accuracy levels. In the case of the incomplete Datto systems, the team implemented an entire mock stack that was not expected at the beginning of this project! However, we stuck to our morals and did things the right way versus doing things the easy way.

Looking back on these challenges and thinking about what we might do different, the complications with the sponsor again comes to mind. Perhaps better communication early on in the project could have indicated these schedule and resource conflicts earlier on. Improving communication is something the team definitely improved on over the year. We learned that we needed to adapt to the sponsor and their preferred method of communication. In particular, we found that lengthy formal emails were less effective than quicker informal messages over a medium such as Facebook chat. The sponsor was more apt to reply to these messages.

Another major lesson that was learned was more technical. Even over this relatively short term project, we learned that technical debt can build up quickly, and that quality and maintainability should always be focused on. As we started building our web application, the code base quickly became unmaintainable. Snippets of code were copy and pasted into a single HTML file without much regard to order or readability. As problems inevitably started to arise, it became impractical to try to sift through the conglomerate of code snippets to find and resolve the bug. The team devoted nearly a week simply refactoring this file into separate cohesive files that become much easier to maintain. The lesson learned, though, is to keep these concerns in mind from the outset of the project.

We also learned that a comprehensive test plan that is run early and often is extremely important. Unfortunately, we learned this lesson the hard way. While our backend code had nearly 100% unit test coverage throughout much of the project, our functional and stress tests were severely lacking for most of the project. Going into the last few weeks of the project, we discovered several major memory issues when testing the application with more than the 30,000 devices that had been the default for testing. While we had tested with up to 100,000 devices in the past, major changes to our underlying API caused significant memory increases when scaling up the number of devices. Unfortunately, these tests were not run on a regular basis and this issue slipped through the cracks until the last few weeks of development. This ended up being a big enough issue that it pushed our intended coding freeze back by a week. If we had been running a full suite of functional and stress tests every week, or even just when major changes were implemented, this issue could have been caught much earlier.
References